Adaptive Management of Multimodel Data and Heterogeneous Workloads

Inauguraldissertation

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von

Marco Dieter Vogt

Basel, 2022

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Heiko Schuldt
Erstbetreuer

Prof. Dr. Christian Tschudin
Zweitbetreuer

Prof. Dr.-Ing. Norbert Ritter
Externer Experte

Basel, 20.09.2022

Prof. Dr. Marcel Mayor
Dekan

Zusammenfassung

Datenverwaltungssysteme sehen sich einer wachsenden Nachfrage nach einer engeren Integration heterogener Daten aus verschiedenen Anwendungen und Quellen in Echtzeit für sowohl operative als auch für analytische Zwecke gegenüber. Die enorme Diversifizierung der Datenverwaltungslandschaft hat jedoch zu einer Situation geführt, in der zwischen hoher Betriebsleistung und enger Datenintegration abgewogen werden muss. Der Unterschied zwischen dem Wachstum des Datenvolumens und dem Wachstum der Rechenleistung erfordert einen neuen Ansatz für die Verwaltung von Multi-Modell-Daten und die Verarbeitung heterogener Arbeitslasten.

Mit PolyDBMS präsentieren wir eine neue Klasse von Datenbankmanagementsystemen, welche die Lücke zwischen Multimodell-Datenbanken und Polystore-Systemen schliesst. Diese neue Art von Datenbanksystem kombiniert die operativen Fähigkeiten von traditionellen Datenbanksystemen mit der Flexibilität von Polystore-Systemen. Dazu gehört die Unterstützung von Datenmodifikationen, Transaktionen sowie Schemaänderungen zur Laufzeit. Mit der nativen Unterstützung für mehrere Datenmodelle und Abfragesprachen stellt ein PolyDBMS eine ganzheitliche Lösung für die Verwaltung heterogener Daten dar. Dies ermöglicht nicht nur eine enge Integration von Daten über verschiedene Anwendungen hinweg, sondern erlaubt auch eine effizientere Nutzung von Ressourcen. Durch die Nutzung und Kombination von hochoptimierten Datenbanksystemen als Speicher- und Ausführungseinheiten profitiert diese neuartige Klasse von Datenbanksystemen von der jahrzehntelangen Forschung und Entwicklung im Datenbankbereich.

In dieser Arbeit stellen wir die konzeptionellen Grundlagen und Modelle für den Aufbau eines PolyDBMS vor. Dazu gehört ein ganzheitliches Modell zur Verwaltung und Abfrage mehrerer Datenmodelle in einem logischen Schema, welches zugleich modellübergreifende Abfragen ermöglicht. Mit der PolyAlgebra präsentieren wir eine Lösung zur Repräsentation von Abfragen, welche auf einem oder mehreren Datenmodellen basieren, wobei die Semantik dieser Datenmodelle erhalten bleibt. Darüber hinaus stellen wir ein Konzept für die adaptive Planung und Zerlegung von Abfragen auf heterogenen Datenbanksystemen mit unterschiedlichen Fähigkeiten und Eigenschaften vor.

Die in dieser Arbeit vorgestellten konzeptionellen Beiträge materialisieren sich in Polypheny-DB, der ersten Implementierung eines PolyDBMS. Polypheny-DB unterstützt das Relationale-, Labeled-Property Graph- und Dokumenten-Datenmodell und ist damit eine geeignete Lösung für strukturierte, halbstrukturierte und unstrukturierte Daten. Ergänzt

wird dies durch ein umfangreiches Typensystem, welches auch Unterstützung für grosse Binärobjekte bietet. Durch die Unterstützung mehrerer Abfragesprachen, standardisierter Abfrageschnittstellen und einer Vielzahl von domänenspezifischen Datenspeichern und Datenquellen bietet Polypheny-DB eine Flexibilität, welche mit bestehenden Datenmanagementlösungen nicht erreicht werden kann.

Abstract

Data management systems are facing a growing demand for a tighter integration of heterogeneous data from different applications and sources for both operational and analytical purposes in real-time. However, the vast diversification of the data management landscape has led to a situation where there is a trade-off between high operational performance and a tight integration of data. The difference between the growth of data volume and the growth of computational power demands a new approach for managing multimodel data and handling heterogeneous workloads.

With PolyDBMS we present a novel class of database management systems, bridging the gap between multimodel database and polystore systems. This new kind of database system combines the operational capabilities of traditional database systems with the flexibility of polystore systems. This includes support for data modifications, transactions, and schema changes at runtime. With native support for multiple data models and query languages, a PolyDBMS presents a holistic solution for the management of heterogeneous data. This does not only enable a tight integration of data across different applications, it also allows a more efficient usage of resources. By leveraging and combining highly optimized database systems as storage and execution engines, this novel class of database system takes advantage of decades of database systems research and development.

In this thesis, we present the conceptual foundations and models for building a PolyDBMS. This includes a holistic model for maintaining and querying multiple data models in one logical schema that enables cross-model queries. With the PolyAlgebra, we present a solution for representing queries based on one or multiple data models while preserving their semantics. Furthermore, we introduce a concept for the adaptive planning and decomposition of queries across heterogeneous database systems with different capabilities and features.

The conceptual contributions presented in this thesis materialize in Polypheny-DB, the first implementation of a PolyDBMS. Supporting the relational, document, and labeled property graph data model, Polypheny-DB is a suitable solution for structured, semi-structured, and unstructured data. This is complemented by an extensive type system that includes support for binary large objects. With support for multiple query languages, industry standard query interfaces, and a rich set of domain-specific data stores and data sources, Polypheny-DB offers a flexibility unmatched by existing data management solutions.

Acknowledgements

This dissertation marks the end of my time as a student. Since I started as an undergraduate almost 10 years ago, I have had the privilege to meet many great people. Though I cannot mention them all by name here, I am grateful to everyone who accompanied me on this journey. I would, however, like to express my gratitude to a few particular people.

First and foremost, I would like to thank my advisor **Prof. Dr. Heiko Schuldt** for his guidance and support over the past years and for giving me the opportunity to obtain a PhD. I am deeply grateful for his tireless commitment and the confidence he has placed in me.

I would like to thank **Prof. Dr. Christian Tschudin** for his willingness to be my second supervisor. Moreover, I would like to thank **Prof. Dr.-Ing. Norbert Ritter** for reviewing this dissertation as an external expert.

During my PhD, I had the great pleasure of working with many wonderful colleagues, many of which I am happy to call my friends. I would especially like to thank **Ralph Gasser** and **Silvan Heller**. Our thesis writing group was a great opportunity for mutual motivation and provided me with very valuable feedback on my dissertation. I also wish to express special thanks to **Sein Coray** for his contributions to Chronos. Additionally, I would like to express my sincere gratitude to **Dr. Luca Rossetto**, **Loris Sauter**, and **Alexander Stiemer** for their commitment in maintaining our research infrastructure, a service for which they are not thanked enough. Furthermore, I would like to thank the following past and current colleagues for the many fruitful discussions and never-ending support: **Dr. Ivan Giangreco**, **Ashery Mbilinyi**, **Dr. Mahnaz Parian-Scherb**, **Dr. Lukas Probst**, **Dina Sayed**, **Dr. Christopher Scherb**, **Dr. Philipp Seidenschwarz**, **Shaban Shabani**, and **Florian Spiess**.

As part of my PhD, I had the opportunity to supervise some extremely talented students. Polypheny-DB and the Polypheny ecosystem shine thanks to their valuable contributions. I would especially like to thank **Isabel Geissmann**, **Nils Hansen**, **Marc Hennemann**, and **David Lengweiler**. Additionally, I would like to thank everyone who has contributed, in particular **Khushbu Agrawal**, **Colin Fingerlin**, **Flurina Fischer**, **Christian Frei**, **Luc Heitz**, **Jannik Jaberg**, **Phaina Koncebovski**, **Cédric Mendelin**, **Nicolas Odermatt**, **Sebastian Philipp**, **Jonas Rudin**, and **Jan Schönholz**.

I would also like to thank **Dr. Heike Freiberger**, who is the best coordinator of a PhD program one could imagine. Her commitment is highly appreciated. In addition, I would like to thank the staff who keep the Department of Mathematics and Computer Science running, especially the sysadmins and the administrative staff.

I would also like to thank all my friends for their support and understanding, especially over the last six months. In particular, I would like to thank the following four people who have not yet been mentioned: **Monika Nagy-Huber**, **Felix Portmann**, **Laurin Stock**, and **Claudio Torello**.

Finally, I would like to express a deep sense of gratitude to my family for their unwavering support and continuous encouragement throughout my years of study and the writing of this thesis.

This work was partly supported by the Swiss National Science Foundation in the context of the Polypheny-DB project, contract no. 200021_172763, which is also gratefully acknowledged.

Contents

List of Figures

Part I

Introduction and Motivation

Data is a precious thing and will last longer than the systems themselves.

— Tim Berners-Lee

1 Introduction

In Plato's dialogue "Phaedrus", Socrates tells the story of Thoth, the god of arts and sciences, and how he presents his innovation of writing to the god-king Thamus. Thoth argues that this *elixir of memory and wisdom* will make Egyptians wiser, as they will have better capacity to remember information.

Storing and persisting information is a key factor of human development [BW06]. The oldest cave paintings created by ancient humans to store information are over 40 000 years old [BOB+21]. Since then, the Homo sapiens has developed more efficient techniques for storing information. The development of the transistor in 1947 marked the beginning of a new era (not only) for the storage of information.

We humans are the only species on this planet capable of persistently storing and preserving information—and we like doing so. According to Forbes [Pre20], the total amount of data increased from 1.2 zettabytes (ZB) in the year 2010 to 59 ZB in 2020. This is an almost 5 000% growth in just 10 years. In this era of *Big Data*, the ability to efficiently find the famous needle in the haystack is more relevant than ever.

Database Management Systems (DBMS) are a means to organize and manage this growing haystack of information. They store the data (i.e., the information) in an organized and efficient form and enable it to be modified and queried. Database systems are at the heart of almost all data processing systems, making them an essential part of most IT infrastructure.

The increasing importance of large amounts of data for progress in all disciplines of science and business demands for faster and more efficient approaches to storing and querying large data collections. The discrepancy between the growth in data volume and the growth in computing power (cf. Moore's law) calls for new solutions.

1.1 The Data Management Landscape

In 1970, Edgar F. Codd published a paper with the title "A Relational Model of Data for Large Shared Banks" [Cod70]. It introduces an abstract model for data representation, independent of how the data is physically stored. This paper laid the foundation for the field of *relational databases*. Four years later, as part of a research project, IBM started the development of *System R* [CAB+81]. The developed prototype did not only demonstrate that relational databases are capable of providing good transactional performance, but it also introduced the *Structured Query Language (SQL)*. This marked the beginning of the era of relational databases with SQL as their standard query language [Bat18].

In the 2000s, the World Wide Web gained increasing importance. Database systems needed to cope with new and rapidly growing workloads. At the same time, storage cost dramatically decreased [Pro16]. The high efficiency in terms of storage space achieved by relational databases was no longer of utmost importance. Instead, the demand for storing less structured data with maximum performance grew. This led to the development of new kinds of database management systems, commonly referred to as *Not only SQL (NoSQL)* systems [EN16].

These NoSQL systems did no longer follow the relational model, instead several new data models for representing data emerged (e.g., graph [AG08] or document [HR16]; see Chapter 4). In contrast to previous systems, these new database systems were typically developed for a specific use case. As their name implies, most of these databases come without support for SQL. Instead, they often come with their own query language tailored to and only supported by this specific database system. Furthermore, most NoSQL systems also have no or only very limited support for transactions and provide only weak data consistency guarantees [GWF+17; ZSL+14]. However, this reduction to the set of features required for a certain application scenario reduces overhead and enables more optimized data structures. This results in a better performance for this specific use case. The performance for other workloads and applications is either significantly lower or the required operations are not even supported at all (e.g., joins) [ZSL+14].

This led to the diversification of the data management landscape we are facing today. A development already predicted in 2005 by Michael Stonebraker and Uğur Çetintemel in their famous "One Size Fits All" paper [SÇ05]. They argue that the general-purpose approach of relational databases will no longer be sufficient. However, they consider that some of these specialized databases might get unified beneath "a common front-end parser" [SÇ05].

1.2 Optimization vs Data Integration

Data is a precious good. When properly connected, it allows deriving all kinds of insights. With the field of *Data Science*, an entire discipline has grown around the effort of extracting (generalizable) knowledge from data [Dha13]. Turing Award winner Jim Gray even refers to data science as the fourth paradigm of science—the other three being empirical, theoretical, and computational [HTT09].

In line with the spirit of having a specialized database for every task, several analytical databases saw the light of day in the last two decades [ZSL+14]. However, the data from the heterogeneous operational databases needs to be transformed and copied into the analytical database(s). The resulting system is called a *data warehouse*. The frequency at which this integration is done is called *refresh interval* [JLV+00]. It determines how recent the data is. Since refreshing larger data warehouses can take hours and also causes additional load on the operational databases [SB08], keeping data warehouses always up-to-date is not feasible. Results derived from a data warehouse are therefore often outdated to a certain degree [ZK01]. For many use cases, a certain degree of outdatedness is acceptable [ASS08]. However, with businesses getting more agile, there is a growing need for business information systems operating on the latest data [ZK01].

Besides dealing with outdated results due to time-consuming integration processes, there is another issue caused by the growing zoo of specialized database systems: redundant and inconsistent information. Instead of having one source of information for all applications, there are several heterogeneous databases containing overlapping data. In order to fulfill the demand for a tight coupling of applications and a full integration of all available data [Sto15], an application needs to process and update data from multiple heterogeneous databases. The price for the operational performance gained by using specialized database systems is therefore paid by a huge overhead and complexity for integrating applications and keeping the data consistent. The fact that the database systems may use different query languages and may be based on different data models, makes this even more challenging.

An example are medical data corpora [JBP+]. The medical record of a patient stored by a hospital consists of various kinds of structured and unstructured data including text (e.g., notes and examination results), images and videos (e.g., MRI or CT scans), structured documents (e.g., blood tests), and time series data (e.g., electrocardiograms). Furthermore, there are also invoicing data, payment histories and insurance information. Manually transferring data between different systems and exporting them to a central document

storage (often in form of a document or report) introduces the potential for human errors, for instance, attaching them to the wrong patient. Moreover, maintaining patient information in several independent applications (e.g., the software of the CT scanner) causes administrative overhead and adds further potential for mistakes. Enabling a unified view on all available information and data points does not only provide physicians with a more holistic view, it also enables automated detection of anomalies and patterns in the raw data of different examinations. Additionally, it allows searching for similar cases taking all available data points into account.

1.3 The Case for a Polystore System?

The "One Size Fits All" article [SÇ05] from Michael Stonebraker and Uğur Çetintemel can be seen as the starting point for the development of the so called *multistore* and *polystore* systems. These are systems combining multiple heterogeneous data stores beneath one facade (i.e., "a common front-end parser" as proposed in the article).

Multistores and polystores aim to solve a similar issue as data warehouses: providing a unified view on data distributed across multiple heterogeneous databases [Sto15]. However, instead of building a data repository by copying data from the different locations and integrating them into a unified schema, they use an approach already known from *federated databases* [HM85]. A federated database system maps multiple database systems into one schema without copying the data first. Instead, queries are translated into the query language of the database where the requested data is physically located.

Unfortunately, there is no commonly agreed upon definition of what exactly constitutes a federated database system, a multistore, or a polystore, respectively. In this thesis, we follow the taxonomy introduced in [TCG+17]. According to this taxonomy, federated databases enable querying multiple *homogenous* database systems through one query interface using one query language. Multistore systems, in contrast, also enable the federated access to multiple *heterogeneous* databases through one query interface using one query language. Polystore systems go a step further by applying the idea of *polyglot systems* to multistore systems [TCG+17]. The idea of polyglot systems is to choose the right tool (i.e., query language) for a concrete use case (i.e., application)[1]. In comparison to multistore systems, polystores therefore also offer support for multiple query languages and interfaces. Turing Award winner Michael Stonebraker argues that polystores are

[1] This should not be confused with the term *polyglot persistence* that is usually used from an application perspective and describes the usage of different data storage technologies within the same application; i.e., an application using multiple database systems.

stepping up as successor for the unsuccessful federated databases and seek to replace data warehouses [Sto15].

At a first glance, the ability of polystore systems to query heterogeneous database systems using different query languages seems to be a suitable solution for achieving a tight integration of applications and data while at the same time exploiting the benefits of optimized databases. However, this is only the case for the analytical portion of the workload. In order to achieve the tight integration of applications and data, transactional workloads need to be handled as well. In addition to competitive performance for simple, short running queries, this especially means support for data manipulation (DML) operations. However, most existing polystores do not support DML operations at all [LHC18a]. Another important feature for handling operational workloads are transactions. This includes providing transactional guarantees (i.e., ACID guarantees [HR83]) and the ability to enforce constraints. These are, however, features not available in existing polystore implementations [VHS+20; LHC18a].

Hybrid transactional/analytical processing (HTAP) systems like SingleStore[2] typically provide support for data manipulation and transactions. By combining a transactional engine (typically with a row-oriented storage layout) and an analytical engine (typically with a column oriented engine) in one system, they can provide good performance for both transactional and analytical workloads. This approach can be seen as the continuation of the "One Size Fits All" paradigm. However, these systems usually require selecting the storage engine (row store or column store) when creating the table. This limits the advantages for mixed workload on the same data. Furthermore, those systems are typically limited to a single data model (usually the relational model) and only support one query language. While some systems support storing other data models, the data is usually mapped to the primary data model; thus only the advantages of one data model (typically the relational model) can fully be exploited.

1.4 Multimodel Databases

The idea of a DBMS supporting multiple data models is an emerging topic in the area of database management systems. Such systems are called *multimodel databases* [LH19]. In contrast to database systems based on a single data model, multimodel database systems offer much more flexibility for representing and processing heterogeneous data.

[2] https://www.singlestore.com/

In recent years, most major database systems have been extended to support additional data models [LH19]. This has usually been done by adding additional layers above the existing storage engine. Graphs in SAP HANA [FCP+12], for instance, are internally mapped to relational column storage [RPB+13]. This can impact the query performance compared to a storage model optimized for a specific data model [WPW+14]. Furthermore, multimodel databases suffer from the fact that they compete against highly optimized systems with—in some cases even decades—of optimization and development to efficiently support a specific data model.

In comparison to polystore systems, most existing multimodel database systems only support a single query language. In a recent survey on multimodel databases [LH19], 23 systems have been analyzed and compared: 20 of these systems support only one query language, and the remaining three systems support two query languages. Another difference between polystores and multimodel databases is their intended application: while multimodel databases enable the efficient representation and processing of heterogeneous data, polystores are intended to replace data warehouses and process analytical workloads.

1.5 Focus of Research

The enormous diversification of the data management landscape has led to a situation where there is a trade-off between high operational performance and a tight integration between different applications. Due to the growing discrepancy between the growth of data and growth of computational power on one side, and more agile business and workflows on the other side, the need for bridging this gap and providing a solution that allows a tight integration of heterogeneous data without sacrificing the performance of the queries or the freshness of the results is of increasing importance [VHS+20].

Since neither existing HTAP systems, multimodel database systems nor polystore systems present a satisfying solution for handling the data and workloads resulting from the tight integration of data between different applications, there is the need for a new kind of database management system. This system needs to be capable of handling the demand outlined by our research question:

> *How can the demand for tight integration of heterogeneous data based on different data models be met with competitive overall cost and performance while allowing consistent manipulation and retrieval of data with mixed, parallel workloads using different query languages and across data models?*

There are three major, yet to be solved, challenges arising when creating a database system that aims to solve the demand outlined in the research question:

- The lack of a unified and holistic model for combining different data models in one logical schema that preserves the semantics of the individual models while at the same time enabling cross-model queries.

- The lack of a conceptual model for representing queries expressed in different query languages and based on different data models that preserves the features and characteristics of the query languages and enables cross-model queries.

- The challenge of adaptively planning the decomposition of queries and their assignment to execution engines to exploit the benefits of the heterogeneous execution and storage engines.

The focus of this doctoral thesis is on presenting a conceptual approach for addressing these three research objectives. Our research is driven by empiricism and has a strong focus on the feasibility and the implementability of the developed concepts in one holistic system. Hence, in addition to the formal presentation and explanation of the conceptual models provided in this thesis, all concepts are also fully implemented and thoroughly tested in a single, fully working system.

Implementing such a system results in numerous engineering challenges. For many individual issues, like, for instance, distributed transactions and concurrency control, there is already preliminary work in the literature. However, the challenge is adapting and adjusting these existing pieces from decades of database systems research and combining them in one system. The result of this endeavor is *Polypheny-DB*. Using this system as an example, we are able to present solutions for the engineering challenges, and more importantly, demonstrate the feasibility and effectiveness of the presented concepts in solving the research question.

Within our research, we put a strong focus on evaluations and the reproducibility of results. An automated and reliable benchmarking of the Polypheny-DB system allows us to gain insights by testing the behavior and measuring the impact of different approaches or certain features on the performance. Furthermore, we consider verifying the correctness of the implementation a fundamental requirement to draw conclusions about the holistic picture and to obtain meaningful benchmarking results.

1.6 Contributions

The contributions of this doctoral thesis can be summarized as follows:

- We propose the specifications for a new type of database management system. These *PolyDBMS* bridge the gap between polystores and HTAP systems and provide the full capabilities of multimodel database management systems. To distinguish a PolyDBMS from existing approaches, we introduce clear and concise requirements. *(→ Chapter 3 & Chapter 7)*

- We propose a holistic model for maintaining and querying multiple data models in one logical schema. Furthermore, we introduce a model for combining data partitioning and data replication in a PolyDBMS system. This is further extended by a model that defines the mapping between different data models. *(→ Chapter 8)*

- We present a solution for representing heterogeneous queries using a single algebra. This *PolyAlgebra* allows representing queries expressed in various query languages based on different data models while preserving the semantics of the data model. Furthermore, it enables cross-model queries. *(→ Chapter 9)*

- We introduce an adaptive approach for planning the execution of queries in systems combining data replication and partitioning across heterogeneous database systems with different capabilities and features and for mixed workloads. This includes decomposing queries and selecting the underlying databases with the best characteristics for executing the query. *(→ Chapter 10)*

- With Polypheny-DB, we present a fully working implementation of the introduced concepts. This PolyDBMS provides the full capabilities of a database management system including schema definition and modification at runtime, support for data modification queries, and transactions. *(→ Chapter 11)*

- We introduce a novel kind of evaluation framework for automating the entire evaluation workflow. By storing all parameters and controlling the whole setup procedure this system makes an important contribution towards reproducible evaluations. *(→ Chapter 12)*

- We present and discuss an evaluation of Polypheny-DB and the introduced concepts using different benchmarks. This evaluation is done using qualitative and quantitative methods. *(→ Chapter 13 & Chapter 14)*

1.7 Outline

This thesis is structured in five high-level parts: an introductory part; a part introducing the foundations; a conceptual part presenting the developed models; an empirical part presenting the evaluation; and a discussion-oriented part summarizing related work and concluding this thesis. In more detail, the reminder of this thesis is structured as follows:

In **Chapter 2**, we introduce a scenario motivating the need for a new kind of database management system. The introduced scenario based on a fictive online auction house will furthermore serve as a use case and example throughout this thesis.

In **Chapter 3**, we introduce a set of specifications and requirements a PolyDBMS needs to fulfill. Furthermore, we introduce and discuss potential architectural models for implementing a PolyDBMS. The chapter, and thus the introductory part of this thesis, concludes with an overview of the research objectives that constitute the basis for the research results presented in this thesis.

In **Chapter 4**, we give an overview on the relational data model, the document data model and the labeled property graph data model. For each data model, a formal model is being introduced. This includes the representation of data, basic operations on the data, and the available building blocks for defining schemas and imposing constraints.

In **Chapter 5**, we discuss the foundations of database systems. This includes their basic building blocks, query optimization, and concurrency control. Furthermore, we introduce the concept and role of transactions in a database system.

In **Chapter 6**, we introduce and discuss the necessary foundations of distributed systems. This includes data partitioning and replication as well as distributed transactions and a discussion of data freshness.

In **Chapter 7**, we discuss the architecture and outline the arising challenges of designing a PolyDBMS. In addition, we present the rationale for the selection of data models that will be used to exemplify the conceptual models.

In **Chapter 8**, we present a schema model for representing schemas based on different data models in one logical schema. Furthermore, we define how schemas can be mapped between different logical schema models and on different physical schemas. The introduced model for combining data replication and data partitioning brings both parts together and presents a novel concept for the efficient, cost-effective and performant management of heterogeneous data.

In **Chapter 9**, we introduce the *PolyAlgebra*, a concept for representing queries based on different data models. Instead of mapping to one data model, this approach fully preserves the semantics of the query and its data model. The physical algebra allows expressing queries executed on multiple heterogeneous data stores and allows such queries to be optimized.

In **Chapter 10**, we present our *four phase query routing* approach. The conceptual model introduced in this chapter presents an adaptive solution for efficiently planning the execution of queries across heterogeneous systems.

In **Chapter 11**, we give a brief overview of our implementation Polypheny-DB, including a high-level overview on its architecture and supported features.

In **Chapter 12**, we present *Chronos*, a system for the automation of the entire systems evaluation workflow. Initially developed for the benchmarking of Polypheny-DB, Chronos can be used to benchmark all kinds of systems.

In **Chapter 13**, we introduce an approach for verifying the correctness of a PolyDBMS using randomly generated queries and comparing their result for different storage configurations. This allows us to perform a qualitative evaluation of Polypheny-DB.

In **Chapter 14**, we present the results of the quantitative evaluation of the concepts presented in this thesis based on our implementation Polypheny-DB. For this evaluation, we use both custom and industry-standard benchmarks.

In **Chapter 15**, we discuss related research and existing database systems in the light of the PolyDBMS concept. This overview includes both commercial database systems and research prototypes, especially polystore systems.

In **Chapter 16**, we conclude this thesis by wrapping up the presented concepts and discussing whether our implementation meets the requirements of a PolyDBMS. The chapter—and thus this thesis—closes with an outlook on future work.

Storytelling is the most powerful way to put ideas into the world.

— Robert McKee

2

The Gavel Scenario

In this chapter, we introduce the scenario of a fictional online auction house called *Gavel*. The described scenario is an extended version of a scenario already introduced in [VSS17; VSS18]. Using Gavel as an example, we motivate the need for a new kind of data management system. Moreover, this scenario will also serve as a use case and example in the following chapters of this thesis.

2.1 The Scenario

The fictitious auction house Gavel is a big player in the market of online auction houses, handling millions of auctions every day. Their business model is to offer a platform for buying and selling used, custom-made, or antique goods offered in the form of short running auctions. Thus, every item sold on the platform is unique and is therefore described by a text and pictures. Besides the infrastructure for offering items and placing bids, Gavel is also taking care of the whole payment process.

As is common for this kind of platform, Gavel charges the seller a percentage of the value for which the item has been sold. Gavel therefore has an interest in achieving the highest possible price for every item in order to maximize their profit.

Research on user behavior has revealed a direct correlation between the performance of an e-commerce platform and the conversion rate (i.e., the percentage of visitors buying something) [SN18]. For Gavel, this corresponds to the number of bids on an auction. However, this only affects active auctions. While completed auctions remain available for legal and transparency reasons, loading time for these outdated items is of minor importance. It is furthermore a characteristic feature of auctions that they are most interesting and also get clicked the most just before they end [SRJ07].

On their platform, customers are presented a list of offerings similar to the one they are currently looking at or have looked at. The list of offerings is provided by a *product recommendation system* [KT18]. However, an issue with these systems is, that they typically work on a dump of the product data created on a regular basis. Since Gavel's business model is to host short running auctions, this is problematic since it misses recently added items. Furthermore, it should also stop advertising products similar to something the customer just bought. The customer should not get the feeling that there might have been a better deal.

Like for every other enterprise, business analytics are an important technique for Gavel to steer and adapt the company. Detecting trends and adjusting the advertising and marketing strategy is crucial to the continued success of Gavel. However, the required ETL process for refreshing the data warehouse used for this purpose is very time-consuming and puts significant load on the operational databases [SB08; ZK01]. The data warehouse can therefore only be refreshed with a low frequency and is therefore usually outdated and does no longer meet the requirements of an agile and competitive market.

2.2 Infrastructure and Requirements

The software stack used by Gavel consists of multiple applications. Most applications used by Gavel are off-the-shelf products that come with their own database system. Gavel's in-house IT only develops the website that interacts with these applications through APIs. It is therefore not possible to change the implementation of the applications.

Figure 2.1 depicts an overview of the applications and database systems currently deployed by Gavel. The **Auction System** manages the whole auction and bidding process. It stores all auctions and bids in a relational database. Furthermore, this application maintains the user accounts of the customers.

All information, descriptions and images on the subject of the auction are stored in the **Auction Item Catalog**. This application stores its data in a document database. This allows storing and querying semi-structured data, such as product property lists, for all kinds of products.

The **Payment System** processes payments. It is independent of the *Auction System*. The information on transactions and the payment status is stored in a transactional database system. The payment system maintains its own user database. This is redundant and often introduces issues that require manual resolutions.

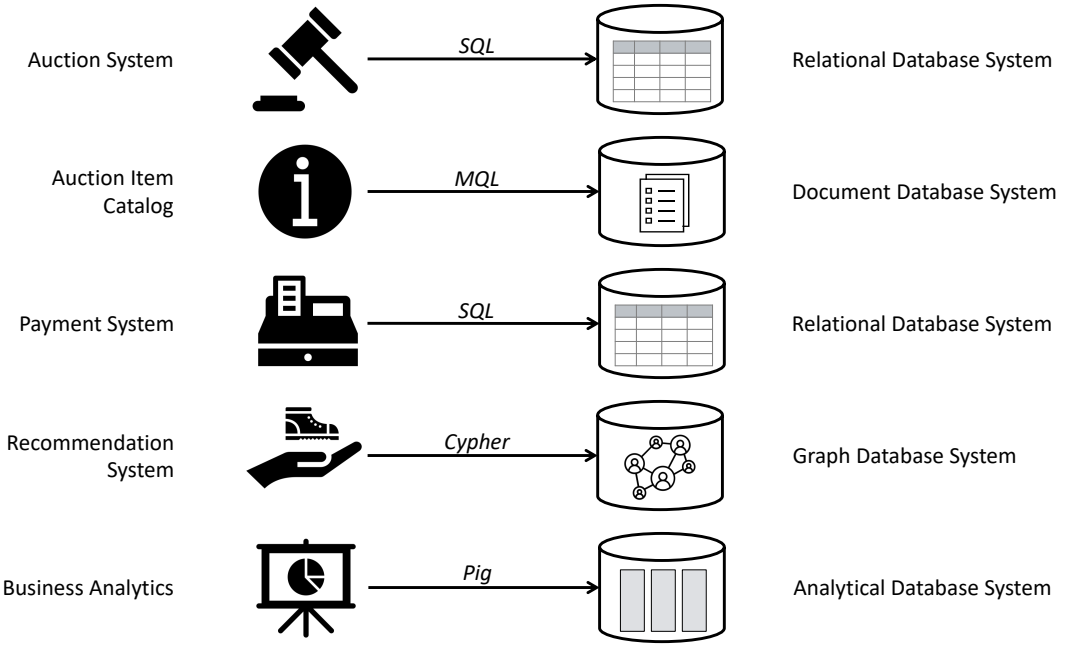


Figure 2.1 The current state of Gavel's application and data storage landscape. The applications store their data in independent database systems based on different data models and optimized for different kinds of workload. Each database system requires a specific query language.

The list of similar offerings is compiled by the **Recommender System**. It operates on data exported from the *Auction System* and the *Auction Item Catalog* at a certain interval. This data is stored together with user tracking data in a graph database. The integration process for building this database takes several hours. A significant portion of the auctions has therefore already ended before this integration is completed.

Business Analytics provides valuable information on trends and allows to identifying opportunities for adapting the business. It operates on a data warehouse with data aggregated from the *Auction System*, the *Auction Item Catalog* and the *Payment System*. The refreshing of this data warehouse takes a lot of time and also puts a significant load on the operational databases [SB08]. It is thus done very infrequently. Decisions of the company are therefore based on outdated data and may miss out on emerging trends.

Besides the already mentioned issues and limitations, this fragmented and independent storage of the data also hinders the implementation of new ideas. One such idea is a more intelligent advertising of auctions that are about to end. Offerings performing poorly compared to similar ongoing auctions (i.e., where the highest bid is lower) should be advertised more aggressively. This requires combining data from the *Auction System*, the

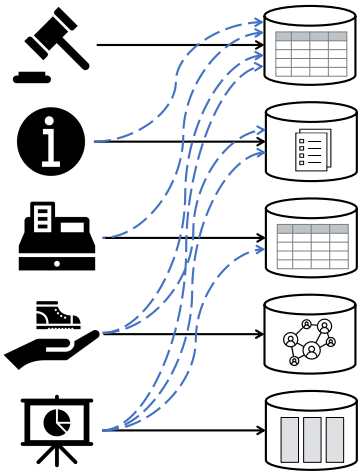


Figure 2.2 The required access paths between applications and database systems to enable the outlined features and providing a tight integration of applications and data using polyglot persistence.

Auction Item Catalog and the *Payment System*. Since this can only be done with the latest data, a data warehouse with its time-consuming data integration process cannot be used.

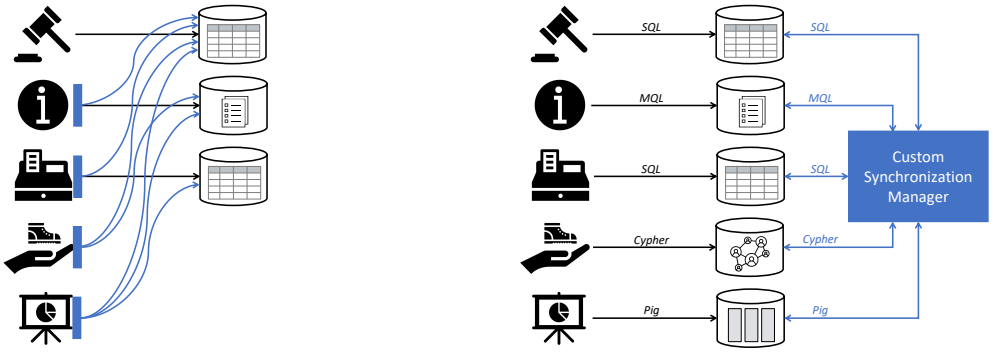
To address these limitations and implement new and innovative features, Gavel searches for a solution that enables a tighter integration of the data from different applications. This is depicted in Figure 2.2. However, the IT department of Gavel does not have the capacity to re-develop the whole stack in-house. It is therefore required to find a solution that allows keeping the existing third-party applications. Hence, changes need to happen entirely on the data storage and management side.

2.3 Evaluation of Existing Approaches

In this section, four potential approaches for implementing the requirements of the Gavel scenario are being discussed. For each approach, we outline the advantages and disadvantages and point out why it is not feasible within the constraints of this scenario.

2.3.1 Adjust Applications

A potential approach would be to adjust the applications to query other databases. This is depicted in Figure 2.3(a). An advantage of this approach is, that it always operates on the latest data and does not introduce any overhead for transactional queries. However, there are several aspects that make this approach unfeasible.



(a) **Adjusting applications and adding support for additional query languages to directly query and update other databases.**

(b) **Synchronizing data between the individual databases using a custom-built synchronization system.**

Figure 2.3 **Possible approaches for implementing the demands of the Gavel scenario that keep the existing database systems.**

First and foremost, there are queries joining data from different databases. These joins need to be implemented in the applications, which can lead to massive amounts of data that need to be transferred and processed within the applications. Another issue is analytical queries: Executing such complex queries on database systems not optimized for this kind of workload can take huge amounts of time and causes significant load on the operational databases—or may even block them entirely depending on their query isolation model.

Furthermore, changing an application is usually not possible since the source code is not available. But even in case the source code were available, performing the changes would result in losing the ability to directly deploy updates of the software released by the manufacturer. The IT department of Gavel would need to apply the necessary adjustments to every release of the software. Depending on the size and complexity of these adjustments, the required labor quickly gets similar to an in-house development of the entire application stack—for which Gavel does not have the capacity.

2.3.2 Data Synchronization

The idea of the approach depicted in Figure 2.3(b) is to synchronize data between the existing database systems shipped with the applications. This would allow keeping all database systems; including the two data warehouses and would solve one of the issues with the previous approach, since analytical queries could be executed on database systems optimized for this kind of workload. However, this approach is quite similar to the current infrastructure and therefore comes with the same challenges and disadvantages.

Identifying changes that need to be replicated to other databases can be complex and results in significant load on the operational databases [SB08]. Depending on the database system, this can, however, be solved using triggers. Nevertheless, what cannot be solved is the lack of global transactions. If the same record is changed in multiple places between two synchronization jobs, there might be conflicts.

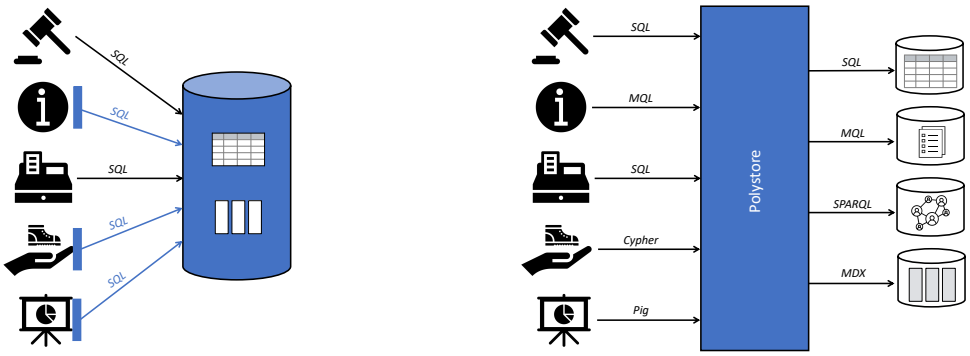
Besides the costs and issues with synchronizing and redundantly storing data on multiple databases, this approach does furthermore not compensate for missing features. If a database does not support a required operation or is not optimized for a certain workload, the performance will be poor. Furthermore, the data is always slightly outdated, which can lead to issues with certain applications that require a high degree of freshness (e.g., the payment system determining the highest bid when a customer pays directly after the auction has ended).

2.3.3 An HTAP or Multimodel Database

As introduced in Chapter 1, an HTAP system is a database management system optimized for transactional and analytical workloads. Usually, this is achieved by storing the data in two or more storage engines optimized for different workloads. Figure 2.4(a) depicts how such a deployment would look like for Gavel. Instead of keeping the databases shipped with the off-the-shelf applications, they are replaced by an HTAP system storing the data for all applications.

The capability of an HTAP system to provide good performance for OLTP and OLAP workloads makes a dedicated data warehouse obsolete [HLC+20; Ell14]. Storing all data in one database system also eliminates redundancies. Most available HTAP systems also support transactions and a large set of query features and operations. However, this approach suffers from the same issues as the first approach: it is necessary to adjust the applications to use the query language and data model of the HTAP system. As already pointed out, this is not feasible.

Multimodel database systems allow to store data represented using different data models. However, as outlined in Section 1.4, existing systems rarely support more than one query language [LH19]. Using a multimodel database therefore still requires adjustments to the applications. In contrast to HTAP systems, multimodel databases typically do not support storing data in different physical layouts at the same time. This reduces the eligibility of multimodel databases for mixed OLTP and OLAP workloads.



(a) Using an HTAP system to build a unified database containing all data.

(b) Using an existing implementation of a middleware-like polystore system.

Figure 2.4 Possible approaches for implementing the demands of the Gavel scenario replacing the current databases with an available data management solution.

2.3.4 A Polystore System

In comparison to an HTAP system, polystores accept queries through multiple query interfaces. It is therefore not necessary to modify the applications. Furthermore, some polystores also support multiple data models. By connecting to multiple database systems optimized for different data and workloads, polystores are able to query data physically located on different database systems through one interface and using one query language. Figure 2.4(b) depicts a setup where the existing databases are replaced by a polystore system. At a first glance, polystores seem to be a perfect fit for this scenario. However, there are some severe issues making polystores unfeasible for the Gavel scenario.

As outlined in Section 1.3, existing polystore systems lack support for transactions and other "typical" DBMS functionality like the enforcement of constraints [Sto15]. However, the most crucial limitation is that most existing polystore implementations do not support data modification queries [VHS+20; LHC18a].

2.4 A New Kind of DBMS

Since none of the aforementioned approaches is a satisfying solution for the outlined scenario, there is the need for a new kind of data management solution. This solution needs to combine the advantages of HTAP systems, multimodel databases and polystores and at the same time provide the full capabilities of a database management system. Figure 2.5 depicts the envisioned solution; we call this new class of database systems *PolyDBMS*.

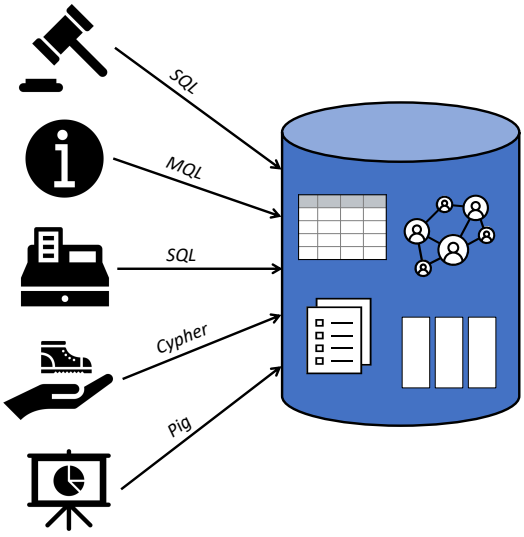


Figure 2.5 PolyDBMS: combining the advantages of multimodel databases, polystores and HTAP systems.

Such a PolyDBMS needs to maintain data represented in different data models, and provide access to this data through one logical schema that enables cross-model queries. Furthermore, it needs to provide the ability to query and join this data using the query languages of the existing applications. There also must be means to present each application with its expected schema and data (e.g., views).

This novel kind of database system needs to be capable of handling mixed workloads consisting of transactional and analytical queries. The performance of the transactional workload needs to be comparable to the performance of the current setup with individual database systems. Furthermore, there needs to be support for data manipulation queries. The system also needs to provide support for transactions and other typical features of a DBMS such as the enforcement of constraints.

Innovation is an evolutionary process, so it's not necessary to be radical all the time.

— Marc Jacobs

3

PolyDBMS

The lack of a suitable solution for the demands outlined in Chapter 1 and Chapter 2 require a multimodel data storage and management solution that bridges the gap between HTAP systems and polystores. With *PolyDBMS*, we propose a new class of database management systems stepping up to fill this gap [VLG+21].

The concept of a PolyDBMS builds on the polystore idea and also shares some major principles, such as support for multiple query languages. However, while polystore systems primarily seek to replace data warehouses and target analytical workloads [Sto15; VHS+20], the focus of a PolyDBMS is more holistic and encompasses the whole data storage and management needs, including heterogeneous data and mixed transactional and analytical workloads. Furthermore, a PolyDBMS combines this with the idea of a multimodel database and enables cross-model queries.

A PolyDBMS is not only a platform for data analytics; it is a full-fledged database management system that hides the complexity and enables all data to be maintained using different data models. Data can be retrieved and modified using different query languages and across different data models.

3.1 Principles & Requirements

In this section, we define the requirements and outline the core principles every PolyDBMS must fulfill. The specifications are intentionally implementation-agnostic and demand only a subset of the features and capabilities provided by our implementation Polypheny-DB (see Chapter 11).

3.1.1 Multiple Data Models

The "variety" of data is one of the most challenging issues in the area of Big Data Management [LHC18b]. Different database data models have been developed to logically represent and structure the data; see Chapter 4. Besides structuring the data, the data model also defines the set of operations that can be performed on the data [GMUW13].

Requirement 3.1 Multiple Data Models

A PolyDBMS must support at least two data models.

A motivation for this requirement is provided by the Gavel scenario introduced in Chapter 2: the fictional auction house uses applications that rely on different data models for storing and processing their data. The auction item catalog, for instance, stores semi-structured data in a database based on the document model while the product recommendation system is based on a graph data model. Migrating everything to one data model is not feasible since the semantics introduced by the data model would be lost. The PolyDBMS therefore needs to be capable of accommodating multiple data models. In the Gavel scenario, for instance, support for three data models is required.

3.1.2 Polymorph and Polyglot

PolyDBMSs inherit the polymorph and polyglot nature of polystore systems. Supporting multiple query languages or query methods and integrating multiple storage and execution engines is a core requirement for every PolyDBMS.

Requirement 3.2 Polymorph and Polyglot

For every natively supported data model, there needs to be support for at least one query language based on this data model. Furthermore, a PolyDBMS needs to support at least one data storage and execution engine for each supported data model.

SQL is often referred as the lingua franca for querying databases [BAH+19; FCP+12]. However, as pointed out in [VJ84], there is no single query language capable of accommodating the full variety of requirements. Furthermore, the top-ten of the *DB-Engines Ranking*[1] from January 2022 contains three database systems without SQL support: Mon-

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

goDB[2], Redis[3], and Elasticsearch[4]. These three systems also experience a continuing growth in popularity. Moreover, there is also not "that" one SQL, rather almost every SQL-enabled database has its own SQL dialect.

Since Requirement 3.1 requires every PolyDBMS to support multiple data models, there also needs to be query languages that allow exploiting the full potential of these data models. Especially in scenarios like the one introduced in Chapter 2, the adjustment of applications can only be avoided if the PolyDBMS supports all query languages and interfaces used by the applications. Hence, supporting several query languages is a key attribute of PolyDBMSs.

A PolyDBMS needs to support at least one storage and execution engine for every supported data model. Mapping data to a different data model for storage and processing can have a significant impact on the query performance. This is demonstrated in [WPW+14], where the performance of MySQL (a relational database management system) and HBase (a key-value store) is compared. For the scenario considered in this paper, the presented benchmarking results show that the key-value store is 5.24 times faster than the relational database system.

3.1.3 Independence of Storage Configuration

The requirement for a PolyDBMS to support multiple storage and execution engines (see Section 3.1.2) introduces a problem already known from polystore and multistore systems: the capabilities and potentially also the result of a query depend on where (i.e., on which storage engine) the data is stored and the query is executed.

Requirement 3.3 Independence of Storage Configuration

The result of a query produced by a PolyDBMS must be independent of how and where the data is physically stored and by which engine it has been processed. Furthermore, the capabilities of the PolyDBMS regarding the available query languages, operations and functions must not depend on the physical storage of the data.

In [Sto15], Michael Stonebraker already identified the first part of this requirement—that the result needs to be independent of where the data is physically stored—as a major issue with existing polystore systems.

[2] https://mongodb.com/
[3] https://redis.com/
[4] https://elastic.co/elasticsearch/

An example for this is the *day of week (dow)* function available in most SQL dialects. This function takes a date or timestamp and returns an integer representing the day of the week. However, some database systems define the first day of the week to be Monday while others define it to be Sunday. In a PolyDBMS, the result needs to be in line with the specification of the query language and dialect in which the query has been expressed. Furthermore, the result must be consistent independently of the involved execution engines. An exception are floating-point operations, which only have to be consistent up to a data type specific precision.

In a PolyDBMS, the set of possible allocations of data to storage engines is called *storage configurations*. Requirement 3.3 requires that a PolyDBMS provides the same capabilities regardless of where the data is physically stored; hence, for all available storage configurations. The available query features and operations are only defined by the query language. A PolyDBMS must not require a specific storage configuration for providing a specific feature of a query language. The only observable difference between different storage configurations of a PolyDBMS may be the time a query takes to execute. The outcome of the query and the resulting state of the data governed by the PolyDBMS must be identical.

Both requirements also apply to data manipulation queries: a query modifying data should always behave the same and result in the same database state, regardless of where and how the data is physically stored. This is especially important if the PolyDBMS supports or requires the replication of data to multiple engines. The PolyDBMS must guarantee that these operations are consistent and reproducible regardless of the selected storage configuration.

A PolyDBMS may reduce the set of available storage configurations to those fulfilling this requirement. This is legitimate as long as it cannot be circumvented and effectively prevents any configuration violating this requirement. However, limiting the set of storage configuration may not conflict with any other requirement.

3.1.4 One Logical Schema

A pillar of the PolyDBMS idea is to foster a tight integration of applications and data by providing a holistic view on all available data across different storage engines and data models.

Requirement 3.4 One Logical Schema

The entities of all data models need to be mapped into a holistic schema that enables cross-model queries. This logical schema needs to be independent of how and where the data is physically stored. Furthermore, the PolyDBMS needs to provide means to define and alter the schema at runtime.

It is important that this holistic schema preserves the semantics of the individual data models. Furthermore, this schema must also enable cross-model queries and may not separate the schema into islands that can only be queried by a subset of the query languages. How this logical schema is organized and represented is up to the PolyDBMS. An approach for building such a holistic schema is introduced in Chapter 8 of this thesis.

The level of detail in which the schema is expressed depends on the data model. Schema-optional or schema-less data models such as the document model might only define the names of the available collections, while a relational schema can be more specific.

The logical schema needs to be adaptable at runtime. Whether this is done using data definition queries, through a user interface, or any other approach is up to the specific implementation of the PolyDBMS—as long as the whole schema can be altered without restarting the PolyDBMS. Furthermore, the schema definition and altering must not introduce an implicit mapping to another data model. This would be the case if the data definition language is based on a data model other than the one to be defined. In such a case, the query language might implicitly prevent the definition of schemas not expressible by the data model the query language is based on. However, the PolyDBMS may provide data model specific approaches for defining and altering the schema.

3.1.5 Multifaceted Schema Model

A key element of a PolyDBMS is the ability to semantically integrate and combine entities in one logical schema (see Section 3.1.4) while at the same time exposing multiple application specific schemas on the query interfaces. The logical schema therefore also serves as an integrated conceptual schema.

Requirement 3.5 Multifaceted Schema Model

A PolyDBMS must provide means to individually adjust the schema that is exposed through a specific query interface.

This feature is especially important if a PolyDBMS is used to replace a zoo of existing database systems without changing the corresponding applications (see the Gavel scenario introduced in Chapter 2). One issue that needs to be handled are colliding names. If, for instance, two applications expect an entity (e.g., a relational table) with the same name (e.g., customer) but with different fields (i.e., columns), directly exposing the logical schema required by Requirement 3.4 causes issues.

If the two customer tables of this example represent the same conceptual entity (the customers of the company) and contain (partially overlapping) information on the same customer, the PolyDBMS needs to provide means for integrating the data in one logical entity but only expose adjusted subsets of this entity to the individual applications. This also needs to be possible across data models.

Similarly, the solution provided by the PolyDBMS also needs to be capable of handling conceptually different entities with the same name and map them to distinct entities in the logical schema. In the Gavel scenario, this could for example be two applications that both maintain their internal information in tables with the name *session*. While both tables have the same name and might even have the same set of fields, they are conceptually different and need to be maintained independently. The PolyDBMS therefore needs to provide means for performing schema integrations.

3.1.6 ACID Compliant Transactions

Operational workloads usually consist of one or more database operation grouped into a *transaction* [GMUW13]. A transaction represents a unit of work that needs to be executed in a coherent and reliable manner and independently of other transactions [EN16]. The acronym *ACID* [HR83] refers to a set of properties a proper transaction needs to fulfill [GMUW13].

Transactions are a pillar of information processing [WV02]. Since PolyDBMSs seek to provide a holistic solution for data management, handling operational workloads is required. Support for transactions fulfilling the ACID properties is therefore mandatory for every PolyDBMS.

Requirement 3.6 ACID Compliant Transactions

Every PolyDBMS must provide support for ACID-compliant transactions. Providing these guarantees only for a subset of the available storage configurations is legitimate. However, providing ACID guarantees must not limit other functionality.

As stated in the requirement, the PolyDBMS does not need to provide support for ACID-compliant transactions for all possible storage configurations (see Section 3.1.3). It is therefore legitimate for a PolyDBMS to trade transaction support against performance. However, transaction support is treated as a property of the storage configuration and therefore falls under the requirements defined in Section 3.1.3. Hence, the PolyDBMS must provide support for the same set of data models, query languages, operations, and query functions, regardless of whether ACID-compliant transactions are supported for a storage configuration or not.

3.1.7 Translytical

The term *translytical* refers to database management systems that, similar to HTAP systems, support heterogeneous workloads including transactional workloads and analytical workloads [YGL+17]. Transactional workload (also referred to as *OLTP* or *operational workload*) usually consists of short running transactions containing simple read and write queries. Analytical workload (also referred to as *OLAP*) involves complex and often long-running queries [GMUW13]. Since the idea of a PolyDBMS is to provide a holistic approach to data management, both transactional and analytical workloads must be supported.

Requirement 3.7 Translytical

A PolyDBMS is optimized for heterogeneous workloads containing both, transactional and analytical queries. Furthermore, a PolyDBMS must provide support for data modification queries.

The performance of a PolyDBMS may depend on the storage configuration. However, the PolyDBMS must be capable of optimizing an entity for multiple types of workload at the same time. This can for example be done by replicating data to multiple storage engines at the same time.

3.2 Architectural Models

There are multiple architectural approaches for implementing a PolyDBMS system. In this section, we will first introduce two approaches that could be seen as continuations of existing kinds of database management systems. We outline the difficulties and disadvantages of these approaches and explain why we do not consider them practical

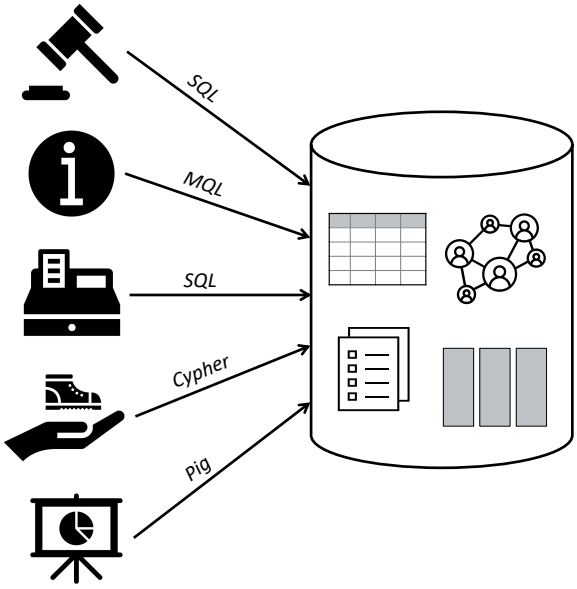


Figure 3.1 A PolyDBMS following the monolithic architecture model. Queries are accepted in different query languages. Integrated within the database are multiple storage and execution engines.

for building a PolyDBMS. Instead, we propose a hybrid of these approaches combining the strengths of multimodel DBMS, polystores and HTAP systems.

3.2.1 Monolithic Architecture

The monolithic architecture model depicted in Figure 3.1 describes the architectural approach of most database management systems: one single software taking care of query parsing, processing, execution and the storage of the data. This architecture is for instance described by Hellerstein et al. in [HSH07].

This architecture is typical for both relational and NoSQL systems. However, there are also multimodel databases and HTAP systems following this architectural approach, for example SAP HANA[5], ArangoDB[6] or OrientDB[7]. The monolithic architecture approach can be seen as the continuation of the "One Size Fits All Idea" that Stonebraker et al. prognosed to come to an end [SÇ05].

Figure 3.1 depicts a PolyDBMS following the monolithic architecture model. This hypothetical PolyDBMS supports multiple data models and query languages. The system also

[5] https://sap.com/products/hana.html
[6] https://arangodb.com/
[7] https://orientdb.org/

takes care of the whole data storage layer, storing data in multiple physical data models. Systems implemented based on this architecture are easy to deploy since there are no dependencies to other systems.

Another advantage of this architectural model is its potential for optimization. Building the whole stack allows reducing overhead and aligning all components. A coherent type system for the entire storage layer, for example, eliminates the need for type conversations. Implementing all execution engines also makes fulfilling Requirement 3.3 (*Independence of Storage Configuration*) much easier. However, the need to implement the entire stack is also the biggest drawback of this approach. Matching the performance of existing (domain specific) database systems with their years or even decades of optimization is very challenging—to say the least.

While the opportunity to align all components and to optimize all aspects of the system that comes with this architectural model might have the potential to build a system superior in terms of performance compared to the approaches introduced in the following sections, we argue that this advantage is of rather theoretical nature. Following this approach means reproducing all the optimizations and tweaks that have been made, sometimes over decades, in systems such as PostgreSQL[8], Oracle[9], MongoDB[10] or Neo4J[11].

3.2.2 Middleware Architecture

The middleware architecture is less complex than the monolithic architecture. The idea is to benefit from the decades of research and development in database systems by using existing, highly optimized, database systems for storing the data and executing the queries. The task of the middleware is to select the most appropriate storage engine for executing a query, to translate the query into the query language of the underlying database system, and to forward the query for execution to the selected database. Furthermore, the middleware is responsible for keeping the data consistent across the database systems. Figure 3.2 depicts such an architecture.

An example of a system following this architecture model is Icarus [VSS17]. In this system, all data is replicated to all underlying databases. For every incoming query, the Icarus system selects the database system with the best characteristics for executing this

[8] https://postgresql.org/
[9] https://oracle.com/database/
[10] https://mongodb.com/
[11] https://neo4j.com/

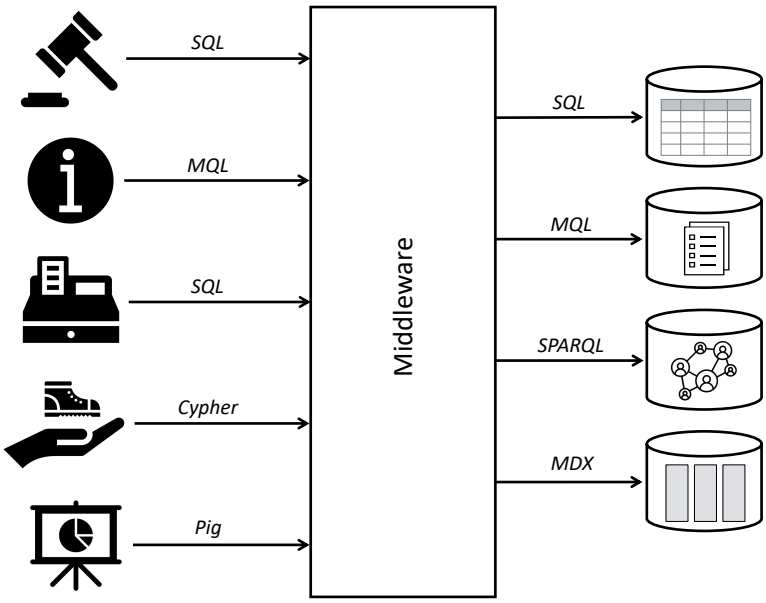


Figure 3.2 A PolyDBMS following the middleware architecture model. Queries are accepted in different query languages and are executed on one of the underlying database systems. For this, the query is translated by the middleware to the query language required by the selected database system.

query, translates the query into the query language of the underlying database, executes the query, and returns the result to the client.

An advantage of this approach is its horizontal scalability: the underlying database systems can be deployed on different physical machines, with the middleware acting as an intelligent load balancer over heterogeneous systems.

However, while this approach is simple, it also comes with some major issues, one being the bad performance for modification operations [NJ00; CGP80]. Since the data is fully replicated across all databases, every data modification needs to be executed on all systems. Another issue is the difficulty to keep the data stored in the heterogeneous database consistent for arbitrary update operations: already small differences in the interpretation of conditions in the query or the implementation of functions between the involved databases can lead to inconsistent data. An example for this is the already mentioned `dow()` function available in many (relational) database systems. This function returns the day of the week as an integer. However, some database systems treat Sunday as the first day of the week, while others treat Monday as the first day of the week. A query that, for instance, updates all entries with a specific date would therefore result in inconsistent data. Furthermore, the Requirement 3.3 (*Independence of Storage Configuration*) cannot be fulfilled using this architecture either.

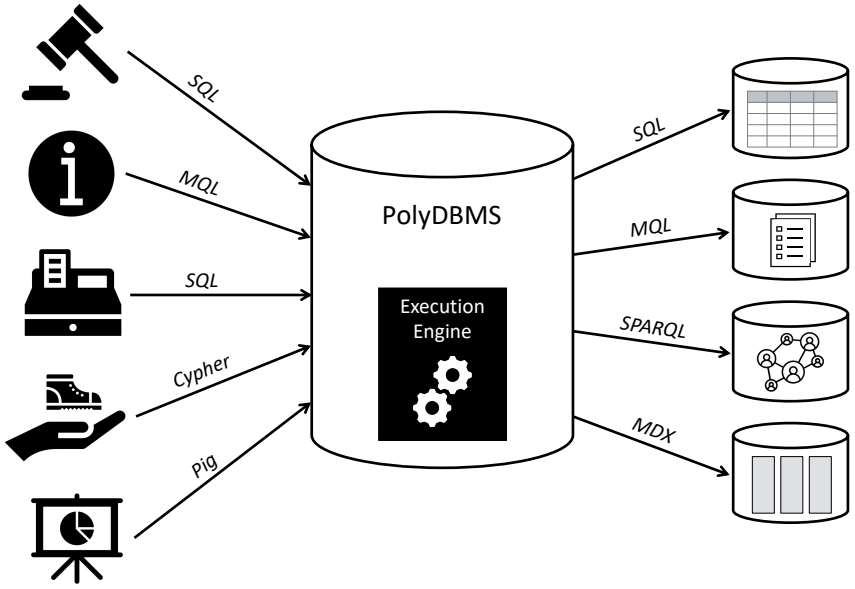


Figure 3.3 A hybrid of the monolith architecture and the middleware architecture for building a PolyDBMS.

3.2.3 Hybrid Architecture

The hybrid architecture model depicted in Figure 3.3 combines the advantages of the monolithic architecture model and the middleware architecture model. Like the middleware architecture, it makes use of the enormous optimizations that went into existing database systems by using them for storing data and executing queries. However, these underlying databases are complemented by an execution engine within the PolyDBMS. This execution engine compensates for missing features on the underlying databases, which allows fulfilling Requirement 3.3 (*Independence of Storage Configuration*). The execution engine also allows combining (e.g., joining) results from different underlying database systems.

The PolyDBMS seamlessly combines replication and partitioning of the data across the heterogeneous underlying database systems. This allows to optimize the storage configuration according to the workload. Furthermore, the integrated execution engine allows the evaluation of complex conditions in data modification queries and can reduce them to basic operations that can be executed consistently on all underlying databases. The hybrid architecture therefore applies a Local-as-View (LAV) based schema mapping over the underlying data stores.

Similarly to the middleware approach, this architecture is inherently distributable. This allows for an easy horizontal scaling by deploying the underlying database systems on different physical machines.

This architecture is more complex than the middleware architecture, but still less complex than the monolithic approach. Since the integrated execution engine is only a fallback and most queries are processed on the underlying database systems, the performance of the integrated execution engine is of secondary importance.

3.3 Research Objectives

A PolyDBMS is not characterized by its architectural model but by its ability to provide certain capabilities. Hence, every system fulfilling the requirements outlined in Section 3.1 is a PolyDBMS. However, to our best knowledge, there is no database system providing support for all of the requirements outlined in Section 3.1. While systems like the three aforementioned systems SAP HANA, ArangoDB and OrientDB fulfill some of the requirements, none of these systems fulfills all of them (see Chapter 15 for a detailed breakdown). Furthermore, all of these systems follow the monolithic architecture approach outlined in Section 3.2.1 and therefore suffer from the drawbacks of this approach outlined before.

In this thesis, we present the results of our research that enables to build such a PolyDBMS. Due to the aforementioned drawbacks of systems built according to a monolithic or middleware architecture, we consider these architectures to be inadequate for building a PolyDBMS that is able to compete with existing systems in single-model workloads and scenarios like the one presented in Chapter 2. The concepts presented in this thesis are therefore based on the hybrid architecture model introduced in Section 3.2.3. In what follows, we refer to a PolyDBMS as a system built according to this architecture.

Since such a system is an evolution and combination of existing database technologies, there are already solid foundations and proven solutions on which we can build. However, we identified three areas where additional research is required to address the emerging challenges of building a PolyDBMS:

- Supporting multiple data models in one system, as it is required by Requirement 3.1, is currently a subject of ongoing research and development. However, combining these beneath one logical schema (Section 3.1.4) and exposing different schemas on the interfaces (Section 3.1.5) requires further research. The required concep-

tual model also needs to enable cross-model queries. Furthermore, it needs a conceptual solution for combining data partitioning and data replication across heterogeneous data models while at the same time meeting the requirements defined in Section 3.1.3. *(→ Chapter 8)*

- Due to the requirement introduced in Section 3.1.2, a PolyDBMS needs to support multiple query languages based on different data models. This requires a model for representing and handling these heterogeneous queries. This model must preserve the data model specific semantics of operations and at the same time also allow queries across data models. *(→ Chapter 9)*
- A database system that features multiple execution and storage engines, and replicates and partitions data across these engines, requires a sophisticated model for planning the optimal execution of queries. This includes the decomposition of queries and the selection of the underlying database systems on which the subqueries should be executed. The requirement of Section 3.1.7 to support transactional workload consisting of short-running queries makes this even more challenging since the routing must not significantly increase the execution time of these queries. *(→ Chapter 10)*

Besides these conceptual models, there are also some implementational challenges that arise from adapting and combining existing solutions in such a system. With our implementation Polypheny-DB, we demonstrate that the described concepts can be implemented in a holistic system. *(→ Chapter 11)*

Part II

Foundations

Preface to Part II:
Outline the Field

In this part, we introduce the concepts that lay the foundations for the work presented in this thesis. Since there is unfortunately not a single commonly agreed up on definition for several of the important terms and concepts, another purpose of this part of the thesis is to introduce a terminology. This allows for a more concise and stringent discussion of the conceptual models presented in the third part of this thesis.

The purpose of this part is not to give a holistic introduction to the whole area of data management. Since this thesis focuses on PolyDBMS built according to the hybrid architecture approach introduced in Section 3.2.3, we only introduce foundations required in the context of this architecture model. Furthermore, we also limit ourselves to the topics that are relevant for the concepts presented in Part III.

As introduced in Chapter 3, a PolyDBMS built according to the hybrid architecture model is a multimodel database management system that combines heterogeneous underlying database systems. This combination of approaches does also reflect in the structure of this part of the thesis.

The foundations for the *multimodel* nature of a PolyDBMS are outlined in **Chapter 4**. This chapter gives an overview of three database data models. For each, a formal model will be introduced. These models include the representation of data, basic operations on the data, and the available building blocks for defining schemas and imposing constraints.

A PolyDBMS is a full-fledged *database management system*. **Chapter 5** therefore introduces the necessary foundations of database management systems. This includes the architecture and building blocks of a database management system, transactions and concurrency control.

Storing data across different underlying database systems requires techniques known from *distributed systems*. In **Chapter 6** these foundations will be introduced. This includes an overview of data replication and data partitioning, temperature-aware data management, distributed transactions, and data freshness.

See first that the design is wise and just; that ascertained, pursue it resolutely.

— William Shakespeare

4

On Data Models

A fundamental characteristic of a database system is its ability to provide a level of abstraction from the data. The *schema* of the database is abstracted from the way in which the data is physically stored. The *data model* is a collection of concepts that can be used to define this schema [EN16]. In the literature, the term is sometimes also used to refer to the particular schema of the database. In this thesis, however, a clear distinction is made between these two terms:

The **database schema** is the blueprint of the database. It defines the layout in which the data is logically organized. Furthermore, the schema might also include a set of constraints imposed on the data [EN16]. However, the level of detail with which a schema is defined depends heavily on the data model.

The **data model** defines the set of building blocks available for defining the schema. Furthermore, it might also define a limited set of operations that can be used to query and manipulate the data [GMUW13], and provides the building blocks for defining constraints on the data.

Over the past decades, several database data models have been proposed. One approach to categorize these data models is based on the conceptual level at which the schema is modeled [EN16]. As depicted in Figure 4.1, usually a distinction is made between three levels of abstraction. In this thesis we refer to them as *conceptual*, *logical* and *physical*.

Conceptual data models are used to model the semantics of a real-world domain. They provide concepts that are close to the way in which the data is perceived by the user [EN16]. The *Entity-Relationship model* [Che76] proposed in 1976 by Peter Chen is a popular approach for modeling conceptual schemas. It uses concepts such as entities, attributes, and relationships. An *entity* represents a real-world object or concept, such as a customer or product. *Attributes* are used to further describe an entity, for example,

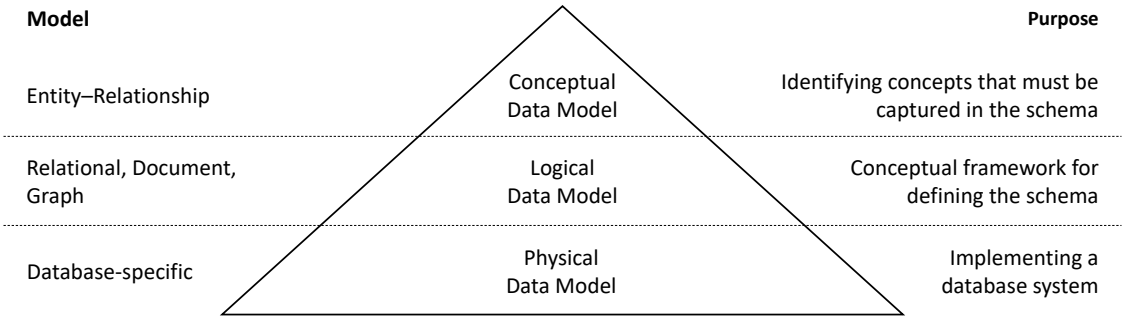


Figure 4.1 Different levels of data models.

the name of the customer or the price of a product. With *relationships*, it is possible to model associations between entities, for example, that a customer can buy a product. Conceptual data models are typically not very detailed. They allow to model real-world things and the relationships between them, rather than the organization of the data about those things [Wes11]. They are used as a tool for understanding and organizing information.

Logical data models formalize how data can be organized, queried, and modified. They are more concrete and data-centric than conceptual models, but still independent of a specific database product or implementation. The relational data model [Cod70] is the most popular citizen of this class. However, other data models gain increasing importance. Not only do they represent data differently, but they also differ in the set of provided building blocks for defining the schema and provide a different set of operators that can be applied to the data organized by that schema.

Physical data models are specific to the implementation of the database management system, and dependent on specific data types and indexing mechanisms of this particular DBMS [Nye17]. They describe the physical data structures used for storing and processing the data on a computer system [Wes11]. Furthermore, they define low-level aspects such as read and write operations on these data structures.

In this work, we are mainly interested in the logical data models. The term *data model* therefore always refers to the class of logical data models. In the following sections, the relational, document, and labeled property graph data models are introduced. For every model, we follow the structure introduced by [GMUW13] and consider the following three aspects of a data model:

- *Structure of the data*: The structures or building blocks provided by the data model to define the schema. Furthermore, the way in which data accommodated by this data model can be represented.

- *Operations on the data*: The operators defined by the data model to construct queries that can be applied to the data.
- *Constraints on the data*: The kinds of constraints that can be defined within this data model and imposed on the data governed by it.

4.1 Relational Model

The relational data model has been introduced in June 1970 by Edgar F. Codd in a research article with the title "A Relational Model of Data for Large Shared Banks" [Cod70]. It presented a major breakthrough in data management [CMR11]. The basic principle of the relational model is the representation of data by means of mathematical relations. A database organized according to the relational data model is called a *relational database*.

4.1.1 Structure of the Data

In the relational model, data is represented as N-ary relations. An N-ary relation $\mathcal{R}$ is the subset of the Cartesian product of N *data domains*; with N being a positive natural number. A data domain $\mathcal{D}$ is a set of atomic values. It represents the possible values for an *attribute*. An attribute $\mathcal{A}$ is the combination of a data domain and a name. The name must be distinct within a relation.

The number of attributes of a relation is called *degree*. A *tuple* t is a list of values that is constructed such that the first element belongs to the first attribute, the second to the second attribute, and so on. The set of possible tuples of a relation $\mathcal{R}$ is therefore given by the Cartesian product of the data domains of the attributes that belong to this relation. Let $(\mathcal{A}_1, ..., \mathcal{A}_N)$ be attributes of a relation $\mathcal{R}$ and $\mathtt{DOM}(\cdot)$ be a function returning the domain of an attribute. A tuple t of a relation $\mathcal{R}$ is defined as:

$$t \in \mathtt{DOM}(\mathcal{A}_1) \times \mathtt{DOM}(\mathcal{A}_2) \times ... \times \mathtt{DOM}(\mathcal{A}_N) \tag{4.1}$$

Since data domains are a set of atomic values, structured or multi-valued entries are not allowed in the classical relational model.[1] The number of tuples of a relation is called *cardinality*. In the relational model, the tuples of a relation are a set; thus, no tuple can occur more than once.[2] Furthermore, the tuples of a relation are inherently unordered.

[1] This property is nowadays usually relaxed in relational database systems.
[2] In relational databases systems, this is extended to allow duplicate values.

Every relation $\mathcal{R}$ has a fixed schema. This schema is defined by an ordered list of attributes, where $(\mathcal{A}_1, ..., \mathcal{A}_N)$ are attributes belonging to the N-ary relation $\mathcal{R}$:

$$\text{SCH}(\mathcal{R}) := (\mathcal{A}_1, ..., \mathcal{A}_N) \tag{4.2}$$

Formally, a relation $\mathcal{R}$ consists of the schema $\text{SCH}(\mathcal{R})$ and an extent (i.e., the values). The extent is a finite subset of the possible tuples:

$$\text{VAL}(\mathcal{R}) \subseteq \text{DOM}(\mathcal{A}_1) \times \text{DOM}(\mathcal{A}_2) \times ... \times \text{DOM}(\mathcal{A}_N) \tag{4.3}$$

A *relational schema* can consist of multiple relations $\{\mathcal{R}_1, ..., \mathcal{R}_M\}$ with $M \in \mathbb{N}$. Every relation has a distinct name within the relational schema.

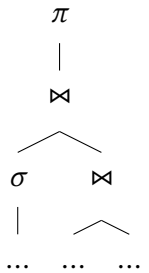
In the context of relational database systems, a relation is usually called *table*. The attributes are typically called *columns* and the tuples are referred to as *rows* or *records*. A data domain usually corresponds to the *data types* (e.g., integer, varchar, ...) supported by the database system.

4.1.2 Operations on the Data

The relational model introduced in [Cod70; Cod90], also introduces the *relational algebra*. It defines multiple operators that can be applied to the data. Every operator takes one or multiple relations as input and returns a new relation as output. For what follows, we can restrict to *unary* operators taking one relation as input and *binary* operators taking two relations as input.

$$\begin{aligned} \text{UNARY} &: \mathcal{R}_{\text{INPUT}} \rightarrow \mathcal{R}_{\text{OUTPUT}} \\ \text{BINARY} &: \mathcal{R}_{\text{LEFT}}, \mathcal{R}_{\text{RIGHT}} \rightarrow \mathcal{R}_{\text{OUTPUT}} \end{aligned} \tag{4.4}$$

Since operators output exactly one relation, it is possible to arrange them as trees. The resulting operator tree allows constructing complex *queries* using these basic operators and involving multiple relations.



In this section, we briefly introduce the most important operators defined by the relational algebra. For every operation we specify the extent and the schema of the resulting output relation.

4.1.2.1 Projection

The projection is a unary operator that allows to restrict all tuples of the input relation to a specified list of attributes. All other attributes are discarded. The list of attributes of the input relation to be preserved in the output relation is called *projection list*, denoted with the symbol β. The schema of the output relation is reduced accordingly.

Let $\mathcal{R}$ be a relation over the attributes $(\mathcal{A}_1, ..., \mathcal{A}_N)$ and β be a list of attributes with $\beta \subseteq (\mathcal{A}_1, ..., \mathcal{A}_N)$. Further, let $\beta(\cdot)$ reduce a tuple to the list of attribute specified in the projection list β. The projection $\pi_\beta(\cdot)$ is defined as:

$$\begin{aligned} \pi_\beta(\mathcal{R}) &:= \left\{\beta(t) \mid t \in \mathcal{R}\right\} \\ \mathrm{SCH}\big(\pi_\beta(\mathcal{R})\big) &:= \beta \end{aligned} \tag{4.5}$$

4.1.2.2 Selection

The selection is a unary operator that filters the tuples of a relation according to a specified filter predicate. The selection only affects the extent of a relation. The schema of the output relation is equal to the schema of the input relation. The selection operator is denoted by the symbol σ.

The filter predicate φ is a propositional formula consisting of attributes of the input relation $\mathcal{R}$, constant values, comparison operators ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$), and the logical operators ($\wedge$ (and), $\vee$ (or), $\neg$ (negation)). The selection operator $\sigma_\varphi(\cdot)$ selects all those tuples t of the input relation $\mathcal{R}$ for which φ is fulfilled. With $\varphi(\cdot)$ being the Boolean function of the predicate φ, the selection operator is defined as:

$$\begin{aligned} \sigma_\varphi(\mathcal{R}) &:= \{t \mid t \in \mathcal{R} \wedge \varphi(t) = \mathsf{true}\} \\ \mathrm{SCH}\big(\sigma_\varphi(\mathcal{R})\big) &:= \mathrm{SCH}(\mathcal{R}) \end{aligned} \tag{4.6}$$

4.1.2.3 Cartesian Product

The Cartesian product is a binary operator that combines every tuple t^ℓ from the left input relation $\mathcal{R}_\ell$ with every tuple t^r from the right input relation $\mathcal{R}_r$. The schema of

the output relation consists of all attributes of both input relations. The extent of the output relations contains all possible permutations of the tuples. The Cartesian product operator $\times$ is defined as:

$$\begin{aligned} \mathcal{R}_\ell \times \mathcal{R}_r &:= \left\{ t^\ell \cup t^r \mid t^\ell \in \mathcal{R}_\ell \wedge t^r \in \mathcal{R}_r \right\} \\ \mathrm{SCH}(\mathcal{R}_\ell \times \mathcal{R}_r) &:= \mathrm{SCH}(\mathcal{R}_\ell) \cup \mathrm{SCH}(\mathcal{R}_r) \end{aligned} \tag{4.7}$$

4.1.2.4 Join

The join is an extension of the Cartesian product that only contains those permutations of the tuples from the input relations $\mathcal{R}_\ell$ and $\mathcal{R}_r$ where the value for a specific attribute $\mathcal{A}_\ell$ in t^ℓ and the value for a specific attribute $\mathcal{A}_r$ in t^r fits the criterion $\otimes \in \{<, \leq, =, \neq, \geq, >\}$. More formally, with $\mathcal{A}_\ell \in \mathrm{SCH}(\mathcal{R}_\ell)$, $\mathcal{A}_r \in \mathrm{SCH}(\mathcal{R}_r)$ and $\mathcal{A}(\cdot)$ returning the value for this attribute, the join is defined as:

$$\begin{aligned} \mathcal{R}_\ell \bowtie \mathcal{R}_r &:= \left\{ t^\ell \cup t^r \mid t^\ell \in \mathcal{R}_\ell \wedge t^r \in \mathcal{R}_r \wedge \left(\mathcal{A}_\ell(t^\ell) \otimes \mathcal{A}_r(t^r) \right) \right\} \\ \mathrm{SCH}(\mathcal{R}_\ell \bowtie \mathcal{R}_r) &:= \mathrm{SCH}(\mathcal{R}_\ell) \cup \mathrm{SCH}(\mathcal{R}_r) \backslash \mathcal{A}_r \end{aligned} \tag{4.8}$$

A join can also be expressed using the Cartesian product and the selection operator

$$\mathcal{R}_\ell \bowtie \mathcal{R}_r := \sigma_\varphi(\mathcal{R}_\ell \times \mathcal{R}_r) \tag{4.9}$$

with $\varphi := \mathcal{A}_\ell(t^\ell) \otimes \mathcal{A}_r(t^r)$. If $\otimes$ is the equals operator (=), this is also called an *equijoin*.

4.1.2.5 Set Operations

The three basic binary operations *union*, *intersection* and *difference* known from set theory can also be applied on relations. However, it is required that they have the same schema. For two input relations $\mathcal{R}_\ell$ and $\mathcal{R}_r$ for which it holds $\mathrm{SCH}(\mathcal{R}_\ell) = \mathrm{SCH}(\mathcal{R}_r)$, the three set operations are defined as:

$$\begin{aligned} \mathcal{R}_\ell \cup \mathcal{R}_r &:= \{ t \mid t \in \mathcal{R}_\ell \vee t \in \mathcal{R}_r \} \\ \mathcal{R}_\ell \cap \mathcal{R}_r &:= \{ t \mid t \in \mathcal{R}_\ell \wedge t \in \mathcal{R}_r \} \\ \mathcal{R}_\ell \backslash \mathcal{R}_r &:= \{ t \mid t \in \mathcal{R}_\ell \wedge t \notin \mathcal{R}_r \} \\ \mathrm{SCH}(\mathcal{R}_\ell \otimes \mathcal{R}_r) &:= \mathrm{SCH}(\mathcal{R}_\ell) = \mathrm{SCH}(\mathcal{R}_r) \quad \text{with} \quad \otimes := \{\cup, \cap, \backslash\} \end{aligned} \tag{4.10}$$

4.1.3 Constraints on the Data

The relational model distinguishes between different types of constraints. A very powerful concept are *keys*. In literature, the term key is typically used to identify a set of attributes that uniquely identify every tuple of the relation. However, in this thesis, we define a key $\mathcal{K}$ as an arbitrary set of one or multiple attributes $(\mathcal{A}_1, ..., \mathcal{A}_k) \subseteq \text{SCH}(\mathcal{R})$. An attribute can be part of multiple keys. There are two special types of keys: *primary keys* and *foreign keys*.

A **primary key**, denoted as $\mathcal{K}_{\text{PK}}$, of the relation $\mathcal{R}$ consists of a subset of the attributes $\mathcal{K}_{\text{PK}} \subseteq \text{SCH}(\mathcal{R})$ that, in conjunction, allow to uniquely identify every tuple of the relation.

A **foreign key**, denoted as $\mathcal{K}_{\text{FK}}$, of the relation $\mathcal{R}$ consists of a subset of the attributes $\mathcal{K}_{\text{FK}} \subseteq \text{SCH}(\mathcal{R})$ that uniquely reference the primary key of the same or another relation.

The relational model furthermore allows constraining an attribute to a specific data type or limiting the set of valid values to a subset of the possible values defined by the data domain.

4.2 Document Model

The document model is designed for storing *semi-structured* data. This is data where there is either no or only a very loose schema on the data. Furthermore, semi-structured data is usually organized in some tree-like structure [Bun97]. This requires a data model that supports such nested data structures. Databases built around the document data model are called *document-oriented databases* or *document stores*.

Despite the popularity that document-oriented databases have gained in recent years, there are only a few attempts at formally defining and understanding their conceptual properties [BCC+16]. In this section, we summarize a formal model introduced in [BCC+16; BCC+18] that is based on the document store MongoDB[3]. According to the DB-Engines ranking[4] from February 2022, MongoDB is the most popular database system based on the document data model.

[3] https://mongodb.com
[4] https://db-engines.com

4.2.1 Structure of the Data

In the document model, data is organized as a *collection* of heterogeneous *documents*. Each of these documents is identified by a unique identifier and contains an arbitrary set of key-value pairs. In contrast to the relational model where the values are atomic (see Section 4.1.1), in the document model, arbitrarily deeply nested structures are possible. These nested structures can consist of key-value pairs (i.e., an associative array or dictionary) and (indexed) arrays:

$$\begin{aligned} \text{Array} &: I \to V \;\text{ with }\; I \subseteq \mathbb{N}_{\geq 0} \\ \text{Dictionary} &: K \to V \end{aligned} \tag{4.11}$$

The sets V and K consist of arbitrary atomic values. The set V further includes the special elements null, true, and false. We can now formalize a document as a finite unranked node and edge labeled out-tree (a rooted directed graph where the path leading from the root to any other node is unique). For labeling the edges, we use the two sets K and I. For labeling the nodes, we use the set V. A labeled tree z is defined as a tuple

$$z := (N, E, \prec, L_n, L_e) \tag{4.12}$$

with N being a set of nodes, E a set of edges, and $\prec$ a binary relation defining an order for certain pairs of nodes in N. This partial order imposes a total order on the children of the nodes in N. L_n and L_e are labeling functions:

$$\begin{aligned} L_n &: N \to V \cup \{\text{'Array'}, \text{'Dictionary'}\} \\ L_e &: E \to K \cup I \end{aligned} \tag{4.13}$$

The labeling function L_n is used for labeling nodes. A node labeled with an element from V is a leaf. The function L_e is used for labeling edges. All outgoing edges of a node labeled with 'Array' must be labeled using consecutive integers (from the set I) starting from 0, according to the order imposed by $\prec$. Outgoing edges of nodes labeled with 'Dictionary' are labeled with keys (from the set K). Given a tree z and a node x, the function $\texttt{type}(x, z)$ can be defined as:

$$\texttt{type}(x, z) := \begin{cases} \texttt{leaf} & \text{if } L_n(x) \in V \\ \texttt{array} & \text{if } L_n(x) = \text{'Array'} \\ \texttt{dict} & \text{if } L_n(x) = \text{'Dictionary'} \end{cases} \tag{4.14}$$

The function $\texttt{root}(z)$ returns the node acting as the root of the out-tree z, i.e., the node without any incoming edges. If such a *root node* has an outgoing edge with the label *id*, it is called a *document*.

To get the value of a node x in z, we define the piecewise function $\texttt{val}(x, z)$ as

$$\texttt{val}(x,z) := \begin{cases} L_n(x) & \text{if } \texttt{type}(x,z) = \texttt{leaf} \\ \big(\texttt{val}(x_1,z), ..., \texttt{val}(x_M,z)\big) & \text{if } \texttt{type}(x,z) = \texttt{array} \\ \Big\{\big(L_e(x,x_1), \texttt{val}(x_1,z)\big), ..., \big(L_e(x,x_M), \texttt{val}(x_M,z)\big)\Big\} & \text{if } \texttt{type}(x,z) = \texttt{dict} \end{cases} \tag{4.15}$$

with $x_1, ..., x_M$ being the children of the node x, so that they are satisfying the order $x_1 \prec ... \prec x_M$. $L_e(x, x')$ is a shorthand for the edge labeling function L_e with $x, x' \in N$ for which there exists an edge in E. Furthermore, we define $\texttt{val}(z)$ as $\texttt{val}(\texttt{root}(z), z)$.

4.2.2 Operations on the Data

Queries on the document data model are modeled as sequences of operators. Every operator transforms a set of trees into another set of trees. These sets of trees are also called *forests*. A forest is denoted with $\mathcal{F}$.

$$\text{OP} : \mathcal{F}_{\text{INPUT}} \rightarrow \mathcal{F}_{\text{OUTPUT}} \tag{4.16}$$

To access values in a tree, concatenations of labels of adjacent edges starting from the root of the tree, called *paths*, are used. A path is a finite sequence of labels. A path p' is a prefix of a path p, if and only if there is a non-empty concatenations of edge labels q for which $p = p'||q$ holds (with $||$ being a string concatenation).

Based on [BCC+18], we introduce the following four operators operating on the data structured according to the model outlined in Section 4.2.1: *match*, *unwind*, *project*, and *group*.

4.2.2.1 Match

The match operator μ_φ selects trees that satisfy the filter φ. This filter is constructed using a Boolean combination ($\wedge$ (and), $\vee$ (or), $\neg$ (negation)) of atomic conditions. These conditions either check the existence of a path p or the value v at a certain path $p = v$ for $v \in V$. Given a forest $\mathcal{F}$, and with $\varphi(\cdot)$ being the Boolean function of the filter predicate φ, the match operator is defined as:

$$\mu_\varphi(\mathcal{F}) := \{z \mid z \in \mathcal{F} \wedge \varphi(z) = \text{true}\} \tag{4.17}$$

4.2.2.2 Unwind

The unwind operator ω_p deconstructs an array at the path p and creates separate output trees for each item in the array. It only transforms the node specified by the path p. The other nodes of every input tree are duplicated, once per array element.

$$\begin{aligned} \omega_p(z) &:= \bigcup_{i<|\texttt{subtree}(z,p)|, i\in I} \texttt{replace}\big(z, \texttt{subtree}(z,p), \texttt{subtree}(z, p||i)\big) \\ \omega_p(\mathcal{F}) &:= \{\omega_p(z) \mid z \in \mathcal{F} \wedge \ \texttt{type}(p,z) = \texttt{array}\} \end{aligned} \tag{4.18}$$

With $\texttt{subtree}(z,p)$ being a function that constructs a tree with all elements below the node specified by the path p and with this node at the path p as root element. The $\texttt{replace}(z_a, z_b, z_c)$ function creates a copy of z_a where the subtree z_b is replaced by the subtree z_c.

4.2.2.3 Project

The project operator ρ_β transforms trees by adding, renaming or removing the nodes and edges on a path p. B is a sequence containing an arbitrary number of elements in the form

- p = true (keeping the nodes and edges on the path p),
- $p = v$ (adding or overriding a leaf at the path p with a constant value $v \in V$),
- $p = \{v_0, .., v_M\}$ (adding or overriding a leaf at the path p with an array definition),
- $p = p'$ (renaming the edges on a path p to p'), and
- $p = \varphi$ (keeping the path p if the Boolean expression φ constructed as introduced for the match operator in Section 4.2.2.1 is fulfilled).

There must be no pair (p, p') where p' is a prefix of p. With the merge operator $\oplus$ that merges trees with identical paths and $\beta(z)$ being the application of $\beta \in B$ on z, the project operator can be defined as:

$$\rho_\beta(\mathcal{F}) := \left\{ \bigoplus_{\beta\in B} \beta(z) \;\middle|\; z \in \mathcal{F} \right\} \tag{4.19}$$

4.2.2.4 Group

The group operator $\gamma_{G:A}$ groups trees based on a sequence of grouping conditions and aggregates values based on a sequence of aggregate specifications. Both consist of tuples

(p_1, p_2) where p_1 is a path in the set of input trees $\mathcal{F}_{\text{INPUT}}$ and p_2 a path in the set of output trees $\mathcal{F}_{\text{OUTPUT}}$.

4.2.3 Constraints on the Data

In the document model, every document has an *id* attribute that allows uniquely identifying every document of a collection. Other than that, the document model is extremely flexible and enforces no constraints on the data.

4.3 Labeled Property Graph Model

Graph data models are used for storing and processing network-like datasets [MSV17]. The ability to represent highly interconnected data makes them suitable for various big data applications including social networks, bioinformatics and astronomy [SS19]. Database systems built around a graph data model are called *graph databases*. There are two important graph database models, the *Labeled Property Graph* (LPG) [RN10] model and the *Resource Description Framework* (RDF) [BM04]. For this thesis, we focus on the LPG model.

4.3.1 Structure of the Data

A graph is a structure known from graph theory. It is composed of a set of *nodes* (also called *vertices* or *points*) interconnected by a set of *edges* (also called *lines* or *links*). In the graph data model, the nodes represent the entities (such as products, companies, or people) and the edges represent the relations between these entities (such as knows, buys, likes). Furthermore, there is an arbitrary number of key-value pairs, called *properties*, associated with a node or an edge. The ability to assign properties to edges distinguishes the LPG model from other graph data models and the relational data model.

In graph theory, a directed graph G is usually represented by an ordered pair $G := (N, E)$ with N being a set of nodes and E being a set of edges. An edge is represented by a pair of nodes.

$$E \subseteq \left\{(n_1, n_2) \mid n_1, n_2 \in N\right\} \tag{4.20}$$

For the LPG model, this definition needs to be extended to allow multiple edges joining the same pair of nodes. In graph theory, such a graph is called a *multigraph*. The definition of G is extended to the tuple $G := (N, E, \texttt{src}, \texttt{tgt})$ with E being a multiset of unordered pairs of nodes and introducing the functions $\texttt{src}(\cdot)$ and $\texttt{tgt}(\cdot)$:

$$\begin{aligned} \texttt{src} &: E \to N \\ \texttt{tgt} &: E \to N \end{aligned} \tag{4.21}$$

The source function $\texttt{src}(\cdot)$ assigns every edge to its source node and the target function $\texttt{tgt}(\cdot)$ assigns every edge to its target node. In graph theory, this type of graph is called a *directed multigraph*. However, for the LPG model, further extensions are required to model labels and properties on both the nodes and edges. For the labeling, we introduce a set of node labels L_N and a set of edge labels L_E. Further we introduce the labeling functions ℓ_N and ℓ_E:

$$\begin{aligned} \ell_N &: N \to \mathfrak{P}(L_N) \\ \ell_E &: E \to L_E \cup \{\mathsf{null}\} \end{aligned} \tag{4.22}$$

With $\mathfrak{P}$ being the power set function $\mathfrak{P}(X) := \{U \mid U \subseteq X\}$ for a set X.

For modeling the properties, we define P_K as the set of property keys and P_V as the set of property values. Furthermore, we introduce a function $\texttt{val}(\cdot)$ which maps a node or edge and a property key to the corresponding property value.

$$\texttt{val} : (N \cup E) \times P_K \to P_V \tag{4.23}$$

With this, a graph in the LPG model can be described by the tuple:

$$G := (N, E, L_N, L_E, P_K, \texttt{src}, \texttt{tgt}, \ell_N, \ell_E, \texttt{val}) \tag{4.24}$$

The LPG model is often described as *schema-optional* [SS19]. It is possible to define a *graph schema* which constrains the graph and enforces a certain structure. However, in contrast to the relational data model, a graph schema is not mandatory to represent data. We therefore introduce the concept of graph schemas as possible constraints on the data in Section 4.3.3.

4.3.2 Operations on the Data

Since there is no common agreement on the definition of what exactly the operators of the LPG model are, we are oriented towards the set of operators introduced by the

openCypher[5] query language. This is not only a widely adopted query language for LPGs, it is also the basis for the upcoming ISO/IEC GQL standard [Gre19]. The formalization of the operators introduced in this section are based on two scientific publications: the article on the Cypher query language [FGG+18] and an article on formalizing openCypher queries [MSV17].

A query in the LPG model takes a graph as input and returns a *graph relation* as output. A graph relation R is a relation over a set of attributes A. The schema of the graph relation R is a list of attributes:

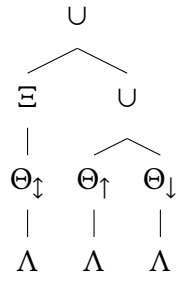
$$\mathrm{sch}(R) := (A_1, ..., A_N) \tag{4.25}$$

The values for an attribute of a graph relation can be complete nodes or edges including their properties, but also single property values. More formally, the $\mathrm{dom}(\cdot)$ function returning the data domain of an attribute is defined as:

$$\mathrm{dom}(A) \subseteq N \cup E \cup P_V \tag{4.26}$$

Similar to the relational model, we define a query as a tree of operators. For this definition of the graph model, we limit ourselves to the following operators: the node operator (Λ), the match operator (Θ), the unwind operator (Ω), the project operator (Π), the filter operator (Ξ), and the union operator ($\cup$). Except for the binary union operators, all operators take up to one graph relation as input.

A query consists of at least one node operator and an arbitrary number of other operators, including zero. The node operator Λ can only appear at the leaves of the query tree and there also needs to be exactly one such operator at every leaf of the tree.



Except for the node operator, all operators take either one or two graph relations as input. The node operator takes a graph as input. The match operator takes both a graph and a graph relation as input. All operators return one graph relation as output. The following paragraphs introduce the elementary operators of this data model.

[5] https://opencypher.org/

4.3.2.1 Node

The node operator $\Lambda_{\alpha,\gamma}$ creates a graph relation with one attribute. It has two parameters: an attribute name α and a list of node labels γ. The node operator produces a graph relation having one attribute with the name α. The relation consists of tuples for every node in the input graph that has all labels specified in γ. For a graph G and with $t[\alpha]$ returning the value of the tuple for the attribute α, the node operator can be defined as:

$$\begin{aligned} \Lambda &: G \rightarrow R \\ \Lambda_{\alpha,\gamma}(G) &:= \{t \mid t[\alpha] \in G \wedge \ell_N(t[\alpha]) = \gamma\} \\ \mathrm{sch}\big(\Lambda_{\alpha,\gamma}(G)\big) &:= (\alpha) \end{aligned} \tag{4.27}$$

4.3.2.2 Unwind

The unwind operator Ω is a unary operator that takes a list from a specified attribute χ of the input relation and multiplies each tuple individually, resolving elements of the list to a new attribute α. With an input relation R, the Cartesian product $\times$, and $t[\chi]$ returning the value of the tuple for the attribute χ, the unwind operator $\Omega_{\chi,\alpha}(\cdot)$ is defined as:

$$\begin{aligned} \Omega &: R \rightarrow R \\ \Omega_{\chi,\alpha}(R) &:= \bigcup_{\forall j \in t[\chi]} t \times j \\ \mathrm{sch}\big(\Omega_{\chi,\alpha}(R)\big) &:= \mathrm{sch}(R \backslash \chi) \cup (\alpha) \end{aligned} \tag{4.28}$$

4.3.2.3 Project

The projection operator restricts all tuples of the input relation to a specified set of attributes. All other attributes are discarded. With an input relation R over the attributes $(A_1, ..., A_N)$, β being a list of attributes with $\beta \subseteq (A_1, ..., A_N)$, and $\beta(t)$ reducing a tuple t to the list of attributes in β, the projection $\Pi_\beta(\cdot)$ is defined as:

$$\begin{aligned} \Pi &: R \rightarrow R \\ \Pi_\beta(R) &:= \{\beta(t) \mid t \in R\} \\ \mathrm{sch}(\Pi_\beta(R)) &:= \beta \end{aligned} \tag{4.29}$$

4.3.2.4 Match

The match operator Θ is the central operator of the graph data model. It adds two additional attributes to the graph relation if and only if there is an accordingly directed and labeled edge to an accordingly labeled node. It has five parameters: d is the direction to match $d \in \{\uparrow, \downarrow, \updownarrow\}$, χ is the name of an attribute of the input relation R containing nodes, α_n and α_e are the names of the two attributes to be added to the graph relation, β_n is a list of node labels $\beta_n \subseteq L_N$, and β_e is a list of edge labels $\beta_e \subseteq L_E$. The operator is denoted as $\Theta_{d,\chi}^{\alpha_n,\beta_n}[\alpha_e, \beta_e]$.

The attributes α_n and α_e are added to those tuples of the output relation were there is an edge with the direction specified by d from the node provided by the attribute χ of the input relation R to a node in the graph G that has all labels in β_n. Furthermore, the connecting edge must have any of the labels in β_e.

$$
\begin{aligned}
\Theta &: G, R \rightarrow R \\
\Theta_{\uparrow,\chi}^{\alpha_n,\beta_n}[\alpha_e, \beta_e](G, R) &:= \Big\{ t \mid t[\chi], t[\alpha_n], t[\alpha_e] \in G \wedge \ell_N(t[\alpha_n]) = \beta_n \wedge \ell_E(t[\alpha_e]) \in \beta_e \\
&\qquad \wedge \mathtt{src}(t[\alpha_e]) = t[\chi] \wedge \mathtt{tgt}(t[\alpha_e]) = t[\alpha_n] \Big\} \\
\Theta_{\downarrow,\chi}^{\alpha_n,\beta_n}[\alpha_e, \beta_e](G, R) &:= \Big\{ t \mid t[\chi], t[\alpha_n], t[\alpha_e] \in G \wedge \ell_N(t[\alpha_n]) = \beta_n \wedge \ell_E(t[\alpha_e]) \in \beta_e \\
&\qquad \wedge \mathtt{src}(t[\alpha_e]) = t[\alpha_n] \wedge \mathtt{tgt}(t[\alpha_e]) = t[\chi] \Big\} \\
\Theta_{\updownarrow,\chi}^{\alpha_n,\beta_n}[\alpha_e, \beta_e](G, R) &:= \Theta_{\uparrow,\chi}^{\alpha_n,\beta_n}[\alpha_e, \beta_e](G, R) \ \cup \ \Theta_{\downarrow,\chi}^{\alpha_n,\beta_n}[\alpha_e, \beta_e](G, R) \\
\mathtt{sch}\Big(\Theta_{d,\chi}^{\alpha_n,\beta_n}[\alpha_e, \beta_e](G, R)\Big) &:= \mathtt{sch}(R) \cup (\alpha_n, \alpha_e)
\end{aligned}
\tag{4.30}
$$

4.3.2.5 Filter

The unary filter operator Ξ removes all tuples from the input relation that do not meet the filter criterion. The filter criterion φ is a propositional formula consisting of attributes of the input relation R, constant values, comparison operators ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$), and the logical operators ($\wedge$ (and), $\vee$ (or), $\neg$ (negation)). With $\varphi(\cdot)$ being the Boolean function of the filter predicate φ, the filter operator Ξ is defined as:

$$
\begin{aligned}
\Xi &: R \rightarrow R \\
\Xi_{\varphi}(R) &:= \{ t \mid t \in R \wedge \varphi(t) = \mathsf{true} \} \\
\mathtt{sch}\big(\Xi_{\varphi}(R)\big) &:= \mathtt{sch}(R)
\end{aligned}
\tag{4.31}
$$

4.3.2.6 Union

The union operator $\cup$ takes two graph relations and combines them in one output relation. Both input relations must have the same schema. For the two input relations R_ℓ and R_r the union operator can be defined as:

$$\begin{aligned} \cup &: R, R \rightarrow R \\ R_\ell \cup R_r &:= \{t \mid t \in R_\ell \vee t \in R_r\} \\ \mathrm{sch}(R_\ell \cup R_r) &:= \mathrm{sch}(R_\ell) = \mathrm{sch}(R_r) \end{aligned} \tag{4.32}$$

4.3.3 Constraints on the Data

It is possible to constrain a labeled property graph by means of a graph schema. A graph schema defines the structure of the graph by specifying how nodes with certain labels need to be connected with edges having a certain label. Furthermore, it specifies the property keys of these nodes and edges and the data type of the corresponding property values. Since the LPG model does not require an explicit data model (i.e., it is schema-optional), a schema is considered as a constraint on the data.

As defined in Section 4.3.1, L_N is the set of node labels and L_E is the set of edge labels. For defining a graph schema S, we introduce a function ℓ_c that defines the allowed labels of an edge between a pair of node labels:

$$\ell_c : (L_N, L_N) \rightarrow L_E^* \quad \text{with} \quad L_E^* \subseteq L_E \tag{4.33}$$

Furthermore, we define the two functions $\mathtt{prop}(\cdot)$ and $\mathtt{type}(\cdot)$. As defined in Section 4.3.1, P_K is the set of property keys and P_V is the set of property values. The $\mathtt{prop}(\cdot)$ function specifies the set of properties (identified by a property key) valid for a node or edge label. The $\mathtt{type}(\cdot)$ function gives the data type for a pair of node or edge labels and a property key.

$$\begin{aligned} \mathtt{prop} &: (L_N \cup L_E) \rightarrow P_K^* \quad \text{with} \quad P_K^* \subseteq P_K \\ \mathtt{type} &: (L_N \cup L_E) \times P_K \rightarrow D \end{aligned} \tag{4.34}$$

The set D is the set of available data types. With this, the graph schema S can be defined as a tuple:

$$S := (L_N, L_E, \ell_c, \mathtt{prop}, \mathtt{type}) \tag{4.35}$$

Database Management System
[Origin: Data + Latin basus "low, mean, vile, menial, degrading, counterfeit."] A complex set of interrelational data structures allowing data to be lost in many convenient sequences while retaining a complete record of the logical relations between the missing items.

— Stan Kelly-Bootle, The Devil's DP Dictionary

5 On Database Systems

A *database management system* (DBMS) is a software to store, retrieve and manipulate data [GMUW13]. In literature, the term *database* is sometimes used to refer to the collection of data governed by the DBMS [EN16]. However, as already mentioned in the introduction, we follow the common practice of using the terms *database*, *database system* and *database management system* interchangeably. This avoids the ambiguity of terms such as "graph database", the established term for a database management system based on the graph data model. In this chapter, we introduce the foundations of database systems elementary to the concepts presented in this thesis.

5.1 Components of a Database System

While every database system has a slightly different architecture, there are some basic building blocks that can be found in most systems. These are depicted in Figure 5.1 as light gray. The components depicted in a darker gray can only be found in systems supporting transactions.

It should be mentioned, that these components might be sliced differently depending on the system. Also, the naming of the components is not standardized. We have also deliberately omitted components such as authentication or authorization, as these are not relevant in the context of this thesis. In this section, we describe this "smallest common denominator" of components of a database system. The order of the paragraphs follows the path a query takes through the components.

Query Interface. The query interface, sometimes also called *client communication manager* [HSH07], is responsible for interacting with the client. A client hereby refers to everything that *directly* interacts with this interface by submitting queries. This can be

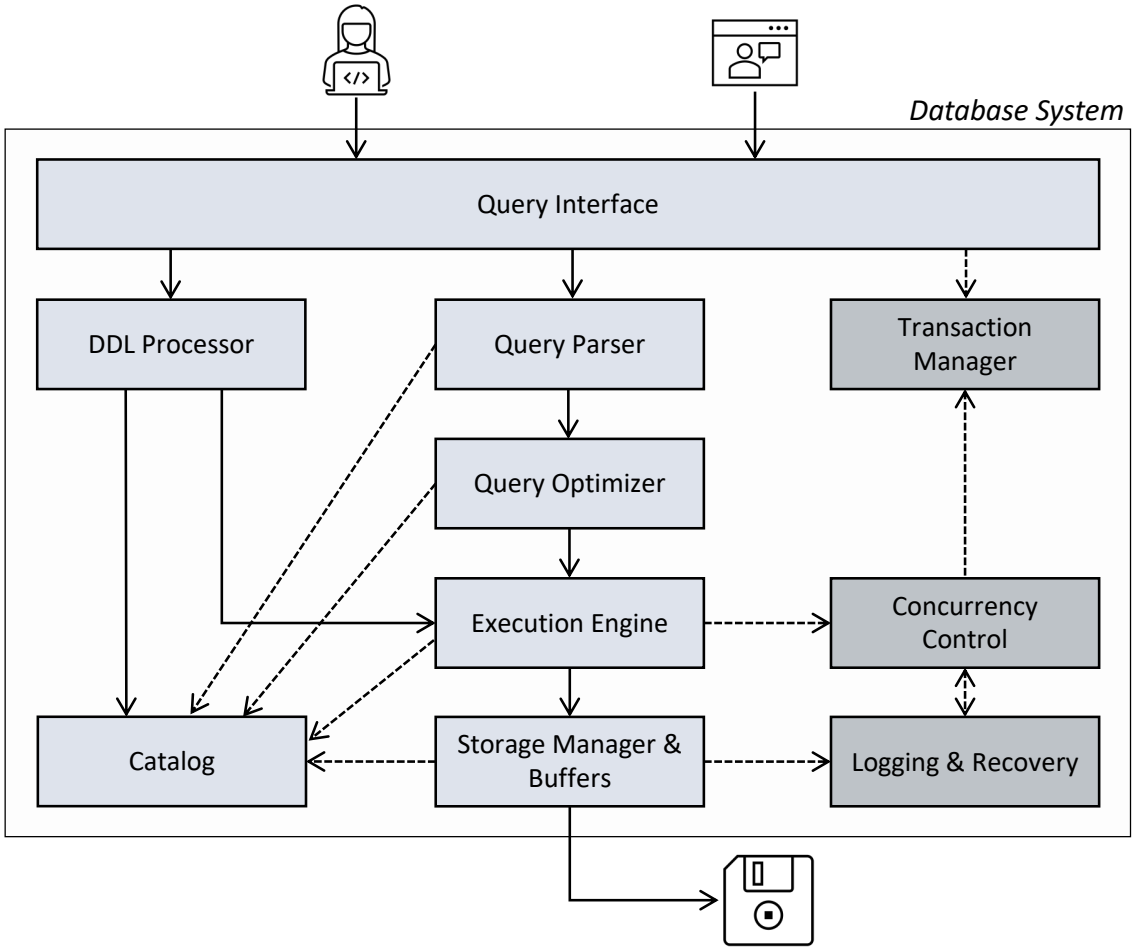


Figure 5.1 Basic building blocks of a database management system and their interactions. The components depicted in light gray can be found in every database system. The components depicted in darker gray only in systems supporting transactions. The solid lines indicate the path of a query. Dashed lines indicate interaction between components (simplified version of [EN16; GMUW13; HSH07]).

locally or over the network. Well-known standards for this kind of communication are the *Open Database Connectivity* (ODBC) or *Java Database Connectivity* (JDBC) protocols. Using these protocols, the client is able to submit queries to the database system and receive the corresponding results. The query itself is expressed using a query language such as the already mentioned *Structured Query Language* (SQL) or *openCypher*.

Query Parser. In the query parser, the query received from the client by the query interface is transformed into an abstract internal query representation which can be processed by the following components of the database management system. This internal representation does typically consist of the operators defined by the data model the database system is based on. In a relational database system, for example, the internal

representation is typically based on the relational algebra (see Section 4.1.2). In literature, the query parser is sometimes also called *query compiler*.

Query Optimizer. The purpose of the query optimizer is to determine the most optimal execution plan for the query received from the query parser. This is done be considering different variants for executing the query. The optimization typically includes the rearrangement and possibly also the reordering of operations and the elimination of redundancies [EN16]. Query optimization will be discussed in more detail in Section 5.2. The output of the query optimizer is an *execution plan*.

Execution Engine. In the execution engine, the query is actually executed. In modern systems, the execution plan is typically converted into native machine code that is then executed in a subsequent step [EN16]. The advantage of this approach is, that the generated code can be cached for subsequent executions of this query.

Storage Manager & Buffers. The storage manager controls the actual storage of the data and provides basic functions to access and manipulate it. These functions are used by the execution engine to interact with the data. The implementation of this component depends on the type of storage media for which the database system is designed. It might also cache recently used data.

Transaction Manager. The transaction manager maintains the state of the *database transactions*. It receives *transaction commands* from the query interface, allowing the client to begin and end transactions. Database transactions are discussed in more detail in Section 5.3.

Concurrency Control. In a system that processes queries from multiple clients, it can occur that several clients update the same data item in parallel. The concurrency control component ensures that this happens correctly. *Correctly* hereby means according to the guarantees set forth in Section 5.3. Concurrency control is discussed in more detail in Section 5.4.

Logging & Recovery. This component handles the case when something has gone wrong. If the database is in an inconsistent state on startup (e.g., after a crash), the recovery kicks in and brings the database back into a valid state. This component also allows aborting ongoing transactions (i.e., undoing the changes made in the context of this database transaction).

Catalog. The catalog stores the metadata on the data governed by the database system, including the schema. However, the amount of metadata depends on the data model the database system is based on.

DDL Processor. The DDL processor handles the execution of the *Data Definition Language*. DDL statements are commands to manipulate the schema or other aspects of the database system. The DDL processor updates the catalog and orchestrates the resulting changes to the data.

5.2 Query Optimization

In the query optimization, a query represented using the internal query representation of the database system is transformed into an execution plan. The goal is to find an execution plan that is optimal in terms of execution time (i.e., where the execution time is minimal). To find this optimal execution plan for a query, the optimizer can apply different techniques:

- rewrite the query according to equivalence rules for the operators of the data model (e.g., combine filter operations)
- change the order of operations (e.g., the order of joins)
- exchange operations (e.g., to use an index or a materialized view)
- eliminate unused expressions (e.g., operations of which the result is not used)
- remove redundant operations (e.g., redundant filter operations)

To find an optimal query plan, the optimizer needs a model for estimating the execution time of a query. Typically, such a model is called a *cost model*. It assigns a scalar cost value to an execution plan. These costs are determined based on statistics on the data and various heuristics.

An interesting approach for calculating the cost of a query plan is introduced in [SLM+01]. The authors introduce a system based on a feedback loop that adjusts the cost model based on the actual cost (i.e., execution time) of the query. A similar approach is introduced in [HG14] for the SPARQL query language.

5.3 Database Transactions

Transactions are a central concept in database systems [EN16]. A *transaction* represents a *unit of work* that should be treated as "a whole". A classical example for demonstrating

the concept of transactions is the transfer of money between two bank accounts. For the money to be transferred, it is first withdrawn from the account of origin and then deposited on the target account. However, if the database system fails in between, the money is lost. By executing both, the withdrawal and the deposit within one transaction, it is ensured that either both are executed or none.

In general, a transaction can be defined as a *sequence of atomic actions*. An *action* is either a transaction itself (i.e., nested transactions) or something that is already guaranteed to be executed atomic (i.e., executed as an indivisible unit). A transaction guarantees that either all actions are executed or none are.

A database system usually guarantees the atomic execution of a single query. From an application's point of view, a query can therefore be treated as an atomic action. Transactions allow grouping multiple queries and provide the application with an *all or nothing* guarantee from the database system. Furthermore, it also gives the application the ability to deliberately abort the transaction and undo the changes.

However, database transactions guarantee more than atomic execution. As introduced by Jim Gray in [Gra81], transactions represent a contract in terms of guarantees provided by the database system. This contract does not only include the atomic execution of the transaction, but also guarantees that the changes are persistent and that the execution of the transaction does not result in a state violating the constraints.

Based on the work of Jim Gray, the authors of [HR83] described these guarantees in more detail. Furthermore, they also introduced a fourth guarantee. These four guarantees known by the acronym *ACID* are:

- **Atomicity.** The atomicity guarantee requires that a transaction is executed as an atomic unit [EN16]. For a transaction to have an effect on the data, all actions within the transaction need to be successful.
- **Consistency.** A transaction needs to maintain the consistency of the data. Executing a transaction must not result in a state where any constraint defined by the schema is violated.
- **Isolation.** The concurrent execution of transactions must result in a state as if the transactions would have been executed sequentially. Ensuring this is the task of the *concurrency control*.
- **Durability.** Once a transaction has been completed, the changes performed by this transaction need to be persistent and survive failures.

5.4 Concurrency Control

Most database system support multiple users (i.e., clients, applications) to access the database at the same time. This is essential since it enables data from different applications to be integrated and stored in a single database [EN16]. However, concurrently executing transactions (that do not guarantee *isolation*), can cause the database state to become inconsistent, even when each transaction individually preserves the correctness of the database state (i.e., provides the *consistency* guarantee) [GMUW13].

Concurrency control techniques enforce the *isolation* between transactions and implement a certain level of "correct concurrent execution" [OV11]. This level represents a trade-off between the performance of the database system under concurrent workloads and the correctness and consistency of the database system. Weaker levels enable a higher performance, but may produce incorrect results or result in an inconsistent state of the database. Roughly, we can distinguish between three levels of isolation:

- **Serial.** All actions of the same transaction are executed consecutively, without any interleaving of actions from another transactions. Thus, there can only be one ongoing transaction in a database system at a time and therefore no concurrency. Hence, isolation of transactions is guaranteed.
- **Serializable.** In this level, concurrent workloads are allowed and there can be multiple ongoing transactions at the same time. The executions of actions from different transactions are interleaved. However, the outcome is equivalent to at least *one of the possible* serial executions of the transactions; thus, serializable also provides isolation between transactions.
- **Non-Serializable.** The interleaved execution of actions from different transactions may result in a state that cannot be obtained by a serial execution. Hence, the database may result in an inconsistent state. Isolation is not guaranteed.

A widely-adopted technique for achieving a serializable execution of actions is the *strong strict two-phase locking* (SS2PL). It uses locks to prevent concurrent transactions from accessing or modifying the same data. There are two types of locks: *shared locks* and *exclusive locks*. While the former only allow to read data, the latter also allow to modify it. As the name implies, a shared lock does not prevent other transactions to also acquire a shared lock. If a transaction cannot acquire a lock, it needs to wait until the lock is released. A mutual blocking of two or more transactions results in a deadlock that needs to be resolved by the DBMS (e.g., by aborting transactions).

A distributed system is one where the failure of some computer I've never heard of can keep me from getting my work done.

— Leslie Lamport

6

On Distributed Data Management

The discrepancy between the growth of data volume and the growth of computational power limits the possibilities of vertical scaling (i.e., adding more power to a machine). This leaves two possibilities for dealing with growing amounts of data: by developing more efficient algorithms or by horizontally scaling the system to multiple machines—or a combination of both approaches, which is what we propose in this thesis. While the data models presented in Chapter 4 introduced the foundations for the first approach, in this chapter, we introduce the foundations for the second approach.

Horizontal scaling means scaling by adding more nodes (i.e., computers, servers). This allows to parallelize the processing of workloads. However, distribution also introduces several new challenges. In the context of this thesis, there are two important topics: Firstly, the *management of resources* in such a distributed system. This includes the selection of distribution models (→ Section 6.1) and the efficient allocation of data to data stores (→ Section 6.2). Secondly, ensuring the *consistency* of the data using distributed transactions (→ Section 6.3) and dealing with data freshness (→ Section 6.4).

A major and also extensively studied problem in the context of distributed database systems is *concurrency control* [OV11]. However, due to the architecture of a PolyDBMS, all queries go through the PolyDBMS, acting as a central instance, allowing concurrency control to be handled solely in this central system, even though the queries are executed—at least partially—in one or multiple of the underlying data stores. As long as the concurrency control technique applied by the PolyDBMS imposes a strict execution order for conflicting transactions that all execution engines adhere to, concurrency control is identical to a non-distributed database system. Since the SS2PL technique introduced in the previous chapter guarantees such an order-preserving schedule, distributed concurrency control is not discussed further in this chapter.

Another major topic with distributed systems is the handling of failing nodes or network connections. The *CAP theorem* [GL02] introduced by Eric Brewer states, that in such an event, it is not possible to keep the system available and at the same time maintain the consistency of the data. The fundamental problem behind this is, that a node in a network cannot distinguish whether another node has failed or only the network link between the nodes (called a network partitioning). This is also the case for a PolyDBMS relying on underlying data stores deployed on different machines. However, due to the architecture of a PolyDBMS where queries are only accepted by a central instance, there is no immediate risk of inconsistencies. This would only be the case if other systems would directly interact with the underlying data stores. Nevertheless, strategies for handling failing data stores in a PolyDBMS or even having multiple instances of the PolyDBMS itself would be very interesting aspects. However, this aspect will not be considered further in the context of this thesis.

6.1 Distribution Models

In this thesis, we distinguish between *partitioning* and *allocation*. Partitioning is the process of dividing a schema object into multiple partitions. These partitions can be of different sizes. Allocation is the process of assigning partitions to physical storage nodes. If the same partition is allocated to multiple nodes, this is called *replication*. This is depicted in Figure 6.1. A schema object (e.g., a relation) is divided in five logical partitions. These partitions are allocated to three nodes (i.e., database systems). Except for the partition P1, all partitions are allocated twice; thus, they are replicated. Partition P1 is only stored on Node A and is therefore not replicated. However, replication does not require partitioning a schema object into multiple partitions. It is also possible to assign the whole schema object to multiple nodes.

It can be distinguished between two fundamental forms of partitioning [OV11]: *horizontal partitioning* and *vertical partitioning*. Both forms of partitioning split the data of a schema object into multiple parts. However, they differ in how the data is divided and how the schema of the partitions looks like.

In **horizontal partitioning**, a schema object is divided along its data. Each partition therefore consists of the full schema of the schema object (e.g., all attributes of a relation) but only a subset of the data (e.g., the tuples of a relation). The mapping of data items to partition can either be specified explicitly (e.g., by lists of values for every partition) or is done based on a round-robin or hash based approach.

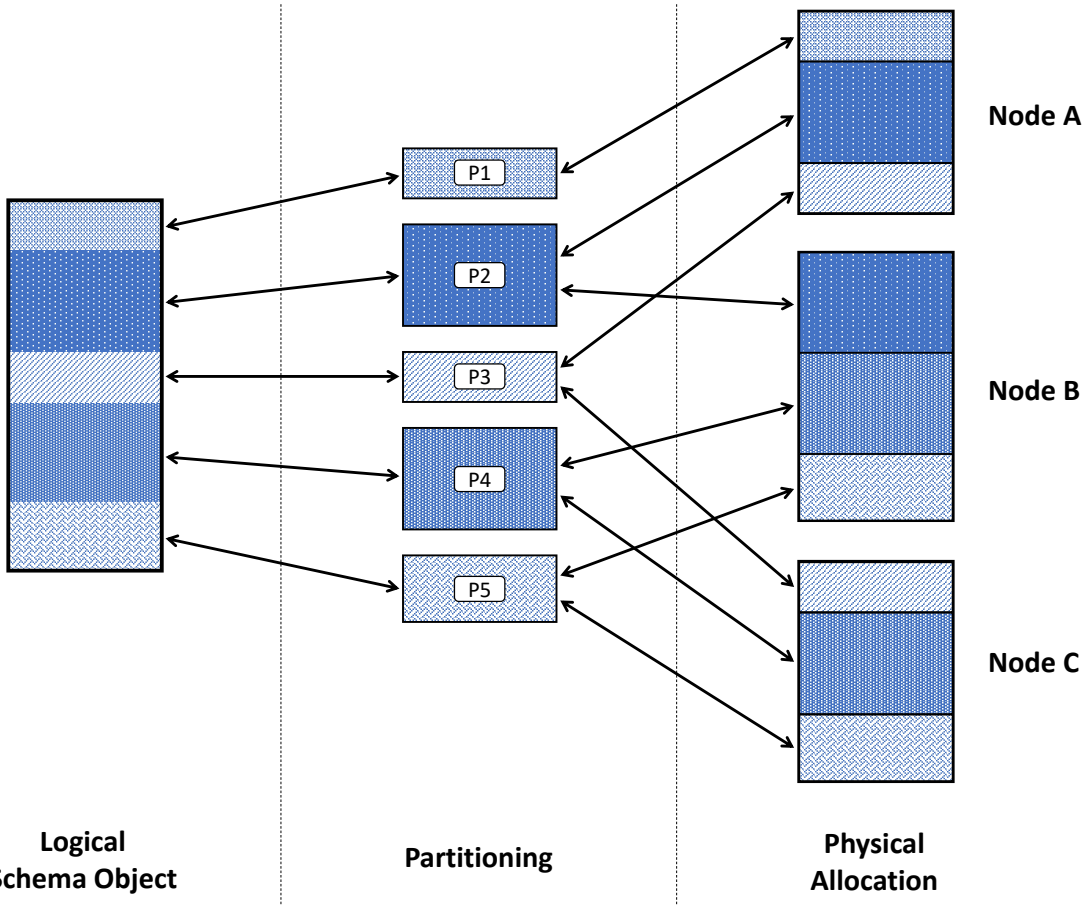


Figure 6.1 Data partitioning and allocation (based on [Dad96]).

The **vertical partitioning** splits the schema of a schema object into partitions containing a subset of the schema (e.g., a subset of the attributes of a relation) and only the data belonging to this subset of the schema (e.g., the data for these attributes of the relation). In order to recombine the vertical partitions, there needs to be some redundantly stored data (e.g., the primary key of a relation) that uniquely identifies the individual data items of the partitions (e.g., the individual tuples of a relation).

It is also possible to combine vertical and horizontal partitioning. In literature, this is usually called *hybrid partitioning* [Dad96]. Furthermore, both forms of partitioning can be combined with replication [OV11]. In literature, partitioning is oft also called fragmentation [Dad96].

The **allocation problem** refers to the issue of finding the optimal distribution of partitions to nodes [OV11]. This heavily depends on the data and the workload. However, based on the exhaustive analysis on the impact of data replication and data partitioning on the query performance presented in [NJ00], we can draw some general conclusions.

The performance of data modification queries is usually negatively affected by data replication. This makes sense since changes need to be applied to all replicas. With partitioning, it depends on the workload: if queries only modify individual partitions, it can have a positive impact on the performance of data modification queries.

Read-only queries typically benefit from data replication. The same applies for partitioning if only one or very few partitions are queried. If several partitions need to be accessed, this has a negative impact on the performance of the query. In general, it can be concluded that horizontal partitioning is beneficial for OLTP and disadvantageous for OLAP workloads.

6.2 Temperature-aware Data Management

The storage device has a major impact on the performance of a database system [OH11]. Traditionally, database systems are built based on a *two-layer architecture* [Hä05] with HDDs as persistence storage layer and the main memory as buffer and cache. In recent years, HDDs have been replaced more and more by *solid-state drives* (SSDs). While SDDs provide a higher read/write speed, they are also significantly more expensive. Together with even faster and more expensive storage technologies like *non-volatile random-access memory* (NVRAM) and very slow but also cheap archival storage systems, database administrators can choose from a wide range of storage technologies—depending on their budget.

However, there is a problem: as outlined by the data backup service provider BackBlaze in a blog post [Kle17], the amount of data is growing much faster than the cost of storage devices is decreasing. As a consequence, even maintaining current performance is becoming more and more expensive due to the constant growth of data. *Temperature-aware data management* is a technique to increase the overall performance, while at the same time reducing storage costs. It makes use of the fact that in many scenarios, only a fraction of the data is heavily used.

Research conducted by the database company Teradata has shown that 85% of the I/O operations use the same 10% of data [Gra12]. While this is of course only an average that depends on the use case, the data has been gathered from productive systems and shows a huge potential for optimization.

In the data management community it is common practice to describe the access frequency of a certain data set by a *temperature* [GP87]. Data that is currently being accessed very

frequently is referred to as *hot*, while data that has not been accessed for a long time is considered *cold*. In between, there can be various shades of *warm*. Temperature-aware data management refers to the approach of storing hot data on fast (and expensive) storage, warm data on somewhat slower storage, and cold data on slow but also very cheap storage.

There are various approaches for calculating the temperature of a data item. The authors of [LLS13] suggest logging the accesses to a data item and performing an offline analysis to estimate the access frequency. In [PD11], techniques for identifying hot and cold data using multiple Bloom filters and in [PDN+12] using sampling-based techniques are presented. Hot and cold data identification is also a topic of extensive discussion in the context of flash memory devices. The authors of [HKC06] present an online approach to hot data identification using multiple hash functions. Their work also includes an analytical study of the probability of false identifications.

6.3 Distributed Transactions

A distributed transaction is a transaction that involves two or more participants. These transactions also need to provide the ACID properties outlined in Section 5.3. The correct completion of distributed transactions is ensured using *consensus algorithms*. A major consensus algorithm widely used in the context of distributed database systems is the *two-phase-commit* (2PC) protocol. Other well-known consensus algorithms are *RAFT* [OO14] and *Paxos* [Lam98]. In this thesis, we focus on the 2PC algorithm.

The 2PC protocol as described in [WV02] models transactions over multiple database systems which can be deployed on different nodes of a network. There are two roles in the 2PC protocol: the *coordinator* and the *participants*. The coordinator role is typically taken by the node where the transaction has been initialized [Lec18]. This coordinator is responsible for steering the correct execution of the transaction. All database systems involved in the transaction are called participants. The possible states and transitions of the coordinator and the participants are depicted in Figure 6.2.

As the name implies, the 2PC protocol consists of two phases: the *voting phase* and the *commit phase*. In the voting phase, the coordinator requests all participants to “prepare”. If the participants are able to commit, they vote with “yes”. Otherwise, they vote with “no”. In the commit phase, the coordinator decides based on the votes whether to commit or to abort the transaction. However, the transaction can only be committed if all participants have voted with “yes”. The coordinator informs the participants about its decision. The

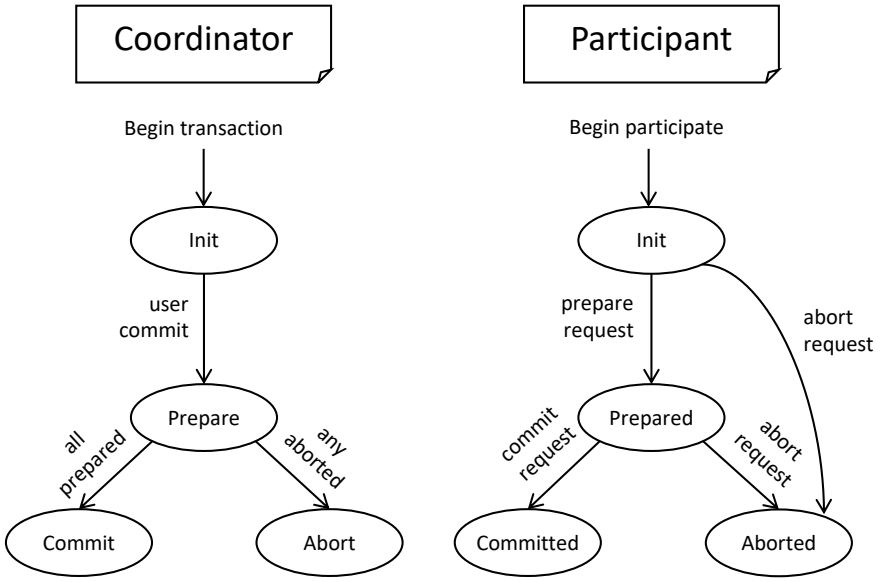


Figure 6.2 Possible states of the coordinator and the participants in the two-phase-commit protocol (based on [Wan21]).

participants then execute this decision. Since the participants already confirmed in the previous phase that they are able to commit, it is guaranteed that the commit happens on all nodes involved in the transactions.

Without such a commit protocol, it could happen that a node is unable to commit a transaction (e.g., because it violates a constraint). This would lead to inconsistencies since changes would be committed on some nodes but not all nodes; a violation of the *atomicity* requirement of ACID.

6.4 Data Freshness

If data is replicated in a distributed system, it can be distinguished between two forms of *update propagation*: *eager* and *lazy*. These describe when changes to a data item get applied to the nodes where a copy of the partition is physically allocated.

In systems where updates get propagated **eagerly**, all copies are modified immediately. Subsequent queries can therefore read from any of the nodes and always get the latest data. However, there are also techniques which defer the propagation to the commit time instead of applying the change immediately on all nodes [OV11]. In this thesis, we refer to eager propagation as synchronously applying changes to all nodes. The main disadvantage of an eager update propagation is the usually higher response time

of update queries since the query can not complete until all nodes have been updated. Hence, the update speed is restricted by the slowest node [OV11].

If updates get propagated **lazily**, changes are not necessarily performed immediately or even within the context of the transaction. The propagation of updates is done asynchronously from the original transaction [OV11]. Subsequent queries therefore might read outdated data. The main advantage of lazy update techniques is the usually drastically lower response time since only one node needs to be updated for the query to complete [OV11].

A consequence of systems applying a lazy update propagation strategy is that the distributed system might store multiple versions of the same data item. Nodes containing such outdated versions of a data item are said to have a lower *data freshness*. However, as outlined in [VSB+10] such slightly outdated data is acceptable for several applications. The authors argue that this can be exploited by keeping replicas at various levels of freshness and executing queries accepting this level of freshness on such outdated, not yet-to-be updated partitions. Such a combination of eager and lazy replication offers a great trade-off between the efficient usage of available resources and the response time of update queries.

There is no commonly agreed definition of **data freshness**. In [NLF99] and [Red96] it is described as the percentage of unchanged data items. According to [CGM00], the freshness of an object o_i at a time t can be expressed as:

$$F(o_i, t) := \begin{cases} 1 & \text{if } o_i \text{ is up-to-date at time } t \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

The average freshness of a schema object s (e.g., a table or a collection) at a time t is given by:

$$F(s, t) := \frac{1}{N} \sum_{i=1}^{N} F(o_i, t) \tag{6.2}$$

The authors of [CGM00] also introduce the notion of *age*, capturing the average time since the outdated objects have been updated the last time. The age of a object o_i at time t is given as:

$$A(o_i, t) := \begin{cases} 0 & \text{if } o_i \text{ is up-to-date at time } t \\ (t - \text{modification time of } o_i) & \text{otherwise} \end{cases} \tag{6.3}$$

With this, the average age of a schema object s at a time t is given by:

$$A(s, t) := \frac{1}{N} \sum_{i=1}^{N} A(o_i, t) \tag{6.4}$$

Another metric for measuring freshness is the number of updates a data item or schema object is behind the latest version. By [BP04], this is called *obsolescence metric*. The authors also introduce a *timeliness metric*, that takes the update frequency of the schema object into account. Instead of calculating the actual "outdatedness", it is estimated based on the time elapsed since the last update of the node in relation to the update frequency of the schema object.

Part III

PolyDBMS

Preface to Part III:
Dare to Try Something New

In the first part of this thesis, we have motivated the need for a new kind of database system that is capable of efficiently handling heterogeneous data and workloads. Moreover, we concluded that a hybrid architecture combining the advantages of a monolithic "one-size-fits-all" system and a middleware system is the most viable option. Based on this architecture, we have identified three areas where additional research is required:

- **Schema Model.** Combining different data models in one logical schema that preserves the semantics of the individual models while at the same time enabling cross-model queries.
- **Query Representation.** Representing queries based on different data models in a way that preserves the features and characteristics of the data models and enables cross-model queries.
- **Query Routing.** Adaptively planning the decomposition of queries and their assignment to execution engines to exploit the benefits of the heterogeneous execution and storage engines.

In this part of the thesis, we present our conceptual contributions to these three areas of research. The presented conceptual models and approaches build upon the foundations introduced in Part II and are independent of a specific implementation.

Besides the conceptual contributions, we also introduce **Polypheny-DB**, a fully functional implementation of a PolyDBMS and the concepts introduced in this thesis. We present the system's architecture and provide an overview on the available interfaces, languages and adapters.

In more detail, this part is structured as follows: In **Chapter 7**, we discuss the hybrid architecture model and motivate the selection of data models we are going to use to exemplify the concepts. The **Chapters 8**, **9**, and **10** present our conceptual contributions to the aforementioned areas of research. In **Chapter 11**, we introduce Polypheny-DB.

We keep moving forward, opening new doors, and doing new things, because we are curious and curiosity keeps leading us down new paths.

— Walt Disney

7

Anatomy of a PolyDBMS

Dealing with the demand for a tight integration of heterogeneous data that is structured according to different data models, and querying this data using different query languages and mixed transactional and analytical workloads, requires a new kind of database management system. In Chapter 3, we have concluded the demand for a new kind of database management system that we have called *PolyDBMS*. The idea of this novel approach is to combine the advantages of HTAP systems, polystores, and multimodel databases in one system. This new class of system needs to be a full-fledged database management system and, at the same time, meet the requirements outlined in Section 3.1.

Also, from an architectural point of view, our approach rethinks traditional approaches. Instead of building a monolithic system or following the idea of a federated database system and building a middleware (similar to most polystore systems), we opt for a combination of both approaches. As depicted in Figure 7.1, a PolyDBMS accepts queries expressed in different query languages and based on different data models. According to the idea of a federated database system, underlying database systems can be utilized to execute the queries. However, there is also an execution engine integrated within the PolyDBMS that allows the processing of queries similar to a monolithic database system.

The idea of having something like an execution engine in a polystore-like system is not new. However, what is new is the scope of this engine. Existing approaches follow the idea of federated database systems as a means to replace data warehouses. Their execution engine is used to combine results from different underlying database systems. In a PolyDBMS, the task of the execution engine goes far beyond merging results: it is an engine comparable with that of a monolithic system, able to handle all data models, operators and functions supported by the PolyDBMS.

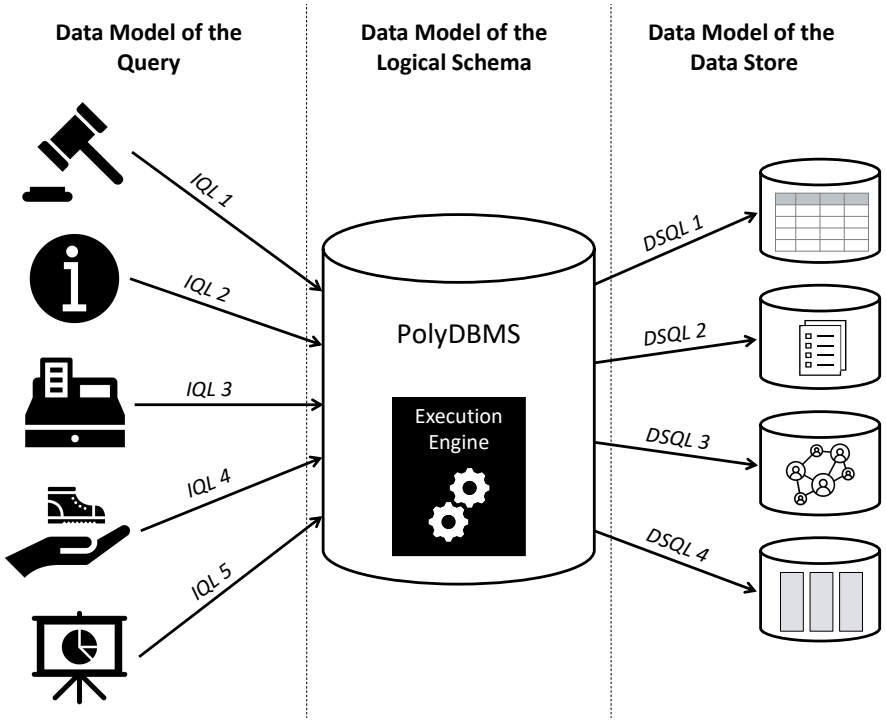


Figure 7.1 Data models in a PolyDBMS. Queries expressed using different input query languages (IQL) and based on different data models are accepted by the PolyDBMS. Every IQL can access the whole logical schema. The data is stored on data stores based on different data models. These data stores and the integrated execution engine are used to process queries.

Another key difference to existing approaches is the role of this new class of system. Polystore systems are pursuing the idea of replacing data warehouses and enabling analytical queries across heterogeneous data stores (cf. Stonebraker [Sto15]). This does not require support for data modification queries. The transactional workloads of the applications are executed directly on the database systems by the applications. In contrast, a PolyDBMS handles all types of workloads and is used by all applications to retrieve and manipulate data. For the applications, the PolyDBMS appears and behaves like a monolithic database system, hiding the underlying database systems. While these underlying database systems might be deployed on different physical machines, they are expected to be under the exclusive control of the PolyDBMS. In what follows, we will refer to this underlying database systems as *data stores*.

By combining the advantages of heterogeneous data stores and utilizing them as execution engines, the PolyDBMS is able to efficiently handle heterogeneous workloads. An integral part of this architecture is the ability to replicate and partition data across these data stores. This enables the system to be optimized according to the current workload. However, distributing the data across multiple data stores requires maintaining a logical schema on the PolyDBMS. Furthermore, this schema needs to track the location of

partitions and enable a consistent mapping to the schemas of the data stores. Due to this architecture, the PolyDBMS also needs to take care of other aspects like concurrency control, transactions, and the enforcement of constraints.

According to the Requirement 3.4 for a PolyDBMS, support for cross-model queries is necessary. Due to the architecture, this encompasses the handling of the potential heterogeneity of the logical schema and the heterogeneity of the underlying data stores. A query based on one data model can therefore join data logically governed according to other data models, while the data is physically distributed across data stores based on yet other data models. As outlined in Section 3.3, this requires a conceptual model for maintaining such a schema, including defined mappings between data models. Furthermore, a conceptual model for representing these heterogeneous queries is required.

Both the schema model and the model for representing queries depend on the supported set of data models. Since it is not feasible to define these conceptual models for every conceivable data model, we must limit ourselves to a selection on the basis of which we introduce and exemplify the concepts. For this, we have identified three key characteristics of a data model that are important for defining the schema and query representation model:

Flexibility. The ability of the data model to deal with unstructured data; thus the lack of a strict schema. The lack of a schema is more difficult to handle than a strict schema. Consistently and reproducibly mapping schema-less data to different data models for storage and cross model queries can be challenging.

Relationships. Expressing relationships and incorporating the explicit or implicit ability to join or combine data. Furthermore, the amount of information that can be encoded within these relationships.

Partitioning. The inherent possibilities to partition data governed by this data model. This indirectly assesses the ability of the data model to deal with unconnected data. Further, it assesses the data models ability to combine independent data using explicit operations.

Figure 7.2 depicts these characteristics as dimensions and positions well-known data models according to these characteristics. The distance to the axis thereby approximates the sophistication of this data model in terms of this characteristic. As it can be seen in the figure, there are three data models that are most distinct: the document model, relational model, and the labeled property graph model. In the figure, these three models are depicted with a blue dot.

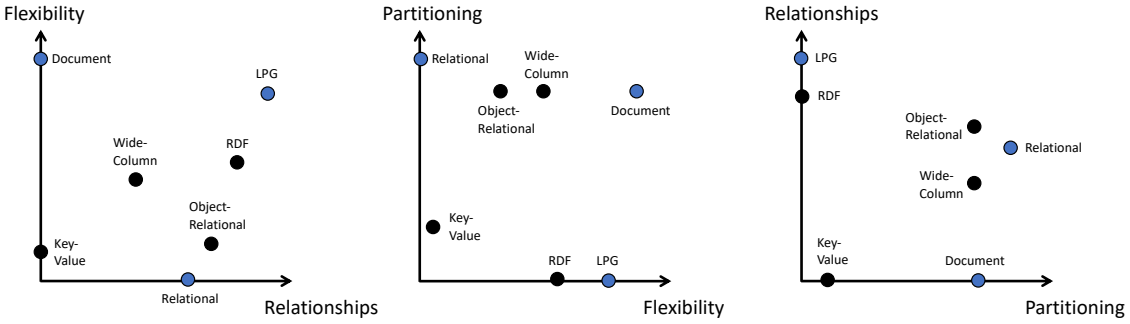


Figure 7.2 Different data models positioned according to their inherent ability to represent unstructured data (*flexibility*), connections between data (*relationships*), and independent data (*partitioning*).

The figure shows two important aspects: Firstly, which data model is the most sophisticated in terms of a specific characteristic. As it turns out, the document model has the greatest ability to represent unstructured data, the LPG model is the most sophisticated model for representing relationships, and in terms of partitioning the data, the relational model offers the most options.

The second and maybe even more important aspect revealed by the figure is the lack of a certain ability. Hence, for which data models it is most difficult to represent this characteristic and provide a certain ability. As it can be seen in the figure, the three data models marked in blue also stand out in this regard: while one of the data models always stands out for both mapped characteristics as sophisticated in combining these aspects, the other two data models are placed on the orthogonal axes (thus only being supported in terms of one of the depicted characteristics).

By introducing the schema model (Chapter 8) and query representation model (Chapter 9) for the relational, document, and LPG model, we make sure to cover the three cornerstones. Other common data models should therefore be integrable as well.

All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer.

— IBM Manual (1925)

8

Schema Model

The *schema* of a database system is the blueprint of how the data is structured in the database (see Chapter 4). The level of detail and the available building blocks for defining a schema depends on the data model. Since the schema is usually defined by a database administrator depending on the specific use case and might also be subject to change over time, a database system provides the means for defining and altering schemas at runtime. In this chapter, a conceptual model for expressing and managing schemas in a PolyDBMS is presented.

In Chapter 4 three data models have been introduced. These data models are fundamentally different in terms of the semantic building blocks provided to define schemas and, more importantly, in terms of the intrinsic role of such a schema. The relational data model enforces and requires a strictly defined schema. The document model is the opposite, enforcing and requiring no schema at all. In the LPG model, the schema is an optional constraint that can be applied to the data. Since a PolyDBMS needs to support multiple data models (see Section 3.1.1), a conceptual model for building schemas based on different data models is required (see Section 3.1.4).

The schema model being presented in this chapter is based on the three data models introduced in Chapter 4. However, as elaborated in Chapter 7, the relational, document, and LPG model are not only very popular and widely used database data models, but they are also conceptually very distinct. With the conceptual model in this chapter exemplified for these three data models, other popular data models like the wide-column, key-value, RDF, or object-relational model can be accommodated accordingly.

In this chapter, we use a semi-mathematical notation. This allows a concrete and formal discussion of the concepts while still maintaining a high degree of abstraction from an

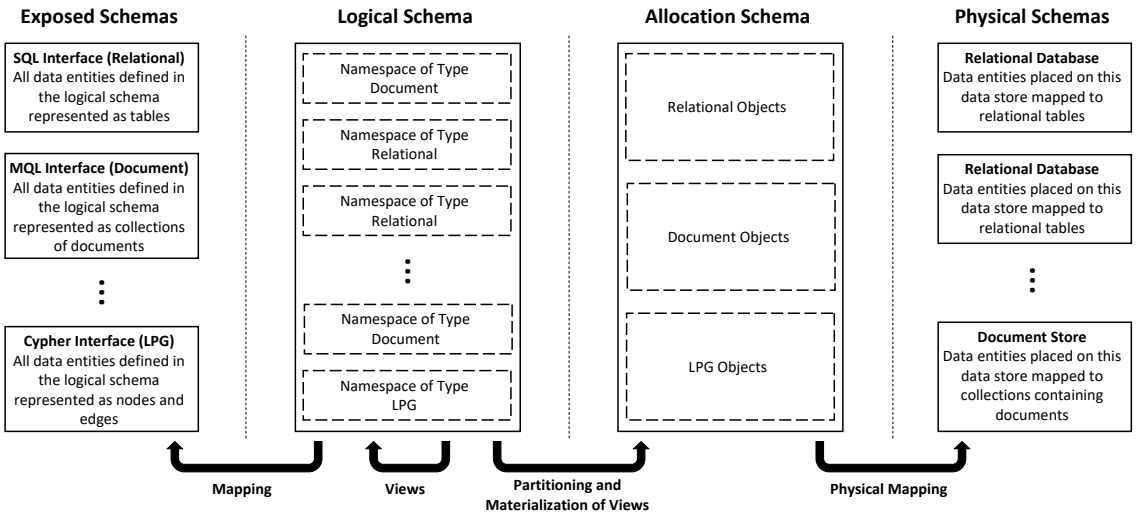


Figure 8.1 Overview of the schema model introduced in this thesis. There are four kinds of schemas and four mappings between and within these individual layers of the schema model.

actual implementation. A strict definition of the data models has already been introduced in Chapter 4.

8.1 Overview

The schema model introduced in this thesis goes beyond describing an approach for representing multimodel schemas; it defines a multi-layer concept distinguishing between different kinds of schemas with multiple data models at each layer. Furthermore, it defines the mappings between these different kinds of schemas. Figure 8.1 gives an overview on our schema model for PolyDBMS systems. It is distinguished between four kinds of schemas (i.e., layers):

Logical Schema. The central schema of the PolyDBMS accommodating different data models. A cornerstone of this concept are the *namespaces*. Every namespace has a specific data model. The building blocks (thus the semantic concepts like tables, collections, nodes, etc.) that can be defined within a namespace correspond to this data model. Additionally, a namespace can also contain views. These views can access entities from the same or other namespaces. If the query defining the view accesses entities from a namespace with a different data model, these entities are mapped to the data model of the namespace in which the view is located. The virtual entity defined by a view always corresponds to the semantic concepts of the namespace it resides in. Furthermore, the logical schema also

allows defining constraints on the data. The set of available building blocks for defining constraints depends on the data model of the namespace.

Allocation Schema. This schema is derived by applying horizontal data partitioning and converting materialized views into entities with a certain freshness. Partitioned entities are resolved into multiple entities. This schema also represents schema elements that are hidden from the user (e.g., internal tuple identifiers).

Physical Schemas. The schemas materialized on the underlying data stores. Our schema concept is based on a Local-as-View approach. A data store contains a subset of the schema defined in the allocation layer. As a result of the PolyDBMS Requirement 3.3, every entity (i.e., table, collection, graph, etc.) can be placed on every data store. If the data model of an entity does not match the data model of the data store, an appropriate mapping is performed.

Exposed Schemas. A view on the logical schema that provides access to all data entities defined in the logical schema. Namespaces with a different data model than the one of the query interface are mapped accordingly. The data model depends on the query language. The schema exposed on a SQL interface, for instance, represents everything as relational tables while for example on a Cypher interface all entities are represented as nodes and edges. Depending on the query language, the schema might expose a hierarchical naming structure to separate entities from different namespaces.

This layered architecture of multiple schemas does not only allow fulfilling the PolyDBMS requirements, it also comes with several advantages: The separation between the logical schema and the schema exposed to the clients allows a virtual mapping between data models, enabling access to data that is structured according to a schema based on a data model different from that of the query language. Furthermore, it also enables cross-model queries. The exposed schema can be represented according to the requirements and standards of each query language individually. At the same time, the logical schema serves as the reference schema providing the full capabilities and semantics of the data model. The concept of namespaces allows accommodating different data models beneath a single schema.

The option to define views accessing entities defined in namespaces with different data models adds a second approach for modeling cross-model queries. A view is a schema object that represents the result of an arbitrary, read-only query expressed in any of the query languages supported by the PolyDBMS. Our schema model thus provides two approaches for mapping data between data models: Firstly, there is a defined mapping for queries accessing schema objects of a namespace with a different data model and

secondly, there is the view based mapping. The difference between these two approaches is, that the first approach maps the schema to the data model of the query language and then applies the query, while the view-based approach does the mapping on the result represented by the view (if the query defining the view is based on a data model different to that of the namespace). However, the schema mapping strategy itself is the same in both cases. In what follows, we refer to this mapping strategy as *virtual mapping*.

The virtual mapping enables querying other data models by expressing their semantic concepts using the building blocks provided by the accessing data model. The mapping is done within the context of a query and is designed to represent semantic concepts across data models. The *physical mapping* in contrast is applied to entire data entities. It is optimized for an efficient storage and processing of the data stored on data stores based on different data models.

The query defining a view can also perform cross-model queries accessing different namespaces and might also access other views. Querying a view can therefore require multiple mappings to be performed within a single query. Our schema model therefore also provides the notion of materialized views. This allows to materialize the result of such a view. Since the allocation model treats materialized views similar as other schema objects, they can be physically stored on an arbitrary set of data stores.

The layered architecture of the schema model introduced in this chapter enables arbitrary combinations of query languages, logical schemas and physical data stores. For example, a query expressed in SQL can join multiple document collections that are physically stored on a data store based on the graph data model. Due to the allocation schema, this data can also be partitioned across multiple heterogeneous data stores. At the same time, the schema model introduced in this chapter ensures that the semantics of the individual data model are fully preserved. Furthermore, the schema model also achieves a full abstraction of the exposed schema from the physical schema. The schema against which a query is specified by the user is therefore completely decoupled from the physical structure in which the data is stored, possibly across multiple heterogeneous data stores.

The discussion of the schema model introduced in this chapter is structured in four sections: The first section formally introduces the building blocks of the *logical schema* and its properties. In the second section, we define the *virtual mapping* strategy used to derive the exposed schema and for enabling cross-model views within the logical schema. In the third section, we introduce the *physical mapping* strategy required for mapping schemas to underlying data stores with a different data model. The fourth section then

formally introduces the *data allocation concept* that enables the flexibility to partition and replicate data across heterogeneous data stores.

8.2 Logical Schema

The logical schema is the central schema of the PolyDBMS. It maintains the schema defined by the user and specifies the structure according to which the data is being organized. All other schemas are derived from this schema through mappings and transformations. Since a PolyDBMS supports multiple data models, an approach is required for accommodating schema definitions based on different data models in one logical schema. We achieve this using the concept of *namespaces*.

The logical schema $\mathcal{S}$ of a PolyDBMS is a set consisting of a finite number ($s \in \mathbb{N}$) of namespaces $\mathcal{N}$:

$$\mathcal{S} := \{\mathcal{N}_1, ..., \mathcal{N}_s\} \tag{8.1}$$

Every namespace $\mathcal{N}$ has a unique name and is of a specific data model $\mathcal{M}$ with $\mathcal{M} := \{\text{REL}, \text{DOC}, \text{LPG}\}$. The symbol $\mathcal{N}^{\text{REL}}$, for example, denotes a namespace of type relational.

8.2.1 Relational Model

A namespace of the model relational $\mathcal{N}^{\text{REL}}$ consists of a finite set of tables and a finite set of views. Tables and views have names which are unique within this namespace. A relational namespace $\mathcal{N}^{\text{REL}}$ is defined as a triple:

$$\mathcal{N}^{\text{REL}} := \left(\text{name},\ \{\mathcal{T}_1, ..., \mathcal{T}_n\},\ \{\mathcal{V}_1^{\text{REL}}, ..., \mathcal{V}_m^{\text{REL}}\}\right) \quad \text{with } n, m \in \mathbb{N} \tag{8.2}$$

A table $\mathcal{T}$ corresponds to a relation in the relational model introduced in Section 4.1. However, in contrast to the set-semantic of relations which allows no duplicate tuples, we define tables as a bag of tuples. We furthermore refer to the tuples as records. A table is defined over an ordered, non-empty list of columns $(COL_1, ..., COL_i)$ and a finite set of constraints $\{CSTR_1, ..., CSTR_j\}$. A table $\mathcal{T}$ can be described as a triple:

$$\mathcal{T} := \left(\text{name},\ (COL_1, ..., COL_i),\ \{CSTR_1, ..., CSTR_j\}\right) \quad \text{with } i, j \in \mathbb{N}, i > 0 \tag{8.3}$$

A column corresponds to an attribute in the relational model. Every column COL has a unique name within the table, a data type, a set of data type properties, and a nullability

information. The data type corresponds to the data domain; however, in contrast to the formal relational model, composite data types allowing non-atomic values are possible as well.

$$COL := (\texttt{name},\ \texttt{type},\ \texttt{properties},\ \texttt{nullable}) \tag{8.4}$$

A constraint *CSTR* has a name that is unique within the namespace and is of a specific type. Types are *primary*, *foreign* and *unique*. A primary key constraint $CSTR^{\text{PK}}$ specifies a set of columns of the table whose values uniquely identify each record in the table. The set of columns is represented by a key.

$$\begin{aligned} \mathcal{KEY} &:= \{COL_1, ..., COL_k\} \quad \text{with } k \in \mathbb{N}, k \leq i \\ CSTR^{\text{PK}} &:= (\texttt{name},\ \mathcal{KEY}) \end{aligned} \tag{8.5}$$

The uniqueness criterion imposed by this constraint is enforced by the PolyDBMS. The columns building the primary key must not be nullable (i.e., are not allowed to contain null values). Furthermore, there can only be one $CSTR^{\text{PK}}$ per table. To enforce uniqueness on another set of columns, uniqueness constraints can be defined. There can be multiple uniqueness constraints $CSTR^{\text{UNIQUE}}$ for a table. In contrast to primary constraints, the key on which the unique constraint is defined may consist of columns allowing null values.

$$CSTR^{\text{UNIQUE}} := (\texttt{name},\ \mathcal{KEY}) \tag{8.6}$$

Foreign key constraints express a relationship between two tables of a relational namespace. A foreign key $CSTR^{\text{FK}}$ is defined by a *local key* and a *remote key*. The local key specifies a set of columns in the table for which the constraint is defined. The remote key specifies a key in the same or another table. There needs to be a unique or primary key constraint for the remote key but not on the local key.

$$CSTR^{\text{FK}} := (\texttt{name},\ \mathcal{KEY}_{\text{LOCAL}},\ \mathcal{KEY}_{\text{REMOTE}}) \quad \text{with } \mathcal{KEY}_{\text{LOCAL}} \neq \mathcal{KEY}_{\text{REMOTE}} \tag{8.7}$$

In the relational model, a view $\mathcal{V}^{\text{REL}}$ is modeled as a special kind of table, representing the result of a query. Similar to a table, it has a non-empty list of columns. However, there are no constraints.

$$\mathcal{V}^{\text{REL}} := (\texttt{name},\ \texttt{query},\ (COL_1, ..., COL_g)) \quad \text{with } g \in \mathbb{N} \tag{8.8}$$

8.2.2 Document Model

A namespace of the type document $\mathcal{N}^{\text{DOC}}$ contains a set of collections and views. Both the collection C and the view $\mathcal{V}^{\text{DOC}}$ have a unique name within the namespace.

$$\mathcal{N}^{\text{DOC}} := (\texttt{name},\ \{C_1, ..., C_i\}, \{\mathcal{V}_1^{\text{DOC}}, ..., \mathcal{V}_k^{\text{DOC}}\}) \quad \text{with } i, k \in \mathbb{N} \tag{8.9}$$

According to the definition introduced in Section 4.2, the document model does not impose any schema, except that every document needs to have an _id field. A field thereby refers to an edge in the model defined in Section 4.2. However, we extend this and introduce the concept of *defined fields*. This allows to define additional fields besides the _id that need to be present on every document. Furthermore, the fields also serve a second purpose: they allow for a vertical partitioning of collections (see Section 8.5). With `label` being a name, `required` being a Boolean specifying whether the field is required on every document, `type` being a data type, and `properties` a set of data type properties, a collection is defined as:

$$\begin{aligned} F &:= (\texttt{label},\ \texttt{required}) \\ F^{\mathrm{T}} &:= (\texttt{label},\ \texttt{type},\ \texttt{properties},\ \texttt{nullable},\ \texttt{required}) \\ C &:= (\text{name},\ \{F_1, ..., F_n\},\ \{F_1^{\mathrm{T}}, ..., F_m^{\mathrm{T}}\}) \quad \text{with } n, m \in \mathbb{N} \end{aligned} \tag{8.10}$$

The set of typed fields at least contains an entry for the _id field. A view $\mathcal{V}^{\mathrm{DOC}}$ represents the result of an arbitrary query. It is similar to a collection; however, instead of a list of fields, there is an arbitrary read-only query.

$$\mathcal{V}^{\mathrm{DOC}} := (\text{name},\ \text{query}) \tag{8.11}$$

8.2.3 Labeled Property Graph Model

In contrast to our definition of a relational namespace or a document namespace, the LPG namespace does not consist of entity structures like tables or collections. Instead, the namespace itself represents one graph. Every node and every edge of this graph have a unique identifier. This identifier is randomly generated by the PolyDBMS system.

As outlined in Section 4.3, the LPG model can be described as schema-optional. This means, there can be a schema, but it is not mandatory. A schema can be enforced using a special graph. With $\mathcal{G}$ being such a graph defining a graph schema, a namespace $\mathcal{N}^{\mathrm{LPG}}$ is defined as:

$$\mathcal{N}^{\mathrm{LPG}} := (\text{name},\ \mathcal{G},\ \{\mathcal{V}_1^{\mathrm{LPG}}, ..., \mathcal{V}_k^{\mathrm{LPG}}\}) \quad \text{with } k \in \mathbb{N} \tag{8.12}$$

With ℓ being labels and $p := (\texttt{key},\ \texttt{type},\ \texttt{type\_properties})$ being a tupel of a property key and a data type specifications, the graph schema $\mathcal{G}$ is defined as:

$$\begin{aligned} P &:= \{p_1, ..., p_i\} \quad \text{with } i \in \mathbb{N} \\ g &:= (\ell_{\text{from}},\ \ell_{\text{to}},\ \ell_{\text{edge}},\ P_{\text{from}},\ P_{\text{to}},\ P_{\text{edge}}) \\ \mathcal{G} &:= \{g_1, ..., g_m\} \quad \text{with } m \in \mathbb{N} \end{aligned} \tag{8.13}$$

It specifies that every edge connecting a node with the label ℓ_{from} to a node with the label ℓ_{to}, can have the label ℓ_{edge}. Further, it specifies that in this case the node with the label ℓ_{from} must have the set of properties specified by the set P_{from}, the node with the label ℓ_{to} must have the set of properties specified by the set P_{to} and the edge must have the set of properties specified by the set P_{edge}. There can be multiple entries for the same pair of labels ℓ_{from} and ℓ_{to}. If there is a graph schema for a namespace, the graph must adhere to this schema.

Similar to the relational and document namespace, a view in the namespace $\mathcal{N}^{\text{LPG}}$ represents the result of an arbitrary query. A view $\mathcal{V}^{\text{LPG}}$ is defined as:

$$\mathcal{V}^{\text{LPG}} := (\texttt{name},\ \texttt{query}) \tag{8.14}$$

8.3 Virtual Mapping

An important part of the PolyDBMS concept is the ability to access data represented in different data models using any of the supported query languages. This requires a mapping between schemas based on different data models. In this section, we define the mappings between the relational, document, and LPG data models. The introduced mappings are designed so that they can be done on-the-fly and potentially be implemented using operators of the data model to be mapped.

Since the data model is defined by the namespace, an approach for accessing other namespaces is required. As it is already common in several query languages like SQL or the MongoDB query language, the hierarchy can for instance be expressed as a path using a language-specific separation character.

The arrows in the headings of the following sections indicate the direction in which the schema is being mapped. The section "Relational → Document" for example, specifies the strategy for mapping a relational schema to a document schema. This is for instance required when accessing a relational schema with a query language based on the document data model.

In the formal definitions of the schema mappings presented in this and the next section, we use $\pi_i(\cdot)$ to reference the i-th element of a tuple (with $i = 1$ being the first element of the tuple). For simplicity reasons and in order to keep the definitions concise, the formal definitions do not consider views. However, if not specified otherwise, the schema mappings can be applied identically to views.

8.3.1 Relational → Document

The mapping of a relational schema to a document schema is trivial: a table is represented as a collection. Every record of the table is represented as a document containing a field for each column of the table. The optional constraints of a relational table are not mapped, since they are enforced using the native form. More formally:

$$\begin{gathered} \mathcal{N}^{\text{REL}} \mapsto \mathcal{N}^{\text{DOC}} \\ \Big(\text{name},\ \{\mathcal{T}_1, ..., \mathcal{T}_n\},\ \{...\}\Big) \mapsto \Big(\pi_1(\mathcal{N}^{\text{REL}}),\ ①,\ \{\}\Big) \\ ① := \left\{\Big(\pi_1(\mathcal{T}),\ \{\},\ \texttt{dtf}(\mathcal{T})\Big) \;\middle|\; \mathcal{T} \in \pi_2(\mathcal{N}^{\text{REL}})\right\} \end{gathered} \tag{8.15}$$

The $\texttt{dtf}(\cdot)$ function that maps a relational table to a set of typed fields in the document model is defined as:

$$\texttt{dtf}(\mathcal{T}) := \left\{\Big(\pi_1(COL),\ \pi_2(COL),\ \pi_3(COL),\ \pi_4(COL),\ \text{true}\Big) \;\middle|\; COL \in \pi_2(\mathcal{T})\right\} \tag{8.16}$$

8.3.2 Relational → LPG

From the perspective of the LPG model, the tables of the relational model are viewed as labels. For every record of the table, a node is created having the name of the table as a label. Columns of the table are represented as properties of that node. The edges between the nodes are constructed based on the existing foreign key constraints. The name of the foreign key constraint is used as label for the edge. The edge points from the table referenced by $\mathcal{KEY}_{\text{LOCAL}}$ to the table referenced by $\mathcal{KEY}_{\text{REMOTE}}$.

$$\begin{gathered} \mathcal{N}^{\text{REL}} \mapsto \mathcal{N}^{\text{LPG}} \\ \Big(\text{name},\ \{\mathcal{T}_1, ..., \mathcal{T}_n\},\ \{...\}\Big) \mapsto \Big(\pi_1(\mathcal{N}^{\text{REL}}),\ \texttt{gs}(\mathcal{N}^{\text{REL}}),\ \{\}\Big) \end{gathered} \tag{8.17}$$

The set of node labels encompasses all table names

$$\left\{\pi_1(T) \;\middle|\; \mathcal{T} \in \pi_2(\mathcal{N}^{\text{REL}})\right\} \tag{8.18}$$

while the set of edge labels consists of the names of all foreign key constraints

$$\left\{\pi_1(\mathcal{CSTR}^{\text{FK}}) \;\middle|\; \mathcal{T} \in \pi_2(\mathcal{N}^{\text{REL}}) \ \wedge\ \mathcal{CSTR}^{\text{FK}} \in \pi_3(\mathcal{T})\right\}. \tag{8.19}$$

With $\texttt{tab}(\cdot)$ returning the table $\mathcal{T}$ referenced by a key $\mathcal{KEY}$, we can define the graph schema function $\texttt{gs}(\cdot)$. This function takes a relational namespace $\mathcal{N}^{\text{REL}}$ and returns a graph schema $\mathcal{G}$.

$$
\begin{aligned}
\mathtt{gs}(\mathcal{N}^{\mathrm{REL}}) &:= \left\{ \left(①, ②, ③, ④, ⑤, \{\} \right) \;\middle|\; \mathcal{T} \in \pi_2(\mathcal{N}^{\mathrm{REL}}) \;\wedge\; \mathit{CSTR}^{\mathrm{FK}} \in \pi_3(\mathcal{T}) \right\} \\
① &:= \pi_1\left(\mathtt{tab}\left(\pi_2(\mathit{CSTR}^{\mathrm{FK}}) \right) \right) \\
② &:= \pi_1\left(\mathtt{tab}\left(\pi_3(\mathit{CSTR}^{\mathrm{FK}}) \right) \right) \\
③ &:= \pi_1\left(\mathit{CSTR}^{\mathrm{FK}} \right) \\
④ &:= \left\{ \left(\pi_1(\mathit{COL}), \pi_2(\mathit{COL}), \pi_3(\mathit{COL}) \right) \;\middle|\; \mathit{COL} \in \pi_2\left(\mathtt{tab}\left(\pi_2(\mathit{CSTR}^{\mathrm{FK}}) \right) \right) \right\} \\
⑤ &:= \left\{ \left(\pi_1(\mathit{COL}), \pi_2(\mathit{COL}), \pi_3(\mathit{COL}) \right) \;\middle|\; \mathit{COL} \in \pi_2\left(\mathtt{tab}\left(\pi_3(\mathit{CSTR}^{\mathrm{FK}}) \right) \right) \right\}
\end{aligned}
\tag{8.20}
$$

This approach does not result in a graph that uses the full potential of the graph model since, for example, join tables required for modeling m-to-n relationships are not resolved and foreign key columns are not filtered from the list of properties. However, this transformation can be done within the query and does not require any artificial names to be generated.

8.3.3 Document → Relational

For this mapping, we use the previously introduced notion of *defined fields*. A collection is mapped to a table with the defined fields as columns. Furthermore, there is an additional column named _data containing all other fields and nested structures represented as JSON. Since it can be specified whether a defined field is required on every document, the table may contain null values for those columns where the required flag is not set. For typed fields, the specified data type is being used. For non-typed fields, the special data type JSON is being used. This type represents nested data as JSON.

$$
\begin{gathered}
\mathcal{N}^{\mathrm{DOC}} \mapsto \mathcal{N}^{\mathrm{REL}} \\
\left(\mathtt{name},\ \{C_1, ..., C_i\},\ \{...\} \right) \mapsto \left(\pi_1(\mathcal{N}^{\mathrm{DOC}}),\ \mathtt{tab}(\mathcal{N}^{\mathrm{DOC}}),\ \{\} \right)
\end{gathered}
\tag{8.21}
$$

The function $\mathtt{tab}(\cdot)$, mapping a document namespace $\mathcal{N}^{\mathrm{DOC}}$ to a set of relational tables $\{\mathcal{T}_1, ..., \mathcal{T}_i\}$ is defined as:

$$
\mathtt{tab}(\mathcal{N}^{\mathrm{DOC}}) := \left\{ \left(\pi_1(C),\ \mathtt{col}(C), \{\} \right) \;\middle|\; C \in \pi_2(\mathcal{N}^{\mathrm{DOC}}) \right\} \tag{8.22}
$$

The mapping function $\mathtt{col}(\cdot)$ is defined as follows:

$$
\begin{aligned}
\mathtt{col}(C) &:= ① \cup ② \cup \left\{ (\mathtt{\_data},\ \mathtt{JSON},\ \{\},\ \mathtt{true}) \right\} \\
① &:= \left\{ \left(\pi_1(F),\ \mathtt{JSON},\ \{\},\ \neg\left(\pi_2(F) \right) \right) \;\middle|\; F \in \pi_2(C) \right\} \\
② &:= \left\{ \left(\pi_1(F^{\mathrm{T}}),\ \pi_2(F^{\mathrm{T}}),\ \pi_3(F^{\mathrm{T}}),\ \pi_4(F^{\mathrm{T}}) \vee \neg\left(\pi_5(F^{\mathrm{T}}) \right) \right) \;\middle|\; F^{\mathrm{T}} \in \pi_3(C) \right\}
\end{aligned}
\tag{8.23}
$$

In case the mapping is done on the result of a relational view, the list of columns defined for every relational view is used as a projection list, reducing the result to a defined list of columns. Since the relational model is strictly typed, the values are converted into the type specified by the list of columns in the definition of the view. If a value cannot be converted to the specified data type, null is being used.

8.3.4 Document → LPG

Document schemas are mapped to LPG schemas by considering a document as a node. All fields of the document containing atomic values (no subdocuments) are represented as properties of the node. Subdocuments are represented as individual nodes without a label. The subdocument is linked by the parent document using an edge having the name of the corresponding field as label. Arrays are represented similarly, using the index as label. Substructures are treated accordingly, resulting in a tree structure that mimics the tree structures of a document. From the graph perspective, a document namespace is accessed using the name of a collection as a label.

$$\begin{gathered} \mathcal{N}^{\mathrm{DOC}} \mapsto \mathcal{N}^{\mathrm{LPG}} \\ \Big(\mathsf{name},\ \{C_1, ..., C_i\},\ \{...\}\Big) \mapsto \Big(\pi_1\big(\mathcal{N}^{\mathrm{DOC}}\big),\ \mathsf{null},\ \{\}\Big) \end{gathered} \tag{8.24}$$

The set of node labels contains the names of all collections in the namespace:

$$\left\{\pi_1(C) \;\middle|\; C \in \pi_2\big(\mathcal{N}^{\mathrm{DOC}}\big)\right\} \tag{8.25}$$

The set of edge labels contains the names of all non-atomic typed and untyped fields of all collections in the namespace:

$$\left\{\pi_1(F^*) \;\middle|\; F^* \in \big(\pi_2(C) \cup \pi_3(C)\big) \ \wedge\ C \in \pi_2\big(\mathcal{N}^{\mathrm{DOC}}\big)\right\} \tag{8.26}$$

8.3.5 LPG → Relational

In the relational context, an LPG namespace is accessed using a specific label of the graph as table name (e.g., `SELECT * FROM mygraph.employee`, with `mygraph` being the name of the namespace and `employee` the label). All nodes with this label are represented as records of this table. The table has three columns: An id column containing the unique id of the node, a properties column containing a map with the property key and value pairs and a labels column containing a list of all labels of this node.

$$\begin{aligned} \mathcal{N}^{\text{LPG}} &\mapsto \mathcal{N}^{\text{REL}} \\ \Big(\text{name},\ \mathcal{G},\ \{...\}\Big) &\mapsto \Big(\pi_1(\mathcal{N}^{\text{LPG}}),\ \texttt{ntab}(\mathcal{N}^{\text{LPG}}) \cup \texttt{etab}(\mathcal{N}^{\text{LPG}}),\ \{\}\Big) \end{aligned} \tag{8.27}$$

Since the mapping of the schema optional LPG graph to the relational schema does not depend on the optional graph schema but on the labels present in the graph at the time of mapping, we define a function $\texttt{labelsOf}(\cdot)$ that returns the set of node labels of a graph namespace. With this function, we can define the function $\texttt{ntab}(\cdot)$ that maps an LPG namespace $\mathcal{N}^{\text{LPG}}$ to a set of relational tables $\{\mathcal{T}_1, ..., \mathcal{T}_j\}$:

$$\texttt{ntab}(\mathcal{N}^{\text{LPG}}) := \Big\{(x,\ \text{NCOL},\ \{\}) \;\Big|\; x \in \texttt{labelsOf}(\mathcal{N}^{\text{LPG}})\Big\} \tag{8.28}$$

NCOL is identical for all node mappings:

$$\begin{aligned} \text{NCOL} := \Big(&(\text{id},\ \text{STRING},\ \{...\},\ \text{false}), \\ &(\text{properties},\ \text{STRING MAP},\ \{...\},\ \text{false}), \\ &(\text{labels},\ \text{STRING LIST},\ \{...\},\ \text{false})\Big) \end{aligned} \tag{8.29}$$

Edges are also represented using tables, mimicking a join table. However, they are not accessed directly by their label but mapped based on the nodes they connect, by using two node labels separated by an "->" as table name (e.g., `SELECT * FROM mygraph.employee->department`). This table contains all edges from nodes with the label "employee" to nodes with the label "department". The table contains a record for every edge between those two nodes in this direction. The table has four columns: The first column has the name of the outgoing node label (in our example this is "employee") and contains the unique ID of this node. The second column has the name of the target node label ("department" in our example) and contains the unique ID of the target node. The third column has the name "label" and contains the label of the edge, and the fourth column contains a map with key-value pairs of the properties of the edge.

More formally, the set of edge tables is obtained by considering all permutations of two node labels. Since there can also be edges referencing the same node, the $x = y$ case is deliberately included. The function maps an LPG namespace $\mathcal{N}^{\text{LPG}}$ to a set of relational tables $\{\mathcal{T}_1, ..., \mathcal{T}_j\}$.

$$\texttt{etab}(\mathcal{N}^{\text{LPG}}) := \Big\{(x\text{->}y,\ \text{ECOL},\ \{\}) \;\Big|\; x, y \in \texttt{labelsOf}(\mathcal{N}^{\text{LPG}})\Big\} \tag{8.30}$$

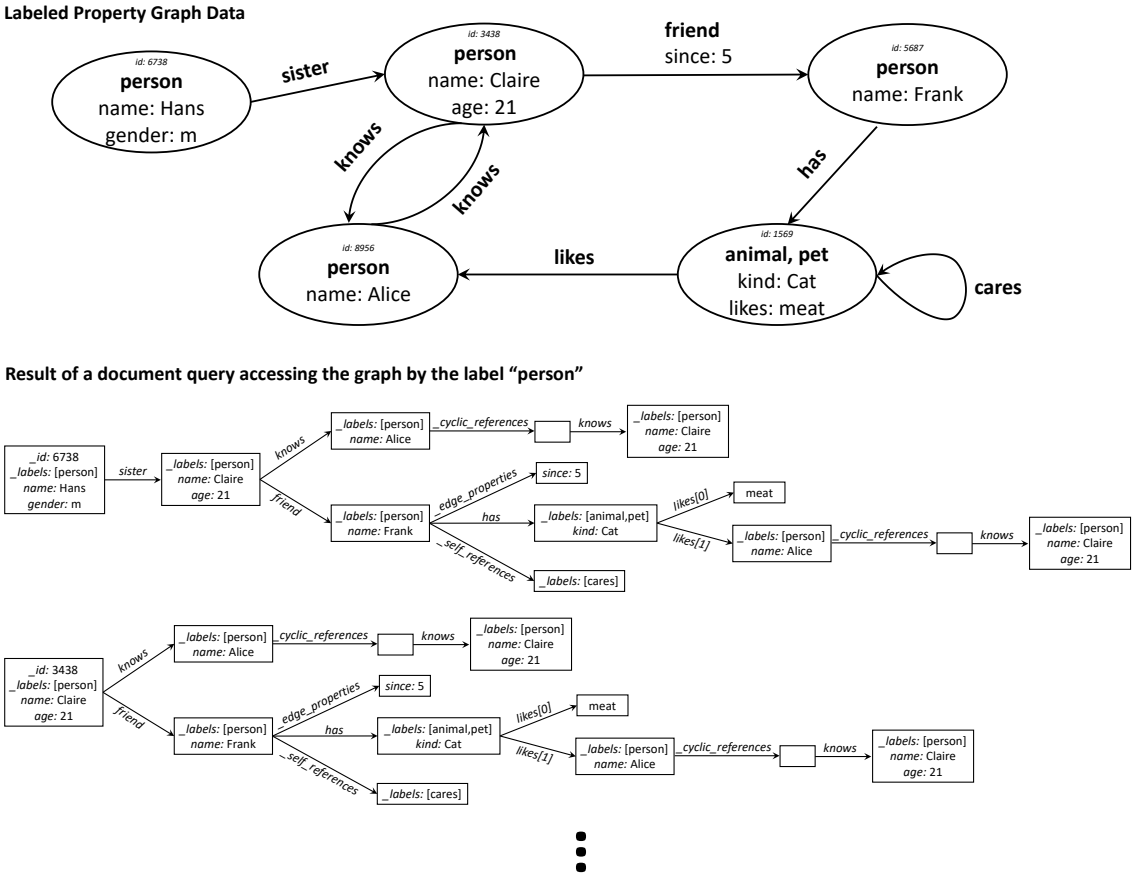


Figure 8.2 An LPG graph is logically mapped to a document by the label "person" (e.g., using the following MQL query: `db.person.find()`). For the depicted graph and query, the result would consist of four documents. In the figure, two of these documents are depicted as trees.

The set of columns of an edge table is defined as:

$$\begin{aligned} \text{ECOL} := \Big(& (\text{src},\ \text{STRING},\ \{...\},\ \text{false}), \\ & (\text{tgt},\ \text{STRING},\ \{...\},\ \text{false}), \\ & (\text{properties},\ \text{STRING MAP},\ \{...\},\ \text{false}), \\ & (\text{labels},\ \text{STRING LIST},\ \{...\},\ \text{false}) \Big) \end{aligned} \tag{8.31}$$

8.3.6 LPG → Document

From the document perspective, an LPG namespace is accessed using a specific label. This label is considered as collection with each node with this label represented as a document. All parts of the graph reachable from this node are added to the document as arbitrarily deeply nested structures. The label and properties of an edge are represented as a nested structure with the linked node. If an LPG node has a property with the same

name as an outgoing edge, or if there are multiple outgoing edges with the same label, arrays are being used.

Since LPGs may contain cyclic structures that cannot be represented in a tree, such cycles must be handled and labeled accordingly. This handling is done individually for every branch of the tree. If a node of the LPG graph has already been mapped as a parent node in the tree, a special marker field is inserted. This field contains the linked node and the properties of the edge. Self-references are handled similarly using a special field in the respective (sub)document. Figure 8.2 exemplifies this mapping.

More formally, the resulting schema of this mapping can be defined as follows:

$$\begin{aligned} \mathcal{N}^{\text{LPG}} &\mapsto \mathcal{N}^{\text{DOC}} \\ \Big(\text{name},\ \mathcal{G},\ \{...\}\Big) &\mapsto \Big(\pi_1(\mathcal{N}^{\text{LPG}}),\ \texttt{dcol}(\mathcal{N}^{\text{LPG}}),\ \{\}\Big) \end{aligned} \tag{8.32}$$

Since we are mapping schemas that are both schema optional, the no complex schema is required. The $\texttt{dcol}(\cdot)$ function thus only returns a set of document collections containing one collection for each node label in the graph. These collections only enforce an _id field:

$$\begin{aligned} \texttt{dcol}(\mathcal{N}^{\text{LPG}}) &:= \left\{ (x, \{\}, \{①\}) \;\middle|\; x \in \texttt{labelsOf}(\mathcal{N}^{\text{LPG}}) \right\} \\ ① &:= (\text{\_id},\ \text{UNIQUE ID},\ \{\},\ \text{false},\ \text{true}) \end{aligned} \tag{8.33}$$

8.4 Physical Mapping

The physical schema is the schema maintained on the underlying data stores. Due to the PolyDBMS Requirement 3.3, a model is required for mapping logical schemas of different data models to a physical schema for storing and processing the data. The physical schemas are hidden from the user (a consequence of Requirement 3.4). The mapping to a physical data store is different from the virtual mapping between the logical schema models introduced in Section 8.3. While the virtual mapping maps a namespace and makes its semantic concepts available using building blocks of a different data model, the physical mapping maps individual entities to the data model of an underlying data store. It is thus optimized for efficiently storing and processing data and avoids redundancies. Since even the capabilities of data stores based on the same data model vary significantly, the mappings introduced in this section should be seen as a vanilla approach. The actual mapping strategy should be adjusted further for every supported data store to fully exploits its capabilities.

In the following sections, we define strategies for mapping logical schemas based on a data model different from the native data model of the data store. The arrows in the headings of the following sections indicate the direction of the mapping. In the section "Relational → Document" for instance, it is outlined how a relational schema is physically stored in a database based on the document data model.

8.4.1 Relational → Document

Storing relational tables in a document database is straightforward. Every table is mapped to a collection on the document database. Every record of the table is represented as a document, storing the values for the columns of the table as fields of the document. The primary key of the table is used as value for the _id field.

$$\begin{gathered} \mathcal{T} \mapsto \mathcal{C} \\ \Big(\mathsf{name},\ (COL_1, ..., COL_i),\ \{...\}\Big) \mapsto \Big(\pi_1(\mathcal{T}),\ \{\},\ ①\Big) \\ ① := \Big\{\Big(\pi_1(COL),\ \pi_2(COL),\ \pi_3(COL),\ \pi_4(COL),\ \mathsf{true}\Big) \ \Big|\ COL \in \pi_2(\mathcal{T})\Big\} \end{gathered} \tag{8.34}$$

8.4.2 Relational → LPG

For storing relational tables in an LPG database, the table names are used as labels. Every record of a table is represented as a node with this label. The columns are represented as properties using the column name as the property key. Such graphs do not contain any edges.

$$\begin{gathered} \mathcal{T} \mapsto \mathcal{N}^{\text{LPG}} \\ \Big(\mathsf{name},\ (COL_1, ..., COL_i),\ \{...\}\Big) \mapsto \Big(\pi_1(\mathcal{T}),\ \mathsf{null},\ \{\}\Big) \end{gathered} \tag{8.35}$$

The set of node labels consists of the table name: $\{\pi_1(\mathcal{T})\}$

8.4.3 Document → Relational

A document schema is mapped to a relational data store by mapping a collection to a table. The table has at least two columns: An _id column storing the value of the primary key and a _data column containing the values for all fields and eventual nested structures encoded as JSON. If there are *defined fields* for the logical collection to be mapped, these fields are mapped to individual columns. Every document of the collection is stored as a record of the table. Since it can be specified whether a defined field is required on every

document, the table may contain null values for those columns where the required flag is not set.

$$
\begin{aligned}
&C \mapsto \mathcal{T} \\
&\Big(\text{name},\ \{F_1, ..., F_n\},\ \{F_1^{\mathrm{T}}, ..., F_m^{\mathrm{T}}\}\Big) \mapsto \Big(\pi_1(C),\ ① \cup ② \cup ③ \cup ④,\ \{\}\Big) \\
&① := \Big\{\big(\text{\_id},\ \text{UNIQUE ID},\ \{\},\ \text{true}\big)\Big\} \\
&② := \Big\{\big(\text{\_data},\ \text{JSON},\ \{\},\ \text{true}\big)\Big\} \\
&③ := \Big\{\Big(\pi_1(F),\ \text{JSON},\ \{\},\ \neg\big(\pi_2(F)\big)\Big) \ \Big|\ F \in \pi_2(C)\Big\} \\
&④ := \Big\{\Big(\pi_1\big(F^{\mathrm{T}}\big),\ \pi_2\big(F^{\mathrm{T}}\big),\ \pi_3\big(F^{\mathrm{T}}\big),\ \pi_4\big(F^{\mathrm{T}}\big) \vee \neg\big(\pi_5(F^{\mathrm{T}})\big)\Big) \ \Big|\ F^{\mathrm{T}} \in \pi_3(C)\Big\}
\end{aligned}
\tag{8.36}
$$

8.4.4 Document → LPG

Document schemas are physically mapped to LPG-based data stores similar to the virtual mapping outlined in Section 8.3.4, mimicking the tree structure of a document using nodes and edges. This approach enables push down of operations, significantly reducing the amount of data that needs to be streamed and processed by the PolyDBMS.

$$
\begin{aligned}
&C \mapsto \mathcal{N}^{\mathrm{LPG}} \\
&\Big(\text{name},\ \{F_1, ..., F_n\},\ \{F_1^{\mathrm{T}}, ..., F_m^{\mathrm{T}}\}\Big) \mapsto \Big(\pi_1(C),\ \text{null},\ \{\}\Big)
\end{aligned}
\tag{8.37}
$$

The set of node labels contains only the name of the collection to be mapped:

$$\Big\{\pi_1(C)\Big\} \tag{8.38}$$

The set of edge labels contains the names of all non-atomic typed and untyped fields of the collection:

$$\Big\{\pi_1(F^*) \ \Big|\ F^* \in \big(\pi_2(C) \cup \pi_3(C)\big)\Big\} \tag{8.39}$$

8.4.5 LPG → Relational

LPG graphs are stored in a relational database using four tables: node, edge, nodeProperty, and edgeProperty. The node table has two columns: one for storing the unique ID of a node and one for storing a label. Since a node can have multiple labels, there can be multiple entries for an LPG node in this node table. Similarly, the edge table stores all edges. It has four columns: the unique ID of the edge, the label of the edge, the unique

ID of the source node, and the unique ID of the target node. Since an edge in the LPG model can only have one label, there is only one entry per edge.

The two property tables store the properties of the edges and the nodes, respectively. Both tables have the same set of columns: a column for the unique ID of the corresponding node or edge, a column for the property key, and a column for the property value. There is an entry for each property of every node or edge.

While this approach requires joining the tables, it does not require any serialization or deserialization. This allows to push down parts of the query. Furthermore, the ability to filter for certain labels or properties massively reduces the amount of data that needs to be streamed to the PolyDBMS and handled in its integrated engine.

$$
\begin{aligned}
\mathcal{N}^{\text{LPG}} &\mapsto \{\mathcal{T}_1, ..., \mathcal{T}_4\} \\
\big(\text{name},\ \mathcal{G},\ \{...\}\big) &\mapsto \{\text{NODE\_TAB},\ \text{EDGE\_TAB},\ \text{NPROP\_TAB},\ \text{EPROP\_TAB}\} \\
\text{NODE\_TAB} &:= \big(\text{node},\ \{①, ②\},\ \{\}\big) \\
\text{EDGE\_TAB} &:= \big(\text{edge},\ \{①, ②, ③, ④\},\ \{\}\big) \\
\text{NPROP\_TAB} &:= \big(\text{edge},\ \{①, ⑤, ⑥\},\ \{\}\big) \\
\text{EPROP\_TAB} &:= \big(\text{edge},\ \{①, ⑤, ⑥\},\ \{\}\big) \\
① &:= (\text{id},\ \text{STRING},\ \{...\},\ \text{false}) \\
② &:= (\text{label},\ \text{STRING},\ \{...\},\ \text{false}) \\
③ &:= (\text{source},\ \text{STRING},\ \{...\},\ \text{false}) \\
④ &:= (\text{target},\ \text{STRING},\ \{...\},\ \text{false}) \\
⑤ &:= (\text{key},\ \text{STRING},\ \{...\},\ \text{false}) \\
⑥ &:= (\text{value},\ \text{STRING},\ \{...\},\ \text{true})
\end{aligned}
\tag{8.40}
$$

8.4.6 LPG → Document

The mapping of LPG graphs to document databases is similar to the approach used to map LPG graphs to relational databases outlined above. However, instead of four tables, the mapping is done using two collections: node and edge. Nodes are represented as documents using the unique ID of the node as value for the _id field of the document. The properties of the node are stored as additional fields of the document. The labels of the node are stored as a list. Edges are represented similarly; however, since edges only have one label, the label of an edge can be stored using a single field in the document.

$$
\begin{aligned}
\mathcal{N}^{\text{LPG}} &\mapsto \{C_1, C_2\} \\
\big(\text{name},\ \mathcal{G},\ \{...\}\big) &\mapsto \{\texttt{NODE\_COLLECTION},\ \texttt{EDGE\_COLLECTION}\} \\
\texttt{NODE\_COLLECTION} &:= \big(\text{node},\ \{\},\ \{①, ②\}\big) \\
\texttt{EDGE\_COLLECTION} &:= \big(\text{edge},\ \{\},\ \{①, ③\}\big) \\
① &:= (\text{\_id},\ \text{UNIQUE ID},\ \{\},\ \text{true}) \\
② &:= (\text{\_labels},\ \text{STRINGLIST},\ \{\},\ \text{true}) \\
③ &:= (\text{\_label},\ \text{STRING},\ \{\},\ \text{true})
\end{aligned}
\tag{8.41}
$$

8.5 Allocation of Data

After we have introduced the logical schema model (Section 8.2) and how logical schemas are mapped to physical schemas (Section 8.4), this section introduces a model for putting both together. Instead of simply allocating a namespace to one underlying data store, our model allows to replicate and partition data across multiple underlying data stores. Furthermore, it provides the capabilities to adjust the allocation depending on the access frequency and allows to delay the update on some of the replicas. This enormous flexibility clearly sets this model apart from existing polystore and multimodel database systems.

8.5.1 Replication and Partitioning

As introduced in Section 6.1, we can distinguish between *data replication* and *data partitioning*. For the latter, it can furthermore be distinguished between *vertical partitioning* and *horizontal partitioning*. While these two terms have their origin in the relational data model, we decided to use them for this schema model as well. Similar to our definition in Section 6.1, we use *horizontal* to refer to the partitioning that only affects the data and keeps the schema identical on all partitions. *Vertical* partitioning in contrast affects both the schema and the data. While support for replication or partitioning of data is not required by the definition of a PolyDBMS introduced in Chapter 3, we consider it crucial for providing high query performance.

In a PolyDBMS with multiple, heterogeneous underlying data stores, replicating data on multiple data stores allows optimizing for different kinds of queries. Vertical partitioning allows to reduce the storage overhead by only storing certain parts of a schema object (e.g., certain columns of a table) on specific data stores. With horizontal partitioning, it

is possible to implement load-balancing between multiple data stores or—in combination with the concept introduced in Section 8.5.3—to optimize the performance for frequently used data.

Our schema model supports arbitrary combinations of horizontal and vertical data partitioning and data replication. The only constraint is, that there is the same set of schema object parts (e.g., columns of a table) for all partitions of an entity (e.g, a table) assigned to a particular data store.

A *data entity* is the structure holding the data. In namespaces of the type relational or document, the data is held by tables ($\mathcal{T}$) or collections (C). Furthermore, data is also represented by views ($\mathcal{V}$). If the result represented by a view is physically stored, the view is called a *materialized view*. In namespaces of type LPG, the data is held by the namespace ($\mathcal{N}^{\text{LPG}}$) itself. A data entity E is therefore defined as:

$$E \in \left\{\mathcal{T}, C, \mathcal{N}^{\text{LPG}}, \mathcal{V}^{\text{REL}}, \mathcal{V}^{\text{DOC}}, \mathcal{V}^{\text{LPG}}\right\} \tag{8.42}$$

Every data entity E has a horizontal partition function ϱ. A partition function maps a value v to a partition P:

$$\varrho : v \mapsto P \tag{8.43}$$

In our model, all data entities have a partition function assigned, making all entities partitioned. However, this partition function can be the *NONE* partition function, assigning all data to one partition. Besides the *NONE* partition function available for every data model, the set of partition functions depends on the data model of the namespace. For relational namespaces, we define a *HASH*, *LIST* and *RANGE* partition function. All three partition functions are applied on a partition column $COL^P \in \mathcal{T}$. The *RANGE* partition function needs to be applied to a column with a numerical data type while the *LIST* function requires a string or integer type. The *HASH* partition function can be applied to an arbitrary data type.

$$\begin{aligned}
\varrho^{\text{HASH}} &: \left(\mathcal{T}, COL^P\right) \mapsto (P_1, ..., P_n) \\
\varrho^{\text{LIST}} &: \left(\mathcal{T}, COL^P, \left(\{v_1, ..., v_i\}_1, ..., \{v_1, ..., v_j\}_n\right)\right) \mapsto (P_1, ..., P_n) \\
\varrho^{\text{RANGE}} &: \left(\mathcal{T}, COL^P, \left((v_{\text{min}}, v_{\text{max}})_1, ..., (v_{\text{min}}, v_{\text{max}})_n\right)\right) \mapsto (P_1, ..., P_n)
\end{aligned} \tag{8.44}$$

Since the list of values and the list of min and max pairs specified for the list and range partition function do not need to be closed (i.e., there might be values that do not appear in the sets or intervals), both partition functions define an *unbound* partition for all those records. However, the specified mapping must be unambiguous.

For the document model, a *HASH* partition function is defined similarly to the relational model. However, this function always operates on the _id field of the documents. For LPGs, horizontal partitioning is more complex. Graph partitioning—which is the theory of reducing a graph into mutually exclusive sets of nodes—is typically considered an NP-hard problem [OW05]. We, therefore, decided to define the horizontal partitioning of LPGs based on the node and edge labels. With ℓ being a node or edge label, the partition function is defined as:

$$\varrho^{\text{LABELS}} : \left(\mathcal{N}^{\text{LPG}}, \left(\{\ell_1, ..., \ell_i\}_1, ..., \{\ell_1, ..., \ell_j\}_n\right)\right) \mapsto (P_1, ..., P_n) \tag{8.45}$$

The set of node labels must not overlap. If a graph is partitioned, the PolyDBMS does not allow edges between nodes residing in different partitions. Further, it does not allow assigning labels belonging to different partitions to the same node. An edge connecting two nodes must have a label belonging to the set of labels of the corresponding partition. Hence, the partitioning enforces unconnected subgraphs.

A vertical partition entity B is the substructure of a data entity that is subject to vertical partitioning. In the relational model, these are the columns (*COL*) of a table and in the document model the typed (F^{T}) and untyped (F) fields of a document. In the LPG model, vertical partitioning is done on the properties of a node or edge (p).

$$B \in \left\{COL, F, F^{\text{T}}, p\right\} \tag{8.46}$$

With ℓ being a label and P_E being a partition of the data entity E, a partition group G of an entity E is defined as:

$$G_E := \left(\ell, \{P_E^1, ..., P_E^j\}\right) \tag{8.47}$$

A data placement L describes the mapping of data to underlying data stores. The mapping of an entity E to a data store A is described by a pair

$$L_E^A := \left(\left\{(G_E^1, \Upsilon_1), ..., (G_E^n, \Upsilon_n)\right\}, \left\{B_E^1, ..., B_E^m\right\}\right) \tag{8.48}$$

with $\Upsilon \in \{\text{EAGER}, \text{LAZY}\}$ describing the update strategy of the data placement (see Section 8.5.2). With a function $E(L)$ that returns the data entity of a data placement, an underlying store can be described by a set of data placements:

$$A := \{L_1, ..., L_n\} \quad \forall\, L_i, L_j \in A : E(L_i) \neq E(L_j) \tag{8.49}$$

Since the set of partition groups and the set of vertical partition entities are independently defined for the entire data placement L_E^A, and since there can only be one data placement for an entity assigned to a specific data store, the constraint that all partitions placed on a data store must have the same set of vertical partition entities is fulfilled.

8.5.2 Data Freshness

In Section 6.4, we have introduced the concept of data freshness in the context of distributed database systems. Depending on the strategy with which updates are propagated in the distributed system and depending on the distribution protocol, individual sites can have outdated versions of certain elements. It depends on the use case and often also on the type of query, whether a certain degree of outdatedness is an issue or not [ASS08].

In the context of a PolyDBMS, this is interesting since it allows to increase the performance of OLTP workloads by applying data manipulation operations only to a subset of the stores immediately; on the other data stores, the operations are deferred. The update strategy $\Upsilon \in \{\text{EAGER}, \text{LAZY}\}$ is defined individually for every partition group placed on a data store. *EAGER* means that DML operations are applied immediately while *LAZY* allows data manipulation operations to be executed later, resulting in outdated data.

A read-only query can specify a required level of data freshness. This model defines data freshness based on the time elapsed since the partition has been refreshed the last time. If there are no deltas to be applied, the partition is always considered up-to-date. With t_s being the timestamp of the last DML operation applied to the data store, t_{now} being the current timestamp, d the accepted outdatedness, and n the number of pending DML operations, the freshness function is defined as:

$$\texttt{freshness}(t_{\text{store}}, t_{\text{now}}, d, n) := \begin{cases} \text{true} & \text{if } n = 0 \\ (t_{\text{now}} - t_{\text{store}}) \leq d & \text{else} \end{cases} \tag{8.50}$$

The function returns true iff the freshness requirements of the query can be fulfilled. It is applied for every partition accessed by the query.

8.5.3 Temperature-aware Data Management

As outlined in Section 6.2, temperature-aware data management refers to the concept of storing frequently used data in faster (but usually also more expensive) data stores while old and only rarely accessed data is stored on slower (but also cheaper) storage. In the context of a PolyDBMS with its heterogeneous data stores, this is even more interesting. Besides replicating frequently used data to stores running on faster storage (e.g., in-memory), data can also be replicated to a larger set of heterogeneous data stores. This allows to optimize frequently used data for a large spectrum of workloads without huge storage costs.

The partition group G introduced above allows grouping multiple partitions and allows assigning them a label. Since the mapping of partitions to partition groups can be changed, temperature-aware data management can easily be implemented in the above model using two partition groups, one with the label "hot" and one with the label "cold". A partition group assignment function $\varkappa$ assign a partition P to partition group G:

$$\varkappa : P \mapsto \left\{G^{\text{HOT}}, G^{\text{COLD}}\right\} \tag{8.51}$$

The assignment is done based on the access frequency and is updated at a certain interval. To avoid partitions oscillating between hot and cold, a hysteresis is specified. HT is the *hot threshold* and CT is the *cold threshold*. It holds: $HT > CT$. With $f(P)$ being a function that returns the access frequency of a partition and $g(P)$ returning the current partition group of a partition P, the assignment function $\varkappa$ is defined as:

$$\varkappa(P) := \begin{cases} G^{\text{HOT}} & \text{if } f(P) > \text{HT} \\ G^{\text{COLD}} & \text{if } f(P) < \text{CT} \\ g(P) & \text{else} \end{cases} \tag{8.52}$$

Instead of assigning the hot partition group only to one or multiple high-performance data stores, it can additionally also be assigned to the data stores holding the cold partition group. This approach is especially beneficial if there are analytical queries since it eliminates the need to union the data from the stores holding the hot and the cold data together. However, this impacts the performance of data modification queries on the "hot" data since the data also needs to be updated on the slower data store holding all data.

However, this can be compensated by combining the temperature-aware data management with the previously introduced data freshness. Since the update strategy Υ can be set individually for every partition group assigned to a data placement, the hot partition group can be allocated with an eager update strategy to the high-performance data stores and with a lazy update strategy to the cold stores. This results in high performance for transactional workload including data manipulation queries on the current data while still being able to execute analytical queries—for which slightly outdated data is acceptable—on the cold data store. If the latest data is required, the query can always be executed on a different data store or the queued delta operations can be applied prior to the query.

Trees sprout up just about everywhere in computer science.

— Donald E. Knuth

9

Query Representation

A PolyDBMS is inherently a multimodel system that must deal with multiple data models at different stages in the query processing and execution process. According to the requirements introduced in Section 3.1, a PolyDBMS must accept queries expressed in query languages based on different data models, internally support multiple data models, and maintain storage and execution engines based on different data models.

A challenge in such a system is the representation of a query in a form that can be processed and enables cross-model queries. The combinatorial complexity of multiple query languages, data models, and data stores requires a single form of representation. The operators are defined by the data model. Since the approach to represent a query needs to preserve the semantics of the data model the query language is based on, mapping every query to a single data model (for instance the relational data model) is not feasible.

Figure 9.1 depicts the various data models that can be found in a PolyDBMS. According to Requirement 3.2, every PolyDBMS needs to support multiple query languages. In the figure, these are depicted as *Input Query Language (IQL)*. All IQLs are translated into a representation based on one of the algebras defined by the native data models of the PolyDBMS. "Native" thereby refers to data models supported by the PolyDBMS to define the logical schema.

In the processing stage, the single algebras are represented using an algebra that is constructed as a superset of the data model specific algebras. This algebra is called *logical algebra*. Besides the model specific operators, it also contains a set of special operators for handling DML queries and modeling the enforcement of constraints. Furthermore, the views introduced in Chapter 8 are expanded in this stage.

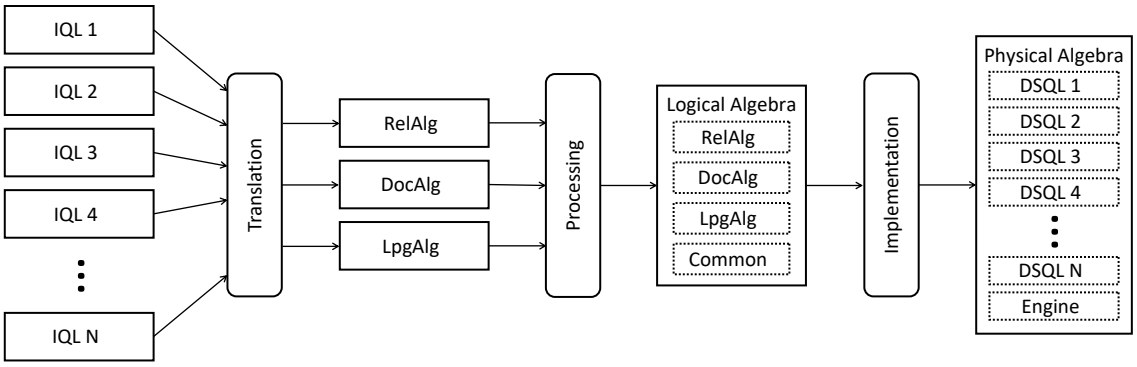


Figure 9.1 The sets of operators (i.e., algebras) used to express queries through various stages. *IQLs* are the input query languages, the query languages supported by the PolyDBMS (e.g., SQL). *DSQLs* are the data models of the query languages of the underlying data stores. *Engine* is the set of operators of the integrated execution engine of the PolyDBMS (see hybrid architecture model; Section 3.2.3).

In the final *implementation* step, the logical algebra is converted into a representation based on the *physical algebra*. This algebra is based on the operators of the query languages of the underlying data stores and the operators of the integrated engine of the PolyDBMS (see architectural model in Section 3.2.3).

The query representation model introduced in this chapter allows representing queries based on different data models. For this, we have chosen an approach similar to that introduced for the schema model in Chapter 8. The *PolyAlgebra* introduced in this chapter is constructed as a superset of the algebras of the data models supported by the PolyDBMS. Together with this PolyAlgebra, this chapter also introduces how the enforcement of constraints and the consistent handling of data modification queries can be expressed using the proposed form of query representation.

9.1 The PolyAlgebra

A fundamental idea of the PolyAlgebra concept is to consider not only the logical data models supported by the PolyDBMS but also the query languages of the underlying data stores and the integrated engine as distinct set of operations. This allows operators from different data models to be added to the same query plan and also enables a partial transition from a logical representation to a physical representation.

The PolyAlgebra $\mathbb{A}$ is a set of operators $\mathbb{A} = \{o_1, ..., o_N\}$. In Chapter 4, symbols for some of these operators were defined. This would allow queries based on these operators to be formally specified in an algebraic notation. However, for the sake of clarity, we have

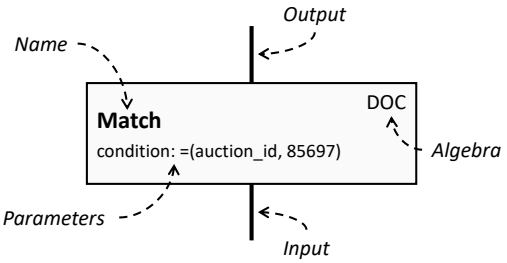


Figure 9.2 Graphical representation of an operator. The name of the operator is depicted bold and the algebra to which it belongs is specified in the upper-right corner. The box furthermore contains the values for the parameters of the operator.

decided against such a notation in most cases. Instead, the examples in this chapter are represented using a graphical representation in which operators are depicted as boxes. Figure 9.2 depicts the structure of such an operator box.

The PolyAlgebra is a superset of the *logical algebra* $\mathbb{A}^{\mathrm{L}}$ and the *physical algebra* $\mathbb{A}^{\mathrm{P}}$:

$$\mathbb{A} := \mathbb{A}^{\mathrm{L}} \cup \mathbb{A}^{\mathrm{P}} \tag{9.1}$$

The logical algebra is itself a superset of the algebras of the data models supported by the PolyDBMS plus a set of common operators. In this chapter, we exemplify the PolyAlgebra for the relational, document, and LPG data models. However, the presented concepts can also be applied to a different set of data models. The logical algebra $\mathbb{A}^{\mathrm{L}}$ is defined as:

$$\mathbb{A}^{\mathrm{L}} := \mathbb{A}^{\mathrm{REL}} \cup \mathbb{A}^{\mathrm{DOC}} \cup \mathbb{A}^{\mathrm{LPG}} \cup \mathbb{A}^{\mathrm{COMMON}} \tag{9.2}$$

The operators constituting the sets $\mathbb{A}^{\mathrm{REL}}$, $\mathbb{A}^{\mathrm{DOC}}$, and $\mathbb{A}^{\mathrm{LPG}}$ have been introduced in Chapter 4. The set of common operators $\mathbb{A}^{\mathrm{COMMON}}$ contains special operators such as operators for converting between data models or for modeling the enforcement of constraints. These operators will be introduced in more detail in the course of this chapter.

The selected architecture model for a PolyDBMS makes use of multiple heterogeneous execution engines to process queries. Primarily, it can be distinguished between the integrated execution engine of the PolyDBMS and the underlying data stores. Determining the optimal strategy for executing queries (i.e., decomposing the query and selecting where the subqueries should be executed), is called *query routing* and will be discussed in Chapter 10.

As mentioned before, every underlying data store provides its own set of operators. Together with the operators of the integrated execution engine, these operators form the

physical algebra; see Figure 9.1. While the logical algebra provides a logical representation of a query, the physical algebra describes where and how a query is executed. It is an intermediate representation which is used by the PolyDBMS to implement the query on the underlying data stores using their native query method (i.e., query language). Furthermore, it contains the instructions for the integrated execution engine of the PolyDBMS. While the logical algebra refers to the entities defined in the logical schema, the physical algebra refers to the entities defined in the physical schema.

Database systems typically do not expose their internal query representation, but expect queries to be expressed in a query language (e.g., SQL). The PolyDBMS must therefore interact with its underlying data stores using these query languages. The operators that comprise the physical algebra are therefore defined based on the structure and operators of these query languages. A convenient side effect is that data stores supporting the same query language can use the same or a similar set of operators.

More formally, the physical algebra is a superset of the algebra of the integrated engine of the PolyDBMS $\mathbb{A}^{\text{ENG}}$ and the algebras of the available data stores $\left\{\mathbb{A}^{\text{DS1}}, ..., \mathbb{A}^{\text{DS}n}\right\}$ where $n \in \mathbb{N}$ is the number of data stores available to the PolyDBMS. With this, $\mathbb{A}^{\text{P}}$ can be defined as:

$$\mathbb{A}^{\text{P}} := \mathbb{A}^{\text{ENG}} \cup \left(\bigcup_{x \in \mathbb{N}, x \leq n} \mathbb{A}^{\text{DS}x} \right) \tag{9.3}$$

9.1.1 Preserving Semantics

With the PolyAlgebra, we introduce an approach to represent queries originating from different query languages and based on different data models using a single algebra. This algebra consists of the algebras defined by the data models supported by the PolyDBMS and thus contains multiple operators for similar concepts originating from different data models. This is different from the typical approach of defining a single algebra based on a data model that incorporates features from all supported data models; which we will refer to as *unified approach*. In a PolyDBMS, this requires similar operators from different data models to be merged into one operator that encompasses the semantics of the data model specific operators. However, we see three strong arguments against the unified approach and in favor of our PolyAlgebra approach:

Firstly, the *preservation of the data model specific semantics of the operators*: A unified approach comes with the risk of implying a specific data model, since the dependencies

and the meaning of a certain combination of operators can be different between data models. Ensuring the semantics are preserved through all stages of query processing and optimization is challenging.

Secondly, the *mapping from and to the algebra*: If a query language is based on a data model similar to one of the data models supported by the PolyDBMS, the operators of the query language can be mapped to the operators defined by the data model of the PolyDBMS without a sophisticated logic. With a unified approach, the mapping is much more complex since the operators themselves are much more complex in order to encompass the different semantics. Furthermore, this complex mapping needs to be done individually for each supported query language and data store, duplicating and spreading logic across the PolyDBMS which is also suboptimal from an implementation perspective.

Thirdly, the *expandability*: This is much easier in the PolyAlgebra approach than in the unified approach. If the PolyDBMS is extended to support an additional data model, an additional set of operators can be added to the PolyAlgebra without any modifications to the operators of other data models. In the unified approach, such an extension might require complex changes to the operators to encompass the additional semantics and behavior. These changes then in turn require changes to all adapters and query language translators.

However, including the full set of operators from all supported data models in the PolyAlgebra has one main disadvantage: it results in a larger set of operators that need to be handled. However, from an implementational point of view, this can also be seen as an advantage since less complex operators are easier to implement and maintain.

9.1.2 The Scan Operator

For the algebras composing the PolyAlgebra, we refer to the operators introduced in Chapter 4. However, we have not introduced an approach for specifying which data the operators should be applied on. In Section 8.5, we have defined a data entity E as the set of schema objects that represent the individual entities holding the data:

$$E \in \{\mathcal{T}, C, \mathcal{N}^{\text{LPG}}, \mathcal{V}^{\text{REL}}, \mathcal{V}^{\text{DOC}}, \mathcal{V}^{\text{LPG}}\} \tag{9.4}$$

In namespaces of the type relational or document, the data is held by tables $\mathcal{T}$ or collections C. In namespaces of type LPG, the data is held by the namespace itself $\mathcal{N}^{\text{LPG}}$. Furthermore, data is also represented by views $\mathcal{V}$.

Since databases usually consist of multiple such data entities, a special operator is required to select the data entity holding the data the operators should be applied on. This operator is called **scan** operator. There is one such operator for every supported data model, taking a schema object E as parameter and returning the data as specified by the corresponding data model.

The LPG algebra has two result representations: *graph* and *graph relation*. This is required since the **match** operator needs to operate on the whole graph in order to match paths and build a graph relation. The LPG node operator introduced in Section 4.3 is thereby subsumed in the **match** operator. It therefore takes a graph as input and returns a graph relation as output. The **union** operator is defined for both graph and graph relation. All other LPG operators solely operate on graph relations.

Besides the **scan** operator there is also the **value** operator. This operator can be seen as a special form of the **scan** operator, which instead of a data entity directly specifies the data. This for instance allows to model *insert* operations; see Section 9.3.

9.1.3 Query Plans

As outlined in Chapter 4, queries can be represented as operator trees. We refer to these trees as *query plans*. With an operator $o \in \mathbb{A}$ and a list of parameters $(v_1, .., v_m)$, a query plan q can be represented by a tuple:

$$q := \left(o, (v_1, .., v_m), (q_1, ..., q_n)\right) \text{ with } n, m \in \mathbb{N} \tag{9.5}$$

We refer to a query plan as *logical query plan* if all operators belong to the logical algebra $\mathbb{A}^{\mathrm{L}}$ and as *physical query plan* if all operators belong to the physical algebra $\mathbb{A}^{\mathrm{P}}$. The process of converting a logical query plan into a physical query plan and optimizing it for an efficient execution is called *query planning* and *query optimization*. This will be discussed in Chapter 10 in the context of *query routing*.

Figure 9.3 depicts a logical query plan. Every valid tree must either have a **value** or a **scan** operator at each of its leaves. The query plan depicted in this example joins two relational tables and filters by a specific attribute. The plan is not optimized and solely consists of logical operators.

Since this representation allows query plans to contain arbitrary combinations of operators from different data models (i.e., from different data models), a mechanism is required to convert the output of an operator to fit the expected input of its parent operator. In order to model this mapping as part of the query plan, we introduce **converter** operators.

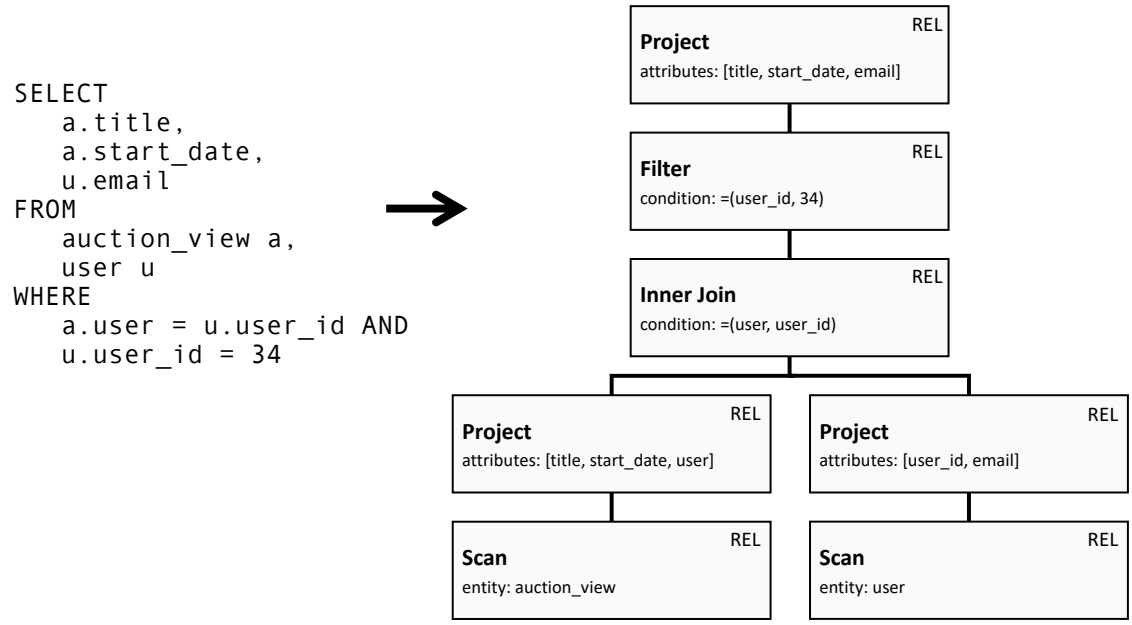


Figure 9.3 An example of a query plan consisting of relational operators. It represents the SQL query depicted on the left side.

These operators take a result represented according to one data model and convert it into another data model. The conversion is done according to the definitions for mapping the schema provided in Section 8.4.

9.2 Push Down

Every data store needs to provide support for a **scan** operator that reads and streams the whole data of a specified data entity. For modifying data, each data store furthermore needs to provide support for a **modify** operator, a **value** operator, a basic **filter** operator, and a basic **project** operator. Basic in this case means, that only a small subset of the functionality is required. The **filter** operator needs to support an equality comparison of a field or attribute with a provided value. The **project** operator needs at least support removing attributes or paths and adding attributes or paths with a specified value. These operators are required to implement DML queries as described in Section 9.3.

To meet the independence of storage configuration requirement demanded by the PolyDBMS Requirement 3.3, the integrated engine of the PolyDBMS must provide full support for all operators supported by the PolyDBMS except for the **scan** and the **modify** operator. Since the PolyDBMS relies on its underlying data stores for persisting data, support for these two operators is only mandatory on the underlying data stores. The integrated en-

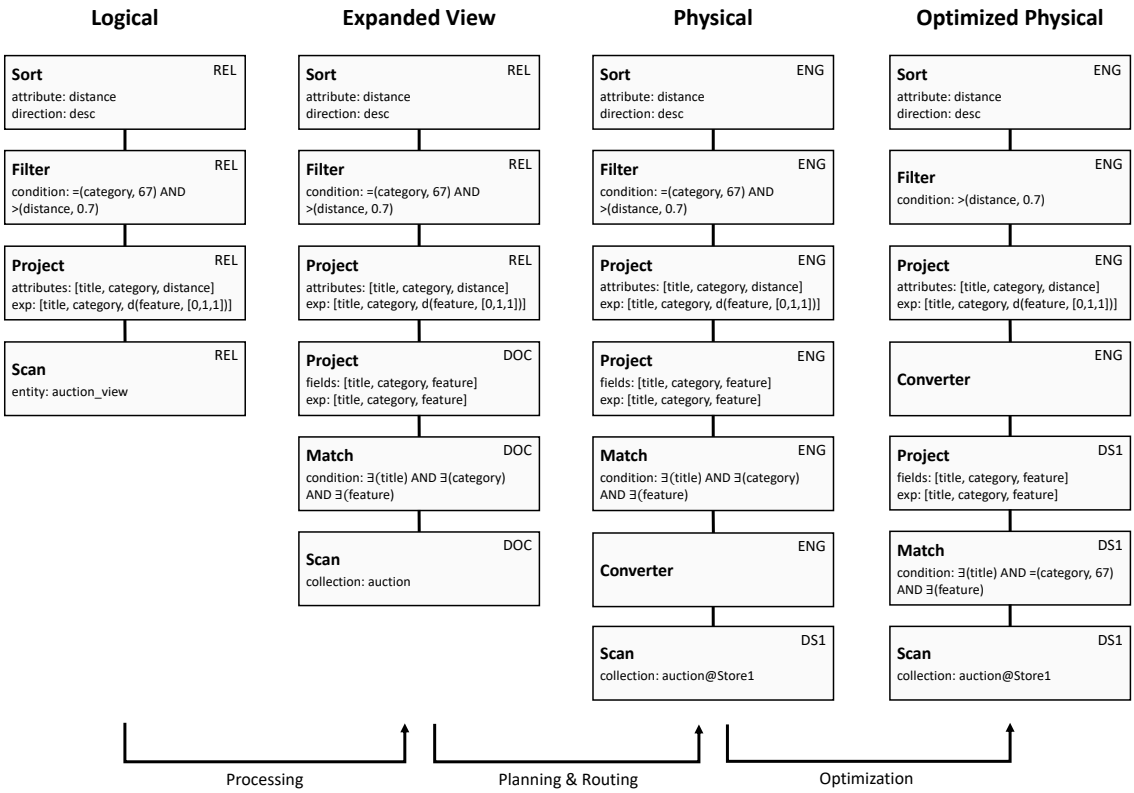


Figure 9.4 Query plans of the same cross-model query at different stages.

gine of the PolyDBMS operates on data streamed from one or multiple of the underlying data stores.

Since reading a data entity from an underlying data store and streaming the whole data into the integrated execution engine of the PolyDBMS is expected to be less efficient than executing the query on the underlying data store, the PolyDBMS makes use of additional operators provided by the data stores. This avoids IO heavy data streaming and makes use of the optimizations of the data store (see Section 3.2.3). This execution of an operator on an underlying data store is referred to as *push down*.

Figure 9.4 depicts the various representations of a query that retrieves all auctions belonging to the category with the ID 67 and having at least a certain similarity to a specified vector. This similarity is calculated using the distance function d. This function returns the Euclidean distance between two vectors. The depicted logical plan on the left side is the result of a query based on the relational model (e.g., SQL). Since auction is a collection of documents, there are two approaches for querying it from a relational context (see Section 8.3.3): directly or by means of a view. In this example, the query operates on a view that has been defined using a query language based on the document data model. Thus, the query plan obtained when expanding the view contains operators from the relation algebra and from the document algebra.

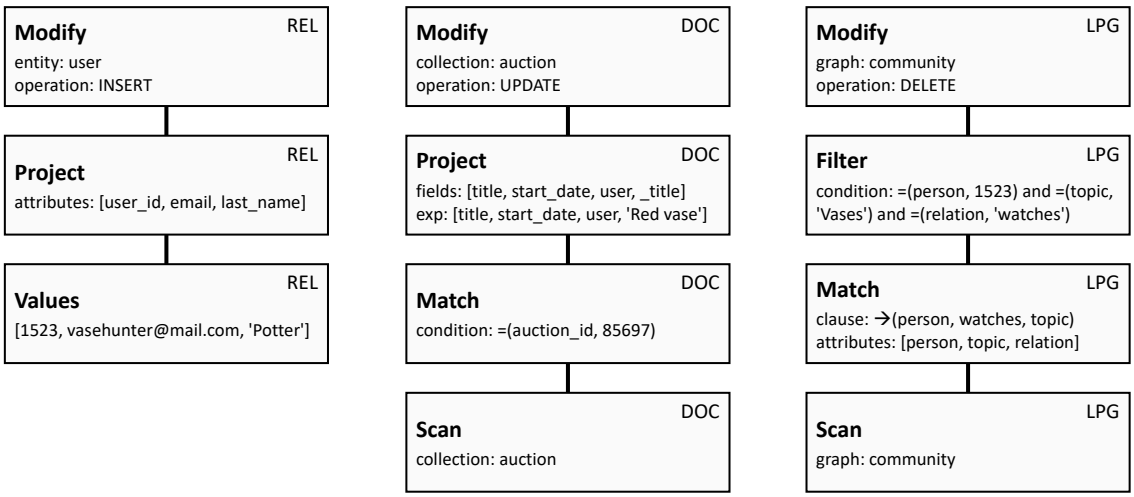


Figure 9.5 Structure of different data manipulation operations represented as query plans using the PolyAlgebra. The left plan shows a relational insert query, the plan in the middle an update of a collection in a document model, and the one on the right depicts the deletion of an edge in an LPG graph.

To execute this query plan, it needs to be implemented using operators of both the underlying data stores and the integrated execution engine of the PolyDBMS. In this example, we assume that the auction collection is neither replicated nor partitioned. Hence, there is only one data store holding the data. In Figure 9.4 the operators of this data store are labeled as *DS1* while the operators of the integrated execution engine are labeled as *ENG*.

The third query plan (physical) depicted in the figure is not optimized at all. The whole data of the collection is read from the data store and converted to the internal representation of the PolyDBMS. All operators are implemented using the integrated execution engine. To improve the performance, in the fourth plan, some of the operators are implemented using operators of the underlying data store (i.e., they are *pushed down*). Since the underlying data store in this example does not support the distance function d, some parts of the query still need to be executed using the integrated engine. However, the amount of data that needs to be streamed to and processed by the integrated engine of the PolyDBMS got significantly reduced due to the push down of operators to the underlying data store.

9.3 Representation of DML Queries

The algebras introduced in Chapter 4 only specify operators for querying the data, but not for manipulating it. We therefore introduce the **modify** operator. This operator allows

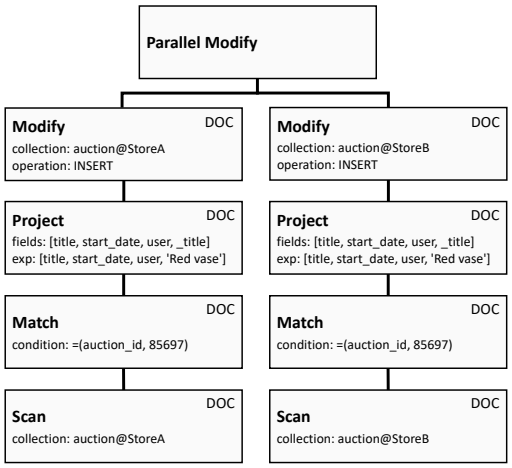


Figure 9.6 An exemplary query plan representing the execution of an insert on two data stores.

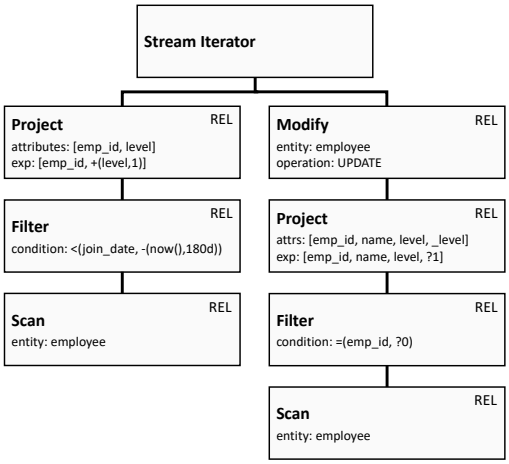


Figure 9.7 A complex update statement on a data store that does not supported the required operations.

modifying the data held by a data entity, according to the specified operation. The **modify** operator is a unary operator, taking the records to be dealt with as input and returning an integer indicating the number of inserted, updated or deleted data items as output.

For insert operations, the input defines the data to be added to the specified data entity while for delete operations, it defines the records to be deleted. For update operations, the input to the **modify** operator specifies both the current values of the records to be updated and the new values. Figure 9.5 depicts the structure of different data modification operations. The query plan beneath the **modify** operator can be arbitrarily complex. This, for example, allows copying data from other data entities or model complex conditions.

Since the schema model introduced in Chapter 8 allows data to be replicated across multiple underlying data stores, data modifications need to be performed on all data stores in order to keep the data consistent. To express this in one query plan, we introduce the **parallel modify** operator. This operator executes data modification queries simultaneously on multiple data stores. This is depicted in Figure 9.6.

More formally, the **parallel modify** operator is applied to a list of query plans as follows:

$$(q_1, ..., q_n) \mapsto \left(\textbf{parallel modify}, (), (q_1, ..., q_n)\right) \text{ with } n \in \mathbb{N}, n \geq 2 \tag{9.6}$$

Another aspect that needs to be addressed using a special operator is the consistent execution of arbitrary complex DML queries on any underlying data store. Since a data store only needs to provide support for a basic set of operators, there can be issues with DML queries containing complex conditions.

For example, consider an update query that increases the "level" of every employee working for the company since at least 180 days. This query requires the data store to subtract and compare timestamps and increment an integer. This would require operators that are not part of the minimal set of operators to be supported on every underlying data store.

Theoretically, executing such queries on a data store without support for all operators in the query plan is no issue since the integrated engine of the PolyDBMS provides support for all operators and therefore compensates for missing functionality of the data stores. However, in practice, we run into issues with the length of the resulting query string (e.g., the resulting SQL string) used to interact with the data store. Executing the operations of the example query not supported by the underlying data store in the integrated engine of the PolyDBMS would cause the query string to contain the individual value of the level attribute for every employee. Furthermore, a query changing multiple records to different values is rather complex and therefore might also exceed the capabilities of the data store.

We therefore introduce the **stream iterator** operator depicted in Figure 9.7. This operator is implemented by the integrated engine of the PolyDBMS. It iterates on the result received from the left input branch and invokes the right branch with these values as dynamic parameters. This allows the PolyDBMS to implement arbitrary complex DML queries on data stores that only support the previously introduced minimal set of operators. The left branch of the query tree can be routed independently of the right branch (see Chapter 10). This allows the left branch containing operators that are not supported on the underlying data store to be executed on a data store with support for this operators—or it is executed within the integrated engine of the PolyDBMS. This is possible since the left branch of the tree is read-only and thus only requires support for a scan operation on the underlying data store. The **stream iterator** can also be combined with the **parallel modify** operator. If supported by the underlying data stores, the actual execution can also be done in batches.

Besides unsupported operations, the **stream iterator** also presents a solution for handling data manipulation operations referencing data that is not allocated to the same data store (e.g., due to partitioning of the schema entity). Furthermore, it enables the modification of data stored according to the schema mapping outlined in Section 8.4. This allows to meet the *Independence of Storage Configuration* requirement for arbitrarily complex data modification queries.

9.4 Enforcement of Constraints

As outlined in Chapter 8, a data model might also define building blocks for specifying constraints to be imposed on the data governed according to this data model. It needs to be distinguished between two types of constraints: those applying to an individual data item (e.g., nullability constraints or the data type) and those applying to the data entity or the whole schema object. While the former type can easily be enforced, the latter is more challenging.

One approach to enforce constraints in a PolyDBMS is to delegate the enforcement to one of the underlying data stores supporting the enforcement of this type of constraint. However, this requires all affected data to be placed on this data store. Since not all data stores support the enforcement of constraints and this approach cannot be applied for partitioned data entities, other solutions are required. Furthermore, solely relying on such a delegation approach might violate the *Independence of Storage Configuration* requirement.

In database systems, uniqueness constraints are typically enforced using indexes. This is also a solution in a PolyDBMS. Thanks to the previously introduced **stream iterator**, correctly implementing such an index update is possible. However, maintaining indexes on the PolyDBMS level introduces a significant overhead, especially with complex update queries. Furthermore, indexes are only a feasible solution for enforcing existence-based constraints (uniqueness, foreign key) but they are not feasible for other kinds of constraints like check-constraints or the enforcement of a graph schema. These constraints can only be enforced using a query that checks whether the main query violates any constraints. The efficiency of this approach depends on the underlying data stores. The uniqueness of a certain value can for instance be checked very efficiently if there is a data store with a matching index. Since indexes are cheaper to maintain on the underlying data stores than on the PolyDBMS level, this approach can also be efficient for enforcing existence-based constraints. However, this heavily depends on the data, the workload and the involved data stores.

Since all three approaches have their strength and weakness, a PolyDBMS should consider all of them. For constraints that are enforced on query time, this requires a universal approach to express the enforcement as part of the query plan that allows the PolyDBMS to choose the proper enforcement technique at runtime. We therefore introduce the **conditional execute** operator. This operator only executes the right branch if the result of the left branch fulfills a specified condition. The left branch can be used to model all

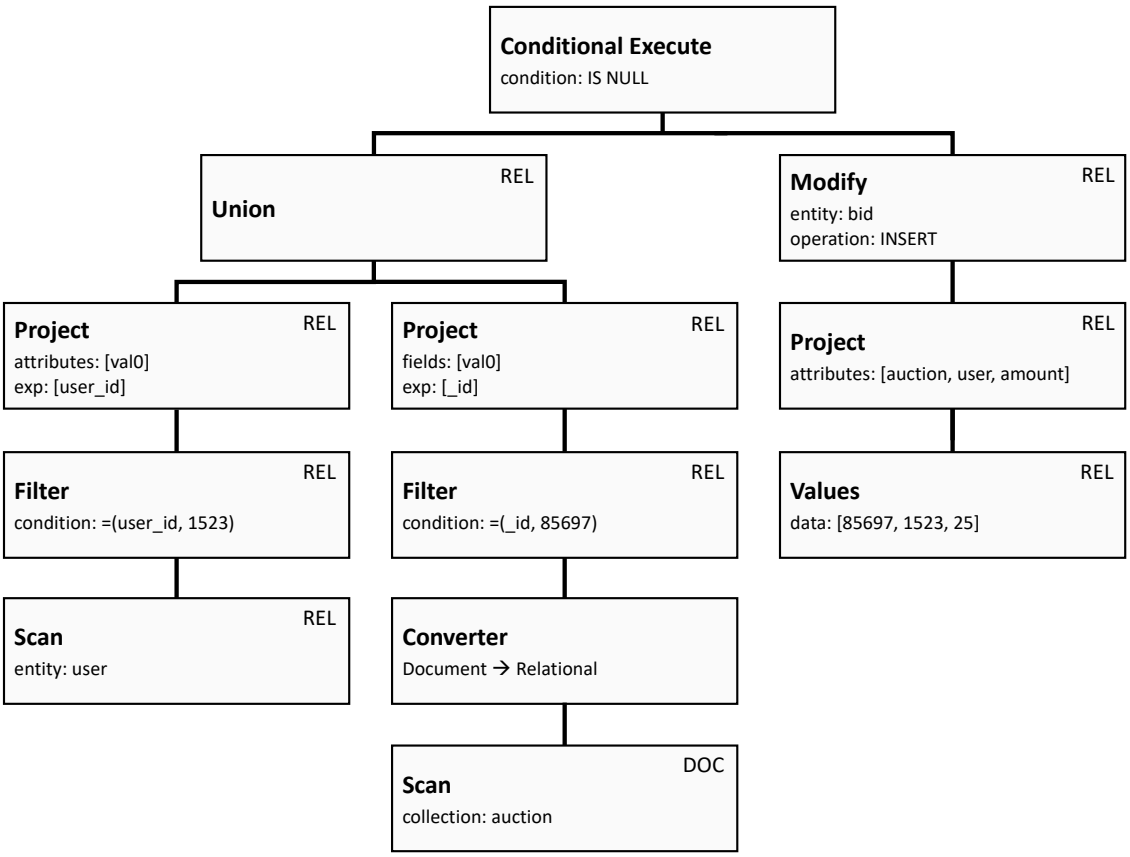


Figure 9.8 The conditional execute operator is used to enforce two foreign key constraints. Since auctions are stored using a document data model, the enforcement combines operators from different data models. The necessary converts are added in the implementation step.

three approaches. Figure 9.8 depicts a plan enforcing the uniqueness of the foreign keys auction and user of the bid table in the context of an insert query. Instead of the union construct, it would also be possible to stack multiple **conditional execute** operators.

The advantage of this construct is that the left part can be arbitrarily complex. This allows to represent even complex check constraints. Since everything is represented in one plan, synergies and redundancies can be eliminated. Furthermore, it is possible to combine different enforcement techniques.

But wherever there is man, there must be some sort of route.

— Robert Edison Fulton Jr.

10

Query Routing

In a PolyDBMS, data can be replicated and partitioned across multiple data stores. To execute a query, the PolyDBMS must plan the execution by decomposing the query and selecting which data store has the best characteristics for executing the query or parts thereof. This process of planning the execution of a query in terms of the optimal utilization of the available data stores is referred to as *query routing*.

What makes query routing complicated is that the execution time of a query on a data store depends on several factors, such as the amount and type of data stored or the performance of the system on which the data store is deployed. In addition, these factors may also change over time. This requires the query routing to be adaptive and to adjust to the conditions of the environment and the use case.

Simply speaking, query routing is about transforming the **scan** operators of a query plan. This includes resolving the partitioning and modeling the access to the individual logical partitions, as well as deciding from which data store the data should be read. In this chapter, we introduce our *four-phase query routing* approach. As the name implies, it consists of four phases:

The first phase is the *resolution phase*. In this phase, the data entities are broken down into their individual logical partitions. The PolyDBMS thereby analyzes the query and determines which of the partitions must be considered for the execution of the query.

In the *parameterization phase*, the query is transformed into a form that allows it to be reused for similar queries. This allows subsequent routing and query processing steps to be cached. Furthermore, the identification of similar queries is also important for the selection strategy.

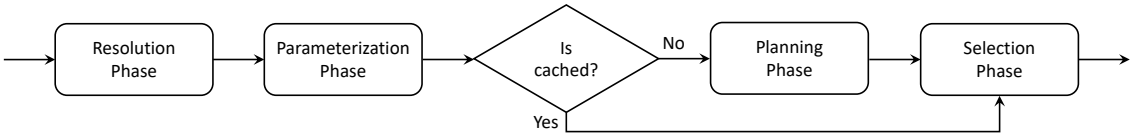


Figure 10.1 The four phases of the introduced query routing model. As depicted, the time-consuming *planning phase* is cached for similar queries.

In the *planning phase*, candidate plans are generated by deciding for each of the **scan** operators, on which data store it should be executed. This phase results in a set of candidate plans.

The final *selection phase* decides which of the previously generated candidate plans should be used to execute the query.

Figure 10.1 depicts the four phases. The caching is an elementary part of this concept since it allows to skip the time-consuming planning phase if a similar query has already been planned. In the following sections, these phases are elaborated in more detail.

10.1 Resolution Phase

In the resolution phase, data entities are resolved to the required set of corresponding logical partitions. As outlined in Section 8.5.1 all data entities have a partition function assigned. Thus, all data entities are inherently partitioned. However, due to the NONE partition function, a data entity can consist of only one partition. Since the horizontal partitioning is defined on the logical data entity, the resolution is independent of which data store(s) the partitions will be read.

Figure 10.2 exemplifies the transformation of a query plan done in the resolution phase. The query in this example selects two users (the one with the ID 34 and the one with the ID 2584). In this example, one user record is stored in the partition 0 and one in the partition 2. Both partitions are therefore unioned together.

The **scan** operators for the individual partitions contain a list of all available placements (i.e., allocations to a data store) of this partition. Depending on the freshness requirements of the query (see Section 8.5.2), this list also contains outdated placements which—at this point in time—meet the requirements. Furthermore, it also contains a list of all *vertical partition entities* (see Equation (8.46)) available at this placement.

The obtained query plan is not optimized. This can for instance be seen in the example depicted in Figure 10.2. In this plan it would probably be beneficial to move the filter

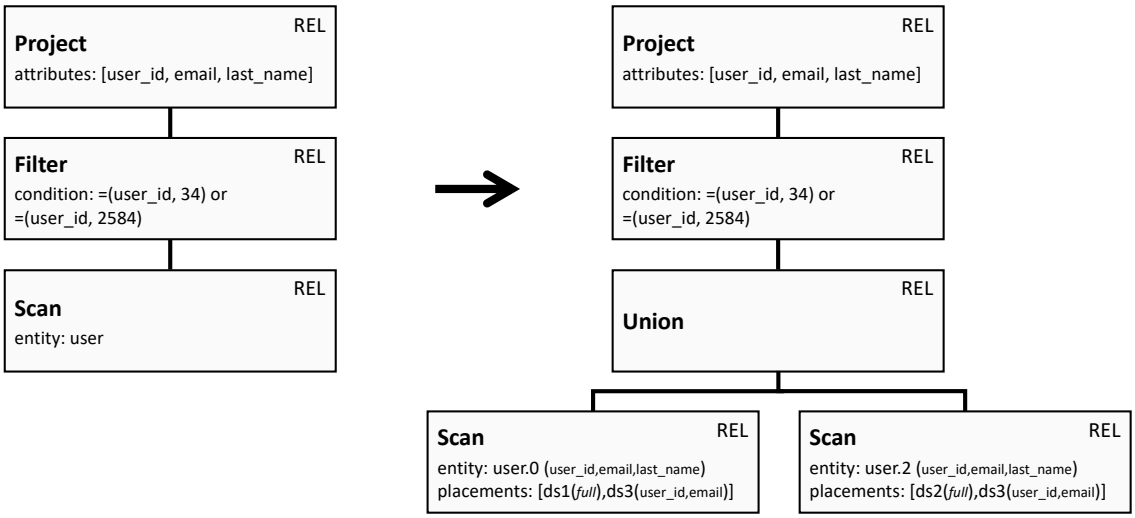


Figure 10.2 This example shows a query retrieving two user accounts by their IDs. In the resolution phase, it is identified that the query accesses data spread across two partitions. The corresponding partition are therefore unioned together. Furthermore, the scan operations on the individual partitions contain a list of all available placements of this partition.

below the union operator in order to reduce the amount of data that needs to be unioned. However, since it has not yet been decided from which data stores the data will be read, the potential for optimization is rather limited at this stage.

The required set of partitions is determined based on the conditions in the query plan. Since these conditions can contain function calls or conditions that cannot be evaluated without partially executing the query, the fallback is to union all partitions and to rely on subsequent optimization steps of the PolyDBMS and the underlying data stores.

More formally: with the set of possible logical query plans Q^{L} being the set of all query plans consisting solely of logical operators, the resolution phase can be described as a function

$$\begin{aligned} Q^{\mathrm{L}} &\to Q^{\mathrm{L}} \\ q &\mapsto q^{*}. \end{aligned} \tag{10.1}$$

10.2 Parameterization Phase

The routing of queries and the subsequent steps in the query processing can be quite complex. This complexity can introduce a significant overhead that is especially problematic for short running transactional queries. To reduce the amount of work that needs to be done for every query, parts of the query routing and processing should be cached and only done once for *similar* queries.

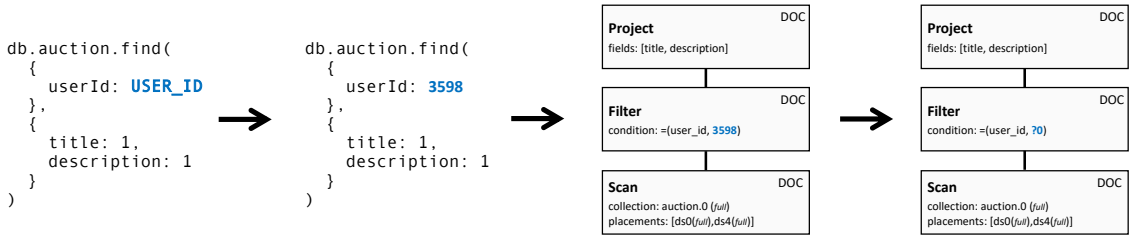


Figure 10.3 The template of a MongoDB query is used to retrieve the title and description of the auction with the ID 3598. As depicted, this results in a query plan with this ID in the filter condition. By replacing the literal 3598 with a parameter, we obtain a query plan that can be reused for all queries retrieving an auction that is stored on this partition.

We define the similarity of queries based on the structure of the query plan. This is based on the observation, that especially transactional workloads produced by applications are generated based on templates. The application typically derives queries from these templates by replacing defined placeholders with literals. By removing the literals from the query plan, we get a query plan which can be reused for queries with a different set of literals.

The process of removing literals from the query plan is called *parameterization*. In this process, all literals in the query plan are replaced by placeholders. Figure 10.3 depicts the parameterization at the example of a simple query plan originating from a MongoDB query. The parameterization only replaces literals. Operators, functions or identifier names are not substituted.

Since the parameterization is done on the query plan obtained from the resolution phase, it results in a generic query plan that can be reused for all queries where the same set of partitions is accessed; thus, where the literals (e.g., the user_id in the example) belong to the same set of partitions. Together with the list of placements of the respective partitions available at this point in time, all information needed for the next routing phase are encoded in this generalized query plan.

This approach results in a robust and easy to maintain solution for caching the time-consuming planning phase. Since the list of available placements of a partition added to each **scan** operator in the previous phase is part of the query plan, adding or removing placements will result in a different query plan and thus a new planning. This also covers data freshness: if a placement no longer meets the freshness requirements of a query, it will not be added to the list of placements in the resolution phase.

Formally, the parameterization phase can be described as a function

$$\begin{aligned} Q^{\mathrm{L}} &\to Q^{\mathrm{L}} \\ q &\mapsto q^{*}. \end{aligned} \tag{10.2}$$

10.3 Planning Phase

The planning phase is the most complex phase of the four-phase query routing. In this phase, the query plan expressed using operators of the logical algebra is implemented using the physical algebra. If there are multiple placements of a logical partition on different data stores, the planning needs to decide from which data store the data should be read (i.e., where to execute the **scan** operator).

The decision on which data store a **scan** operator is executed determines whether other operators can also be pushed down. As outlined in Section 9.2, *push down* refers to the execution of operators on the underlying data store instead of the integrated engine of the PolyDBMS. Since pushing down operators avoids streaming the whole data entity to the integrated engine of the PolyDBMS, it usually results in significantly better performance.

The decision on which data store the **scan** operator is executed therefore influences the potential for pushing down additional operators. For an operator to be pushed down, this operator and all other operators between this operator and the **scan** operator need to be supported by the data store. Since not all data stores provide the same set of operators, the decision on which data store to execute a **scan** operator can therefore have a crucial impact on the performance of a query.

As already mentioned for data modification queries in Chapter 9, it would theoretically be possible to implement individual operators on a data store by streaming the data from and to the data stores. However, since database systems typically restrict the maximum length of a query string, this option is not feasible. Hence, for an operator to be pushed down on a data store, all previous operators starting from the **scan** operator need to be pushed down as well.

In order to come up with an optimal plan for executing a query, the capabilities and the expected performance of the data stores need to be considered. Especially for heavily replicated and partitioned data entities, finding such an optimal plan can be complex. However, due to the parameterization of the query plan, this phase can be cached. This is especially important for short running transactional queries where the total execution time should be a few milliseconds.

With Q^{L} being the set of all logical query plans and Q^{P} being the set of all physical query plans, the planning phase can be described as a function

$$\begin{aligned} Q^{\mathrm{L}} &\rightarrow Q^{\mathrm{P}} \times Q^{\mathrm{P}} \times ... \times Q^{\mathrm{P}} \\ q &\mapsto \left\{q_1^*, q_2^*, ..., q_N^*\right\}. \end{aligned} \tag{10.3}$$

10.3.1 Query Decomposition

There are data stores with different sets of features and capabilities. While a data store might excel for some types of data and workloads, it performs badly for other types. Since database queries can be arbitrarily complex, there might be no single data store suitable for it. The idea of query decomposition in a PolyDBMS is to split the query into multiple parts that are then executed on different data stores. This allows to utilize and combine of the strength of multiple underlying data stores.

Using the query representation introduced in this thesis, decomposing queries is fairly simple. Since every read-only query plan has **scan** operators on all of its leaves, a query can be decomposed by implementing the **scan** operator using operators from different data stores. In the subsequent query optimization, supported operators are pushed down, resulting in a decomposed query.

10.3.2 Vertical Partitioning

As outlined in Section 8.5.1, data can be partitioned horizontally and vertically. The horizontal partitioning has already been resolved in the first phase. The vertical partitioning, however, has not yet been dealt with. As described in Section 6.1, in our schema model, vertical partitioning refers to a partitioning that affects both the schema and the data. Hence, the physical partitions allocated to the underlying data stores have a different schema. Resolving vertical partitioning therefore results in some sort of **join** operation that "glues" the distributed parts of a record, document, node or edge together.

As part of the resolution phase, every **scan** operation has been extended with a list of data placements including the vertical partition entities they provide and a list of required vertical partition entities. In the planing phase, these lists are used to determine an optimal combination of multiple partitions. Since vertical partitioning always results in multiple data stores being involved, these **join** operators cannot be pushed down. The routing therefore tries to first deal with the horizontal partitioning and implement as many operators as possible before addressing the vertical partitioning. The rules for merging and splitting operators and changing their order is the subject of query optimization. Since there are no fundamental differences to "traditional" database management systems, there is no need to elaborate this in more detail. For the relational data model, for example, these rules can be found in [Cha98].

10.3.3 Static Planning Rules

One approach for selecting where (i.e., on which data store) to execute a **scan** operator and to find a valid execution plan for data spread across different data stores is by means of a static set of rules. This set of rules can be extended based on the use cases and the requirements of the scenario. Every rule can return one or multiple candidate plans.

An obvious approach is to select a candidate plan based on the data model of the **scan** operator and to prefer native data stores. This increases the chance of pushing down additional operators. Furthermore, native data stores are expected to perform better than data stores based on a different data model.

Another approach is to identify the type of workload and select the data store based on the identified workload. This is especially interesting for simple queries where everything can be pushed down to one data store.

The third approach is to decide based on the usage of certain features (e.g., nearest neighbor search or geospatial functions). If a query contains such a special function, it should be preferred to utilize a data store with native support for this feature.

The fourth rule deals with the latency of the communication between the data store and the PolyDBMS. It prefers data stores with a low latency (e.g., those deployed on the same machine as the PolyDBMS). This is especially beneficial for simple, short running queries where the communication overhead quickly exceeds the execution time.

Every candidate plan proposed by the planning rules is optimized in a subsequent step. This includes identifying operators that can be pushed down to the underlying data stores, merging and splitting operators and adjusting their order.

The advantage of the rule-based approach is that it is very fast. The rules can be applied in parallel and result in a set of query plans. However, it is not very adaptive. While it is possible to adapt the rules at runtime, especially with complex queries, this approach reaches its limit.

10.3.4 Cost-based Planning

A major disadvantage of the static planing rules is the split between the decision where to execute a query and the optimization of the query. Merging these two steps would allow to select the data store according to the potential optimizations its choice enables.

By using a cost-based query optimizer, this can be modeled. However, the difficulty is to design the cost-model. This cost-model must assign costs to every operator of the physical algebra. The ability to dynamically adjust this cost model results in a highly adaptive approach for query planning. However, building and adjusting this cost model is quite difficult.

Optimally, the cost model is designed such that the cost for a query plan implemented using operators from data store *A* and a query plan implemented using operators from data store *B* reflect the relative difference between the actual execution time of the two query plans. Since the execution time of a query plan does not only depend on the data store but also on the environment and the data, a static cost model has its limitations. We are therefore working on an approach for learning the cost model based on the previously executed queries. Our approach is to adapt existing techniques developed for single-model database systems (see Section 5.2) and apply them to a PolyDBMS.

The most important factor is, that the query optimizer takes the amount of data that needs to be streamed to the integrated engine of the PolyDBMS into account. This results in query plans that maximize the push down of operators to the underlying data stores. However, the disadvantage of this approach is the required time to find an optimal query plan, especially with a large number of placements. Furthermore, it is difficult to design a static cost model for simple transactional query plans since for such queries, aspects of the deployment like the latency for communicating between the PolyDBMS and the data store can result in an overhead exceeding the actual execution time of the query.

10.3.5 Combining Both Approaches

Both approaches have their advantages and disadvantages. The integration of the data store selection and planning into the query optimization results in good query plans. With a learned cost model, query routing also gets adaptive. However, a cost-based planning is more time-consuming than applying static routing rules. Furthermore, simple query plans are challenging since network latency and other aspects are more significant than the actual execution time of the query on an underlying data store.

While we are currently working on integrating these aspects by learning and adjusting the operator costs at runtime, this introduces another issue: In order to learn the operator costs, enough data points need to be available. After significant changes to the data allocation and the set of data stores, the PolyDBMS suffers from an outdated cost

model. Furthermore, the cost-based optimization is time-consuming if a large number of placements needs to be considered (e.g., with partitioned data entities).

Our approach is to combine the cost-based planning with the static routing rules introduced before. The static rules are especially advantageous for simple, short running queries and can be adjusted to the scenario. They are furthermore very fast.

The idea is to identify the type of query: simple and potentially short running or complex and potentially long-running. For simple queries, only the static routing rules are used and the resulting query plans are optimized. For long-running queries, both the routing rules and the cost-based planning approach are being used. However, since we do not have a reliable technique to classify queries as short-running and long-running without executing them, the static planning rules still need to be sophisticated enough to handle misclassified analytical queries.

In both cases, we obtain a set of potential query execution plans. Duplicate plans are removed. These plans are then cached for subsequent queries which are similar to this one.

10.4 Selection Phase

The selection phase is the final phase of the four-phase query routing. Its purpose is to select an execution plan from the set of candidate plans generated by the planning phase. While the planing phase is cached for subsequent similar queries, the selection phase is applied to every query. The reason for this split of planning and selection phase is to enable a caching of the planning phase: the dedicated selection phase allows making a final decision on the query plan based on the current state of the system and by taking previous executions or updates to the cost model into account. This avoids constantly re-planning query plans.

The selection is done using *score functions*. These score functions allow to select which of the previously proposed execution plans is optimal given the current state of the system and collected knowledge about previous executions of similar queries.

In the following sections, we outline multiple score functions. Every score function $\mathtt{sf}(\cdot)$ returns a value between zero and one. Furthermore, there is an adjustable weight w for every score function. The score S of a physical query plan q is calculated as

$$S := \sum_{i=1}^{n} w_i \cdot \mathtt{sf}_i(q) \tag{10.4}$$

with n being the number of score functions. Instead of always executing the plan with the highest score, it has shown advantageous to normalize the scores and use them as probabilities. The plan with the highest score is selected most frequently, while plans with lower scores are selected less frequently. This makes things even more adaptive, especially with the scoring function we introduce in Section 10.4.1. Executing these less efficient query plans allows the PolyDBMS to learn about changes and update its cost model. This can for example be the case due to growing amounts of data or external effects like changes on the host system or the network.

The selection phase can be described as a function

$$\begin{aligned} Q^{\mathrm{P}} \times Q^{\mathrm{P}} \times \ldots \times Q^{\mathrm{P}} &\to Q^{\mathrm{P}} \\ \left\{q_1, q_2, \ldots, q_N\right\} &\mapsto q^*. \end{aligned} \tag{10.5}$$

10.4.1 Previous Execution Time

A very powerful technique is to score the candidate plans based on their actual execution time. This is an approach we have first described in [VSS17]. It makes use of the introduced parameterization technique for identifying similar queries and the observation that queries are typically derived from templates.

This score function requires the PolyDBMS to monitor and record the execution time of every chosen execution plan. This allows to compare the average execution times of the proposed candidate plans. By normalizing the execution times, it is possible to calculate a score. If there are no execution times for an execution plan (thus this plan has not yet been executed), the maximum score of 1 is assigned. Depending on the other scoring functions and their weights, this fosters the execution of this query plan, resulting in an execution time to be recorded. This eventually converges to a state where all candidate plans have been tried, and the fastest query plan is assigned the highest score.

With a function $\mathtt{at}(\cdot)$ that returns the average execution time of an execution plan or zero if there are no previous executions, the scoring function $\mathtt{sf}_{\mathrm{pev}}$ for a physical plan q of the set of candidate plans $Q^* \subset Q^{\mathrm{P}}$ is given as

$$\mathtt{sf}_{\mathrm{pev}}(q, Q^*) := 1 - \frac{\mathtt{at}(q)}{\mathtt{max}\left(\{\mathtt{at}(*) \mid * \in Q^*\}\right)}. \tag{10.6}$$

This approach is adaptive, especially if only recent execution times are considered for calculating the average execution times (i.e., using a sliding window approach). An issue of this scoring function are parallel workloads. The obtained values are influenced by

parallel executions on the data store. However, we realized that it is also an advantage in scenarios with relatively constant workloads. In such scenarios, it perfectly depicts the actual execution times that could be achieved on this data store and thus also results in some sort of load-balancing between the data stores.

10.4.2 Operator Cost Model

This scoring function uses the operator cost model introduced in the planning phase to calculate the cost of a query plan. With a function $\texttt{cost}(\cdot)$ that estimates the cost of an execution plan based on its operators, the scoring function $\texttt{sf}_{\texttt{cost}}$ for a physical plan q of the set of candidate plans $Q^* \subset Q^{\text{P}}$ is given as

$$\texttt{sf}_{\texttt{cost}}(q, Q^*) := 1 - \frac{\texttt{cost}(q)}{\texttt{max}\big(\{\texttt{cost}(q^*) \mid {}^* \in Q^*\}\big)}. \tag{10.7}$$

Using the cost-model in the selection phase increases the adaptivity of the query routing, since changes to the cost-model influences the routing even for cached query plans.

Creativity is thinking up new things.
Innovation is doing new things.

— Theodore Levitt

11 Polypheny-DB

With Polypheny-DB, we present a fully working implementation of a PolyDBMS and the concepts introduced in the previous chapters. Polypheny-DB goes beyond a typical research prototype: It is a full-featured database management system released under the Apache 2 license. The source code is available on GitHub[1].

The maturity of Polypheny-DB is also evidenced by the fact that it was selected twice in a row by Google for the *Google Summer of Code*. The success of Polypheny-DB was also made possible through multiple excellent student projects (see page 213 for a list of projects).

In this chapter, we give a brief overview on the architecture and capabilities of the Polypheny-DB system. More details on specific aspects of the implementation can be found in the documentation published on our website[2].

11.1 Overview

Polypheny-DB is the heart of the growing Polypheny ecosystem. Besides the database system itself, there are various applications, drivers and tools:

- **Polypheny-UI.** A feature-rich and easy to use browser-based user interface for Polypheny-DB. It is deployed together with Polypheny-DB and allows a convenient management of a Polypheny-DB instance. This includes monitoring the current status of the system, modifying the configuration, defining and altering the schema, configuring data replication and partitioning, and executing and analyzing queries.

[1] https://github.com/polypheny/Polypheny-DB/
[2] https://polypheny.org/

- **JDBC Driver.** Polypheny-DB provides a standards-compliant JDBC query interface. This driver allows to conveniently access this interface from JVM-based query languages (e.g., Java, Scala, Kotlin). This driver also supports meta functions defined by the JDBC standard for retrieving schema information. The whole JDBC stack is implemented using *Avatica for Polypheny*, a fork of *Apache Avatica*[3] containing Polypheny specific adjustments and extensions.

- **Python Connector.** Similar to the JDBC driver, this connector provides an interface for developing Python applications that can connect to Polypheny-DB. The connector is a pure Python package that has no dependencies on JDBC or ODBC. It is released to the *Python Package Index* (PyPi) and can therefore be installed using pip on Linux, macOS, and Windows platforms. The connector follows the *Python Database API v2* specification (PEP-249).

- **Query-to-File.** This tool allows to materialize and navigate the result of an arbitrary query as a file system (set of files in a folder). Depending on the data type, the data is either kept in memory or (e.g., with blobs) is fetched on demand from Polypheny-DB. Besides a query, the tool also accepts the name of a data entity to be represented as a file system. By editing the content of the files, the corresponding data can be modified. Query-to-File also supports transactions. The primary use case of this tool are multimedia collections (e.g., videos, images, files) stored in Polypheny-DB. Query-to-File allows applications that read files from the file system to operate on data governed by Polypheny-DB.

- **Polypheny Hub.** A platform for storing and exchanging multimodel datasets and schemas. The frontend is seamlessly integrated into Polypheny-UI. The corresponding server application maintains and stores the actual datasets and schema definitions. Importing and exporting datasets and schemas can conveniently be done directly within Polypheny-UI. The system features a role-based user management. All maintenance tasks can be done through the Polypheny-UI. While we maintain a public instance of the Polypheny-Hub server, own instances can be deployed as well.

In this chapter, we focus on Polypheny-DB itself. The system is developed in Java. As mentioned before, the source code is available under the Apache 2 license. Furthermore, we provide binary releases[4] for the Windows, macOS and Linux platform. These releases come with an installer and include a Java runtime environment.

[3] https://calcite.apache.org/avatica/
[4] https://get.polypheny.org/

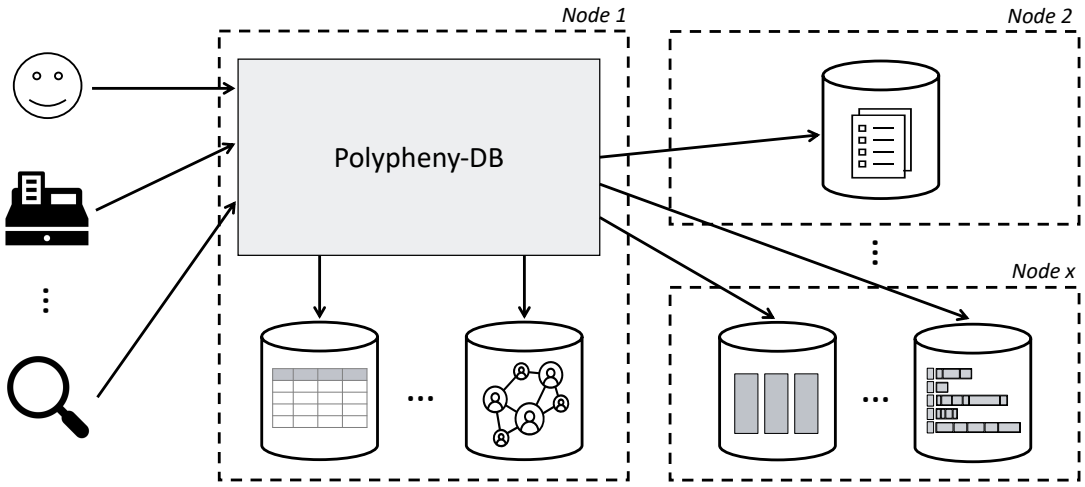


Figure 11.1 Deployment of Polypheny-DB. The data stores can be deployed on the same or on other machines. All queries go through Polypheny-DB.

According to the hybrid architecture model of a PolyDBMS discussed in Chapter 7, Polypheny-DB uses underlying data stores for storing data and executing (parts) of a query. As depicted in Figure 11.1, the data stores are either deployed on the same machine as Polypheny-DB or on other machines. This allows an easy horizontal scaling of a Polypheny-DB deployment.

The underlying data stores are expected to be under the full and exclusive control of Polypheny-DB. However, the ability to query independent databases or sources of information known from polystore systems (i.e., *data lake*) is available in Polypheny-DB as well. Besides data stores, Polypheny-DB also has the concept of *data sources*. A data source can be everything from a plain file (e.g., a CSV file) over a traditional database system (e.g., a PostgreSQL instance) to a blockchain (e.g., the Ethereum blockchain). In contrast to data stores, data sources are not expected to be under the full and exclusive control of Polypheny-DB, but rather shared with other participants. A query can access and combine data from multiple data sources. It is also possible to access and combine data from data sources and data stores within the same query.

The schema mapping technique is different for data sources and data stores: For data sources, Polypheny-DB applies a *Local-as-View* (LAV) mapping, describing which parts of a logical entity are physically stored on which of the underlying data stores and how the data is to be transformed. For data sources, in contrast, a *Global-as-View* (GAV) mapping is being applied that describes which parts of the data source are integrated into the global schema and how the transformation is performed. Since Polypheny-DB supports views combining entities from data sources and data stores, the schema mapping can be described as *Global-and-Local-as-View* (GLAV).

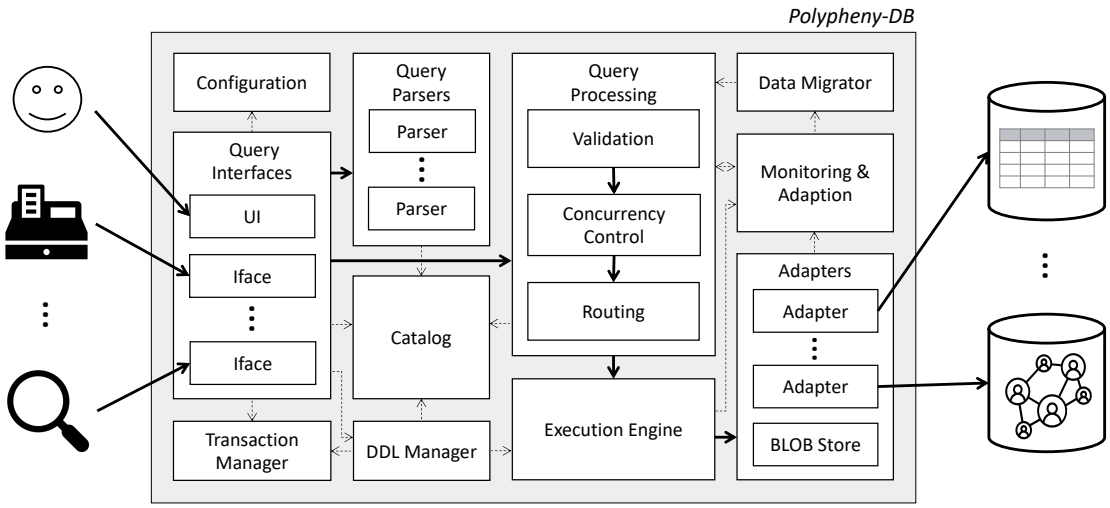


Figure 11.2 Simplified architecture of Polypheny-DB, depicting the most important components. The solid arrows indicate the path of a DQL or DML query. The dashed arrows depict communication between the components.

Polypheny-DB implements the concepts introduced in the previous chapters and also supports the three data models (relational, document and labeled property graph) used to exemplify these concepts. Besides native support for these data models (according to the requirement outlined in Section 3.1.1), Polypheny-DB also provides support for at least one query language and data store based on each of these data models.

11.2 Architecture

The Polypheny-DB system is written in Java and has been developed with a strong focus on modularity and the ability to adapt to changes in the workload or the available infrastructure at runtime. As depicted in Figure 11.2, the software is structured in several components. Additional query interfaces, query language parsers, or adapters can be deployed at runtime.

Polypheny-DB has all components of a "traditional" database system (cf. Section 5.1). With the *BLOB Store*, there is also a storage engine integrated within Polypheny-DB itself. However, besides the fact that there are multiple query interfaces and query language parsers, there are three major differences to a traditional database system.

Firstly, the position of the *concurrency control* component. In the architecture we introduced together with the foundations of database systems (Figure 5.1), the concurrency control happens while executing the query. With the multiple storage and execution engines containing replicated data, concurrency control is more similar to a distributed

database system, requiring locks on the logical data entities instead of the physical storage structures.

Secondly, the *routing* component and the *adapters*. These are elementary components of a PolyDBMS that enable the efficient usage of underlying databases as storage and execution engines.

Thirdly, the lack of a dedicated *logging & recovery* component, which is a result of the architecture using underlying databases as storage engines. Since recovery is a functionality that can ultimately only be provided efficiently by the storage engines, Polypheny-DB relies on these very engines. Hence, the necessary logic is implemented in the respective adapters. Furthermore, Polypheny-DB also has a recovery logic for its catalog.

In the following paragraphs, we briefly introduce the components depicted in Figure 11.2. Since the implementation of Polypheny-DB closely follows the concepts outlined in this thesis, we refer to the previous chapters for a detailed description of the schema model, query representation, and query routing.

Query Interfaces. In Polypheny-DB, we strictly distinguish between query interfaces and query languages. A query interface provides a method to interact with Polypheny-DB according to a defined protocol. This includes accepting queries expressed in one or multiple query languages, transaction control, and accessing meta information. The query interface is also responsible for encoding and returning the result. There can be multiple query interfaces accepting the same query language. Furthermore, a query interface may accept multiple query languages. The default-namespace used for resolving identifiers is defined individually for every query interface.

Query Parsers. Queries received by the query interfaces are parsed and translated by a corresponding query parser. There is a dedicated parser for every query language. The output of the query parser is a query plan constructed using the algebra of the corresponding data model.

Validation. The validation component checks whether a query plan is valid. This includes checking that there are values for all mandatory fields and that data types match the data types of the schema. Since identifiers are checked by the query parser, the validation is especially important for data modification queries.

Concurrency Control. The purpose of concurrency control is to ensure that parallel queries do not violate the guarantees set forth in Section 5.3. In Polypheny-DB concurrency control is provided using *strong strict two-phase locking* (see Section 5.4). The

locking is done based on the partitions (every data entity consist of at least one partition; see Section 8.5.1). The implementation includes the necessary detection and handling of deadlocks.

Router. The routing component plans the execution of a query by selecting on which of the underlying data stores a query or parts of it should be executed. The component is implemented according to the concept introduced in Chapter 10. The query optimization is done using a modified and extended version of a Volcano optimizer [GM93] we have forked from the Apache Calcite project [BCRH+18].

Execution Engine. The execution plan produced by the query routing component is implemented as Java code and compiled at runtime. Since these implementations can be cached, this approach results in a very good performance for subsequent queries.

Adapters. Compared to a traditional database system, the adapters take over the role of the *storage manager*. Every adapter provides a set of operators that can be executed using this adapter. These operators form the physical algebra used to construct execution plans. Furthermore, the adapter is responsible for maintaining the connection to the corresponding data store or data source. There can be multiple instances of the same adapter connecting to different data stores or data sources. The *BLOB Store* is a special adapter that implements a whole data store. It is intended for storing large binary objects such as videos.

Catalog. The catalog maintains the schema and other metadata. Furthermore, it stores deployment specific information such as the set of adapters and interfaces. It also maintains the allocation of partitions to data stores. Moreover, the catalog stores the definition of views and stored procedures.

Transaction Manager. Similar to a traditional database system, the transaction manager maintains the ongoing transactions. It receives *transaction commands* from the query interfaces, allowing to begin and end transactions.

DDL Manager. The purpose of this component is to process and execute changes to the schema. It orchestrates the changes to the physical schemas of the underlying data stores, triggers the data migrator, and performs the necessary changes to the catalog. The DDL Manager therefore plays an essential role in implementing the schema model outlined in Chapter 8.

Data Migrator. The data migrator copies data between data stores and therefore allows changing the data allocation and data partitioning layout of a running Polypheny-DB instance.

Monitoring & Adaption. This component monitors the workload and creates statistics on data governed by the Polypheny-DB instance and the type of workload. It provides important information for planning and optimizing queries. Furthermore, it allows basic adaptions of the systems based on pre-defined policies.

11.3 Capabilities

In this section, we give an overview of the functions and capabilities of Polypheny-DB. This includes the set of supported query interfaces, languages and adapters. However, this is not an exhaustive list that includes every aspect. Rather, it is an overview on those features that are relevant for the evaluations and benchmarks presented in the next part.

As mentioned before, Polypheny-DB and other parts of the Polypheny ecosystem have greatly benefited from various contributions made in the context of student projects or the Google Summer of Code. This especially applies to the implementation of the interfaces, languages, adapters, and features mentioned in this section. Furthermore, parts of the implementation are based on code from the Apache Calcite project. The implementation also makes use of existing parsers and libraries. A complete overview of all contributors as well as an overview of all used libraries can be found on the website and in every release of Polypheny-DB. We would like to thank everyone who has contributed to the Polypheny project or the projects on which it is built.

11.3.1 Query Interfaces

Polypheny-DB comes with the following query interfaces:

- **Avatica.** Apache Avatica[5] is a framework for building database drivers. With *Avatica for Polypheny*, we have created a fork of the Avatica project containing Polypheny-specific adjustments and extensions. The Avatica query interface is used by both the JDBC driver and the Python connector. The queries are serialized using *Protbuf*[6].

- **HTTP.** A query interface that can easily be integrated in all types of applications. Queries are accepted using HTTP POST requests. There is a route for every query

[5] https://calcite.apache.org/avatica/
[6] https://developers.google.com/protocol-buffers

language supported by Polypheny-DB (e.g., '/sql'). The query itself needs to be placed in the body of the POST request. The result is serialized as JSON.

- **REST.** A query interface that does not require a query language to express queries. Instead, the query is expressed in a RESTful fashion. It primarily supports querying relational namespaces. Support for other data models is limited.

- **Polypheny-UI.** The powerful browser-based user interface of Polypheny-DB. It allows executing and analyzing queries expressed in all supported query languages. Furthermore, it contains additional query methods such as explore-by-example. Additionally, it also allows to graphically build and execute logical query plans.

11.3.2 Query Languages

As it is required for a PolyDBMS, Polypheny-DB supports multiple query languages. The queries expressed in these query languages are accepted through one our multiple of the query interfaces mentioned in Section 11.3.1. The supported query languages include:

- **SQL.** The most widely used query language for (relational) databases [Bat18]. The language is well suited for structured data organized according to the relational model. Despite the fact that there is a SQL standard [Sec87], there are differences in the SQL implementations of different database systems. The implementation in Polypheny-DB is closely oriented at the SQL standard. When there is no clear standardization (e.g. for schema definition statements) we have followed the PostgreSQL dialect or defined our own syntax.

- **openCypher.** A very popular and widely adopted query language for querying labeled property graphs. The implementation in Polypheny-DB closely follows the openCypher specifications[7].

- **Contextual Query Language (CQL).** A query language primarily used in information retrieval systems such as search engines and bibliographic catalogs. The implementation in Polypheny-DB adds some extensions and additional keywords to the CQL specification[8]. However, the implementation does not support prefix assignments or search-term-only filters. The CQL query language, and thus our implementation, does not support data manipulation or the definition of schemas.

[7] https://opencypher.org/resources/
[8] https://www.loc.gov/standards/sru/cql/spec.html

- **MongoDB Query Language (MQL).** The query language of the popular document store MongoDB. It is intended for querying document schemas. The implementation provided in Polypheny-DB strongly follows the language definition of MongoDB version 5.0[9].

- **Pig.** A high-level language designed for working with MapReduce. It allows writing powerful analytical query scripts. The implementation in Polypheny-DB is adapted to operate on database entities instead of files. Our implementation does not support materializing results as files.

11.3.3 Adapters

Polypheny-DB distinguishes between *Data Stores* and *Data Sources*. Data stores are under the full and exclusive control of Polypheny-DB and serve as storage and execution engines for the data governed by Polypheny-DB. Data sources allow data from other sources (of a data lake) to be mapped into the schema of Polypheny-DB.

The set of data store adapters includes:

- **BLOB Store.** A data store integrated into Polypheny-DB that is optimized for storing large binary objects like videos or images. The adapter has a columnar storage layout and materializes every data item as a file. The BLOB Store supports two-phase commit and features a write-ahead log.

- **Cassandra.** A popular wide-column store (i.e. a two-dimensional key–value store). It has some similarities with the relational data model. However, the number and data type of the columns can vary from row to row.

- **Cottontail DB.** A column store tailored for multimedia retrieval applications. It is optimized for both Boolean and vector-space retrieval [GRH+20].

- **HSQLDB.** A row-oriented relational database system developed in Java. It supports an embedded mode that provides low-latency access from Polypheny-DB.

- **MonetDB.** A column-oriented relational database management system providing high performance for OLAP queries.

- **MongoDB.** A popular document database system that uses JSON-like documents with optional schemas.

[9] `https://mongodb.com/docs/v5.0/`

- **Neo4j.** A graph database based on the labeled property graph data model. According to the db-engines ranking[10] from May 2022, Neo4j is the most widely deployed graph data platform.
- **PostgreSQL.** A very popular and widely deployed row-oriented and feature-rich relational database system. It provides good performance for transactional workloads.

The set of available adapters for data sources additionally includes:

- **CSV.** This adapter allows to query CSV files as relational tables.
- **Ethereum.** An adapter for querying the Ethereum blockchain. It uses the Ethereum JSON-RPC API[11].
- **File System.** This adapter allows to represent a folder as a relational table. It allows to query meta information on the files as well as the content of the file itself. This is especially useful for working with multimedia files.
- **MySQL / MariaDB.** An adapter for the popular relational database system MySQL and its fork MariaDB. The adapter supports read and write queries as well as transactions.

11.3.4 Query Functions

The set and behavior of functions and operations is defined by the query language. Where applicable, Polypheny-DB follows the definition of the official documentation or standard of a query language. Furthermore, Polypheny-DB also supports additional functions. An example for this is the `distance` function available in our SQL implementation that calculates the distance between two vectors according to a metric.

All functions are supported by the integrated engine of Polypheny-DB. If a function is not supported by an underlying data store or if the underlying data store handles this function differently, Polypheny-DB executes it in its integrated engine. In order to improve performance, Polypheny-DB also registers user-defined functions on the underlying data stores to add or adjust certain functionalities.

[10] https://db-engines.com/en/ranking
[11] https://eth.wiki/json-rpc/API

Part IV

Evaluation

Preface to Part IV:
On the Art of Evaluations

In this part, we present the evaluation results of Polypheny-DB, our implementation of the concepts presented in this thesis. According to the holistic nature of the word *evaluation* this part also approaches the topic from different directions and using different methods.

In software development, *evaluation* refers to the assessment of software by executing it under defined conditions and with specified inputs. The practice of evaluating software is as old as the practice of software development itself [OU86]. In the context of database systems, it can primarily be distinguished between two types of evaluations: *verification* (i.e., the qualitative analysis) and *benchmarking* (i.e., the quantitative analysis).

Verification. This is the process of determining whether the software is working according to the specification and without technical errors [Boa11]. In Chapter 13 we introduce the blueprint of an approach for verifying the correctness of a PolyDBMS using randomly generated configurations and queries. Furthermore, we discuss how we have implemented this approach for verifying the correctness of Polypheny-DB.

Benchmarking. The performance of a system is measured by benchmarking it under specified conditions. This allows to compare the efficiency of the implementation with similar systems. Furthermore, it is an important part of the development process to identify performance bottlenecks and to optimize the system. The benchmarking results of our implementation Polypheny-DB are presented in Chapter 14.

A meaningful evaluation of a system requires both a holistic verification of the system and a comprehensive benchmarking of its performance. Benchmarking a system without verifying its correctness cannot prove superiority. A database that, for instance, always returns an empty result can be extremely fast—but is also useless.

Since the benchmarking of a system is a very tedious and time-consuming labor, we have developed a system called **Chronos** that automates the entire benchmarking process. In Chapter 12, we introduce the system. How Chronos is used for benchmarking Polypheny-DB is outlined together with the benchmarking results in Chapter 14.

benchmark *v.trans. To subject (a system) to a series of tests in order to obtain prearranged results not available on competitive systems.*

— Stan Kelly-Bootle, The Devil's DP Dictionary

12

Chronos

System evaluations are an important part of empirical research in computer science. They involve the systematic assessment of the runtime characteristics of a system depending on its parameters [Jai91; KLK20]. The consideration of all conceivable parameters and settings is a very time-consuming and tedious matter with numerous manual activities. Preferably, the exhaustive evaluation of an entire evaluation space can be fully automated. This includes the definition of the parameters, the scheduling and monitoring of the execution and the analysis of the results. In this chapter, we introduce *Chronos*, a system for the automation of the entire evaluation workflow [VSC+20].

12.1 Motivation

Benchmarking a system requires the systematic execution of defined workloads in a controlled environment [Jai91]. In the process, parameters of the system, the workload and the environment are varied to gain insights. Since the benchmark needs to be executed multiple times for all combinations of the varied parameters to obtain reliable results, evaluation campaigns can get extremely time-consuming and labor-intensive.

The idea of *Evaluations-as-a-Service (EaaS)* is to automate the evaluation process. The demand for an EaaS-system has already been identified in various publications [HMB+15; LE14; LE13; MET+21]. In comparison to benchmarking scripts often used by developers, an EaaS-system can speed up the process by orchestrating a parallelized execution while, at the same time, minimizing the required manual activities.

Another topic of high importance in the research community is *reproducibility* [Hu20; CHI15; SYF+20; FBS12]. An automated solution can make results more reproducible since all steps, configurations and results can be archived. Hence, the "human factor"

in the benchmark can be reduced. The authors of [Hu20] emphasis that "Replicability and reproducibility (R&R) are critical for the long-term prosperity of a scientific discipline" [Hu20]. An EaaS-system that automatically executes benchmarks and at the same time increases the reproducibility of results is therefore a great support for both, research and software development.

12.2 Existing Work on Automating Evaluations

In [HMB+15], the authors outline the issues of the *Data-to-Algorithm* approach of benchmarking were researches download datasets and execute their algorithms against these datasets themselves. The authors motivate the idea of Evaluations-as-a-Service where the algorithms come to the data (*Algorithm-to-Data*). The paper is focused on the demands and requirements of the machine learning and retrieval domain. The main focus of the paper is the application of such an EaaS approach for competitions and challenges.

In [LE14; LE13], the concept of evaluation-as-a-service is introduced to overcome restrictions with sharing Twitter data. Instead of distributing the data to all participants of a text retrieval competition, the authors developed an API for accessing the data. The introduced concept is therefore more an API service for evaluation than an actual evaluation system. The step towards an evaluation system is mentioned as future work: The authors propose that, instead of granting the participants access to potentially sensitive data like medial records, the participants submit their implementation which is then executed on a central infrastructure and without the participants touching the data.

The authors of [MET+21] introduce Dynaboard, a system for the automation of holistic evaluations of natural language processing (NLP) models using their benchmarking system Dynabench. The system allows NLP models to be uploaded to the Dynaboard platform. The uploaded models are then automatically evaluated in the cloud.

In [Hu20], the author proposes building a benchmarking framework for comparing methods and software tools in the Geo Information Systems (GIS) domain. The author motivates an Algorithm-to-Data approach in order to overcome various copyright and license issues with datasets. Furthermore, the author envisions building a system which allows to easily reproduce results of other researchers. This would allow to easily compare the results with the own work in order to prove superiority. The envisioned system should automatically report data on the experiments and make them publicly available by a unique identifier assigned to every execution of an experiment.

While there are several vision and position papers emphasizing the need for an EaaS-system [HMB+15; LE14; LE13], there are only very few implementations. Furthermore, only a subset of those implementations can be applied to the benchmarking of systems.

A major portion of the existing EaaS-systems, like for example Dynaboard [MET+21], PEEL [BAK+18] or mlpack [ESC14], are targeting the evaluation of machine learning models and algorithms. Most of the remaining systems are focused on a specific domain and environment. PROVA! [GBM16], for instance, is a tool for the distributed benchmarking of stencil compilers. We are only aware of two EaaS-systems suitable for the benchmarking the performance of systems.

The NIST Automated Benchmarking Toolset [CMM+00] allows the distributed execution of benchmarks using a distributed queueing system. The tool allows the definition of benchmarks and stores the collected data in a central repository. Unfortunately, there is no further information on this system. The tool seems no longer to be developed, and the source code is not available.

The BEEN system [KLM+06] is an EaaS-system that takes care of the deployment of the system under evaluation and schedules the execution of the benchmark in a distributed environment. All results are stored in a central repository and can be viewed and analyzed in a web user interface. BEEN also detects performance regressions between consecutive software versions. Unfortunately, BEEN seems no longer to be developed. Since the source code is not available, it is unclear which of these features actually have been implemented. The paper only states that it is "the ambition of BEEN is to provide a generic distributed and multi–platform execution framework" [KLM+06] and that "the implementation of BEEN is in beta stage, currently capable of handling a comparison analysis of a nontrivial distributed benchmark" [KLM+06].

12.3 Reproducible Evaluations as a Service

In this section, we outline six requirements an EaaS-system needs to fulfill in order to provide automated and reproducible benchmarks. These requirements are derived from [Jai91] and [KLK20].

Clarity. Defining meaningful benchmarking campaigns is a very crucial task. Mistakes in the definition of the benchmark can lead to wrong or misleading results [Jai91]. A benchmarking framework therefore needs to assist the user in defining benchmarks. This includes providing an easy-to-use interface presenting a clear overview of the parameters

and the resulting configuration of the benchmark. The clarity-requirement therefore encompasses the transparency and comprehensibility of an evaluation framework. One can distinguish between the *semantic clarity* and the *visual clarity*. The semantic clarity assesses the conceptual and functional transparency of the evaluation framework (i.e., that there is no discrepancy between the expected and the actual behavior). The visual clarity assesses the look and feel of the user interface. This includes efficiently fulfilling the user's need for information. As shown in [OST20], visual clarity has a direct impact on the usability.

Flexibility. The required set and type of parameters varies between different systems and benchmarks. In order to support a wide range of systems and benchmarks, a broad set of parameter types is required. Furthermore, the logic for creating benchmarking configurations should be adjustable by the user. This is necessary to support complex use cases and scenarios. An improper modeling or representation of the system under evaluation or the benchmark within the benchmarking system can produce miss leading results due to an inappropriate experimental design [Jai91].

Reproducibility. The reproducibility of results is a subject of high importance in the research community [Hu20; CHI15; SYF+20; FBS12]. It describes the ability to repeat the benchmark in another but identical environment and to obtain the same results [KLK20]. A benchmarking system should assist, and where possible also force, the user (i.e., a researcher or software developer) in making their results as reproducible as possible. This includes storing all required information for repeating an evaluation and providing proper methods for storing all necessary environment parameters (e.g., software version numbers and hardware information) required to recreate the environment [KLK20]. This serves multiple purposes: First, it fosters the reproducibility of evaluations. Second, it allows to identify differences to previous executions of the same benchmark which avoids wrong conclusions [Fei03]. And third, it allows the verification of the results by other researchers without completely redoing the evaluation.

Scalability. Even simple benchmarks can quickly result in a large number of configurations that need to be benchmarked [Jai91]. Especially when considering multiple parameters at the same time, testing all combinations can easily result in hundreds of individual configurations. When every configuration is then also executed multiple times in order to get reliable results [Jai91], this can quickly result in a massive amount of required executions—and that's only for one specific investigation. A benchmarking system needs to be able to efficiently and reliably deal with this. Individually executing the benchmark for all resulting parameter configurations can take a great amount of time. However, the individual configurations can perfectly be executed on several identical machines

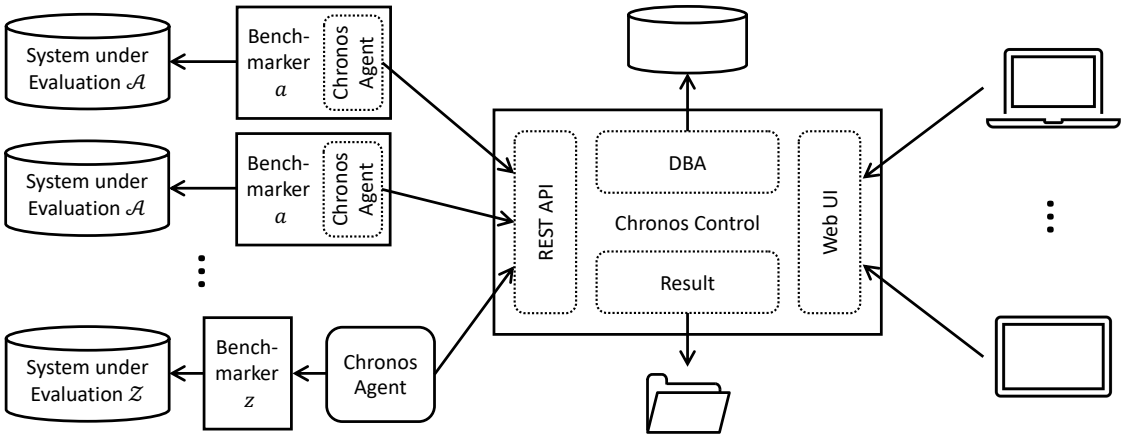


Figure 12.1 Overview of the evaluation framework Chronos (based on [VSC+20]).

in parallel. Supporting such a parallelized execution is a requirement a benchmarking system needs to fulfill in order to handle larger benchmarking campaigns in acceptable time frames. The evaluation system needs to ensure that all parts of one evaluation are executed on machines with an identical hardware and software configuration [KBT04].

Monitorability. Since evaluation campaigns can take a long time, identifying potential issues as soon as possible is essential. The evaluation system needs to provide functionality for monitoring the execution and observing the progress [KBT04]. Reducing the need for direct access to the machines running the benchmarks also avoids intended and unintended interference of the user and therefore further fosters reproducibility.

Transparency. The evaluation system must not impact the results of the benchmark. Background tasks required by the evaluation system running on the machines executing the benchmarks need to be as light wight as possible. If there are multiple benchmarks executed in parallel, the results need to be independent of the number of benchmarks running in parallel [BLW19].

12.4 The Chronos System

Driven by the need for a reliable solution for benchmarking Polypheny-DB, and due to the lack of such a system, we created *Chronos*. It fulfills the requirements outlined in Section 12.3. While our primary use case is the benchmarking of database systems, the Chronos evaluation framework is nevertheless extremely generic and can be used for benchmarking other types of systems as well.

As depicted in Figure 12.1, the Chronos evaluation framework consists of two building blocks: *Chronos Control* and *Chronos Agent*. The main software component is Chronos Control, the heart of the evaluation framework. It provides a browser-based user interface for managing evaluations and a REST API for interacting with the Chronos Agents. It is also responsible for visualizing and archiving the results.

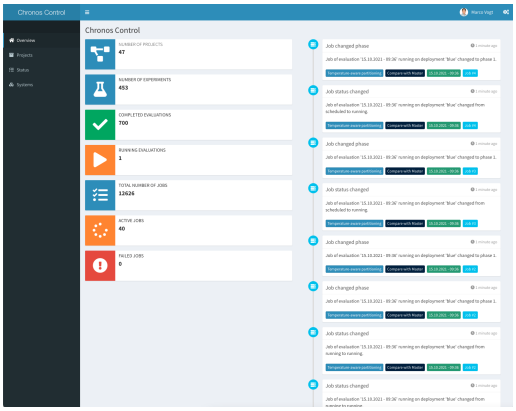
The Chronos Agents steer and monitor the execution of the benchmarks. They request the specifications and parameters for a benchmarking task from Chronos Control. The actual benchmark (i.e., executing the workload, collecting metrics etc.) is done by a *benchmarker*. As shown in Figure 12.1, the agent can either be part of the benchmarker itself or is deployed as a separated piece of software that triggers and interacts with the benchmarking software (e.g., YCSB [CST+10]).

As depicted in Figure 12.1, there can be multiple instances of a Chronos Agent requesting tasks from Chronos Control. This allows to parallelize the execution of a benchmark (see Section 12.6.4). An agent can support multiple benchmarks and systems under evaluation. This allows using a pool of benchmarking machines for multiple projects.

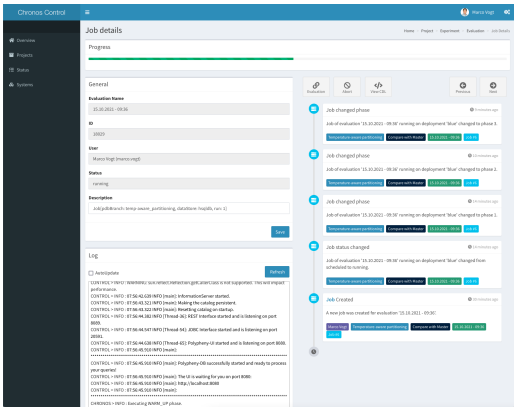
All communication between the agents and Chronos Control is implemented in a polling-fashion. This has been a conscious design decision which massively simplifies the deployment of larger setups. Especially in scenarios where clients are deployed at multiple locations (e.g., on internal machines and in the cloud), the necessary firewall configuration for a reversed communication approach would certainly violate network security guidelines.

The full benchmarking results and the log output of every benchmarking job is stored at a central location. Chronos Control includes the necessary logic for mounting external storage systems for this purpose. The data and configuration of Chronos Control itself (user accounts, projects, etc.) are stored in a dedicated database system.

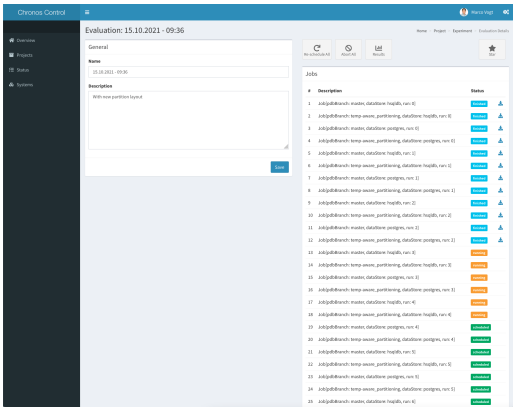
The browser-based user interface allows an easy interaction with Chronos Control. The user interface allows creating and managing experiments and evaluations, monitoring their execution, and analyzing their results. It also allows adding new benchmarks or systems under evaluation and to define their parameters. Since the user interface is browser-based, it can be accessed using different devices (e.g., computers, tablets, smartphones) and by multiple users at the same time. As depicted in Figure 12.2(a) the start-page gives an overview on the current status of all evaluations. This allows to quickly check if everything is fine.



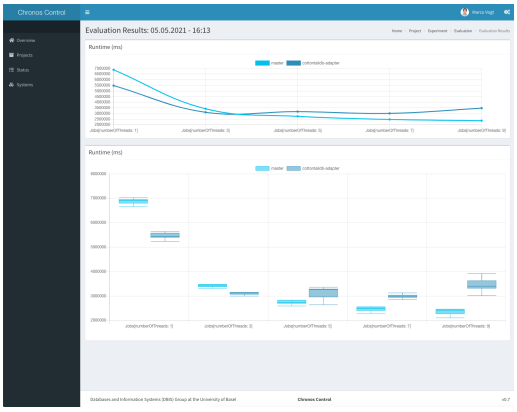
(a) The overview page summarizes the status of the user's evaluations.



(b) Details on the status and progress of a running job.



(c) Status of the individual jobs of an evaluation.



(d) Results of an evaluation rendered by Chronos.

Figure 12.2 Screenshots of the user interface of Chronos Control.

Chronos supports benchmarking multiple systems using different benchmarks at the same time. In combination with its role-based user management, Chronos is suitable for larger groups and even entire departments. This allows for a more efficient use of resources, since Chronos can schedule the execution of jobs across all available machines.

A consequence of such a setup is a heterogeneous pool of machines for executing the benchmarks. For benchmarks that produce metrics that are influenced by the hardware and software configuration (e.g., transactions per second) it must be ensured that all benchmarks are executed on identical machines. In Chronos, this is implemented by the concept of execution *environments*. Environments are user-defined identifiers for a pool of machines with identical hardware and software configuration. Evaluations which require the availability of certain hardware or software can also be handled by using corresponding environments.

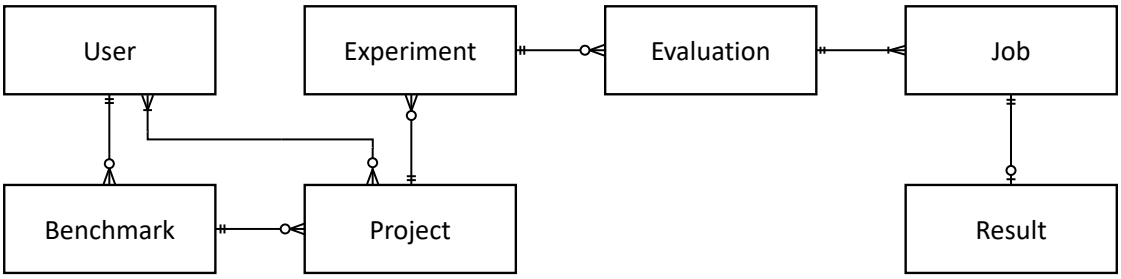


Figure 12.3 Simplified data model of Chronos (Crow's foot notation).

The log-output from the benchmarkers and the systems under evaluation can be forwarded to Chronos Control by the Chronos Agents. The agents also submit the progress. As depicted in Figure 12.2(b), this allows to monitor the execution and to abort or reschedule specific jobs or whole evaluations directly in Chronos Control. Furthermore, it is also possible to do a basic analysis of the results directly within Chronos Control.

Benchmarks or systems under evaluation can be added in the user interface of Chronos Control. Similar to the graphical tool for building result configurations, there is also a graphical interface for defining the *parameters* of a benchmark.

12.5 Implementation

As illustrated in Figure 12.1, the Chronos stack consists of two building blocks: *Chronos Control* and *Chronos Agent*. This section introduces the common data model and gives an overview on the implementation of the two building blocks.

12.5.1 Data Model

Chronos was built with the goal of being as versatile as possible. The idea was not to build a benchmarking solution for one specific project, but a generic solution for all kinds of benchmarking projects. This versatility is reflected in the data model depicted in Figure 12.3.

Project. Projects allow to organize benchmarking campaigns by grouping *experiments*. A project consists of an arbitrary number of experiments. Furthermore, projects are also used to assign and manage privileges. Every project can have an arbitrary number of members. Every member (i.e. *user*) of a project has access to all *experiments*, *evaluations*, *jobs*, and *results* associated with this project.

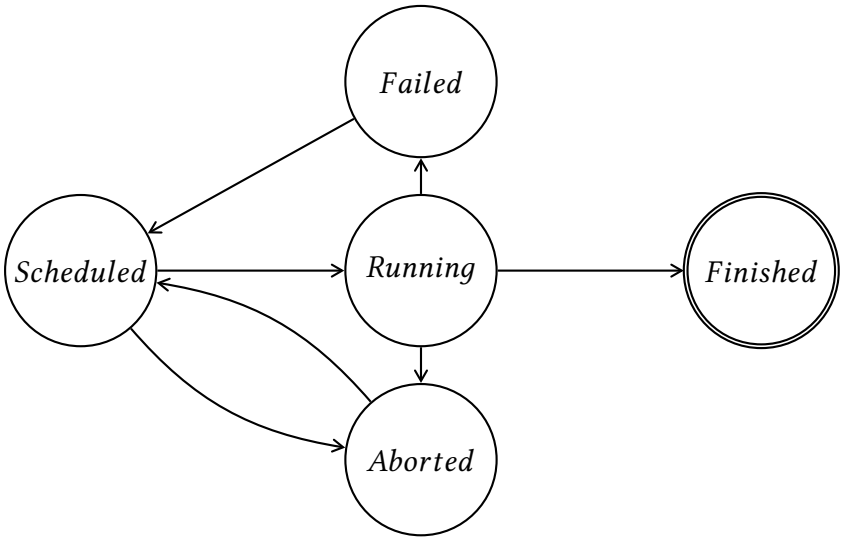


Figure 12.4 State diagram of a Chronos job depicting the possible transitions between the states.

Experiment. An experiment is the definition of an *evaluation*. Once created, experiments are immutable. Every experiment is part of a *project*. There can be multiple executions (i.e. *evaluations*) of an experiment. This allows to easily repeat an experiment and to investigate the effect of changes to the system under evaluation.

Evaluation. An evaluation is an execution of an *experiment* in a specific *environment*. All evaluations derived from an experiment have the same specification. An evaluation consists of at least one *job*. If the objective of the benchmark is to investigate the effect of a certain parameter (e.g., the number of threads), there are r jobs per configuration of this parameter with r denoting the *number of repetitions* per configuration. The total number of jobs of an experiment is therefore given by $\prod_{i=0}^{p} n_i r$ with p denoting the number of varied parameters and n_i the number of configurations of the ith varied parameter. The execution of jobs is parallelized within the selected environment.

Job. A job represents a specific *parameter configuration* for the benchmark and the system under evaluation. Jobs can be in five different states. The states and the possible transitions between these states are depicted in Figure 12.4. Every finished job has a *result* assigned.

Result. The result of a *job* consists of a JSON document and a ZIP file. The JSON document contains the results which can be analyzed and rendered by Chronos Control. How results are being rendered is defined by *result configurations*. Supplemental data submitted by the agent is stored in the ZIP file (e.g., for further analysis outside Chronos Control).

Benchmark. The representation of a benchmark for one or multiple systems under evaluation. It includes the available parameters and configuration options for creating *experiments*. Every benchmark has a unique identifier. The Chronos Agents submit a list of supported benchmarks. Due to these identifiers, Chronos Agents can be dynamically deployed and removed. For every benchmark, an arbitrary number of execution *environments* can be defined.

Parameter Configuration. Chronos supports a large set of parameter types. Primarily, it can be distinguished between two types of parameters: *static parameters* and *varied parameters*. Static parameters allow for specifying a value. This value is then used for all configurations (i.e. jobs). In contrast, varied parameters allow specifying multiple values or ranges. The available set of parameters can be configured individually for every *benchmark* using a graphical interface in Chronos Control.

Result Configuration. How the results of an evaluation are rendered is defined by result configurations. There can be an arbitrary number of result configurations for every *benchmark*. New result configurations can be added using a graphical interface directly within Chronos Control.

Environment. Environments define groups of machines with similar or identical software and hardware configurations. This allows to parallelize the execution of benchmarks while keeping the results comparable. Environments can be defined individually for every benchmark.

12.5.2 Chronos Control

Chronos Control is an application implemented in PHP offering a browser-based user interface and a REST API for interacting with the Chronos Agents. With the Apache HTTP server[1], PHP[2], MariaDB[3] and git[4], Chronos Control has only few runtime requirements.

The implementation follows the *Model-View-Controller (MVC)* software design pattern. MVC has first been introduced for Smalltalk'80 [KP88]. It is a popular design pattern for developing web applications. The idea is to separate the application into three main logical components: the model, the view, and the controller.

[1] https://httpd.apache.org/
[2] https://secure.php.net/
[3] https://mariadb.org/
[4] https://git-scm.com/

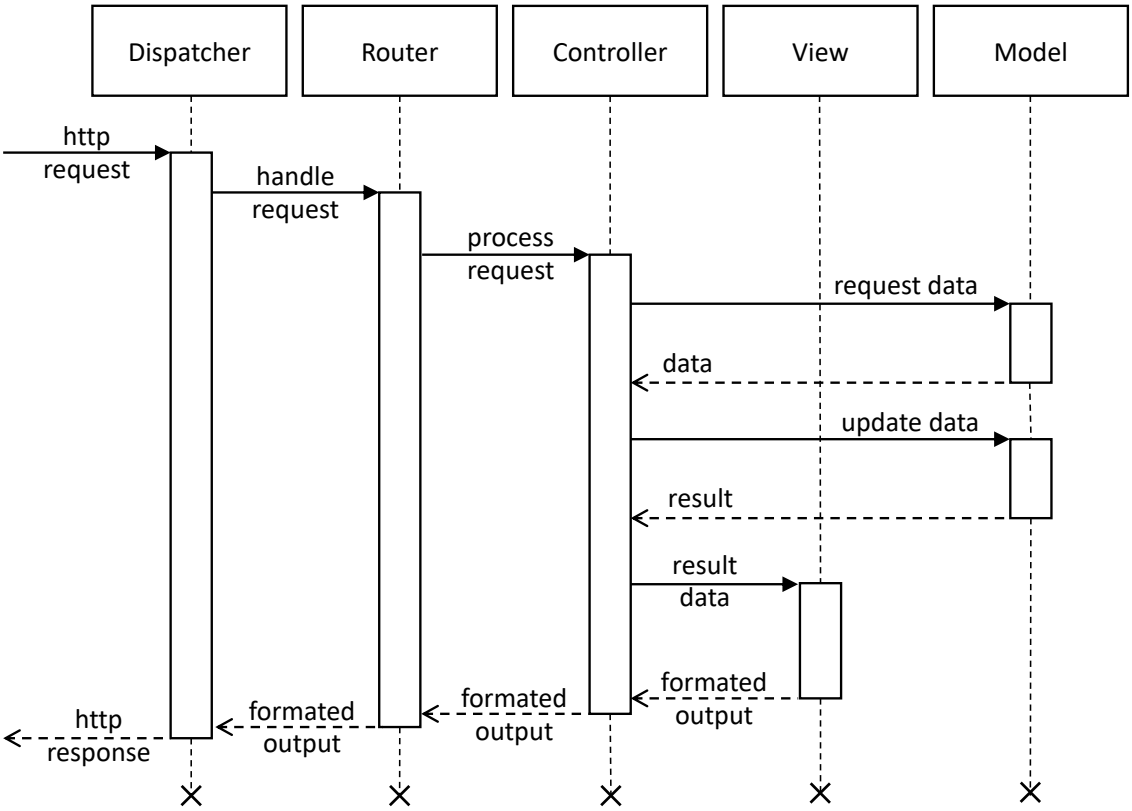


Figure 12.5 Sequence diagram of the MVC implementation (without object instantiations).

The *model* is responsible for managing the data. This includes validation rules and the logic for atomic operations (e.g., creating a user account). The data handled by the model can be stored in an external data storage (e.g. a database system) that also provides persistency.

The *view* takes care of representing the data. In the context of web applications, this means generating an HTML page for the user interface or a JSON response for the API. The view does not contain any application logic.

The *controller* contains the application logic. It accepts input values, processes them using data obtained from the model, and composes the result data to be rendered by the view. The controller also initiates updates of the data through the models.

Usually, there are multiple models, views and controllers in an application. However, there are various adaptions of the MVC model. Figure 12.5 shows a sequence diagram of the MVC implementation in Chronos Control. A request is accepted by the dispatcher and handled by the router. The router is the central component that decides which controller and view needs to deal with this request. While processing the request, the controller might request and update data through multiple models. Finally, the output is

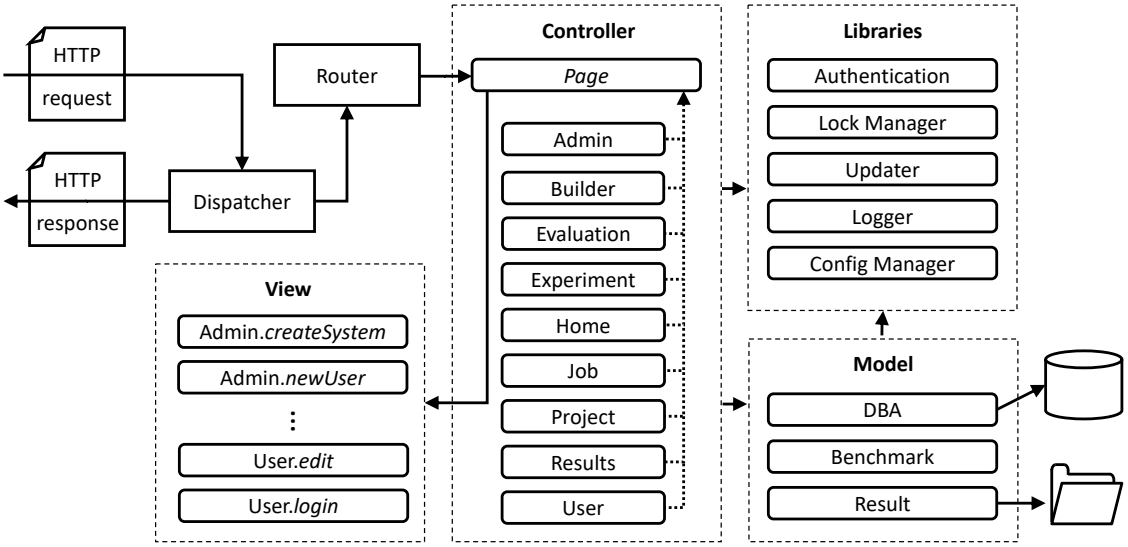


Figure 12.6 Software architecture of Chronos Control.

rendered by the selected view. Which view is being selected depends on the action and the interface through which the request has been received.

Chronos Control features two interfaces: a browser-based user interface and a REST API. The user interface is rendered using Bootstrap[5] and the AdminLTE[6] template.

Figure 12.6 depicts the architecture of Chronos Control. In addition to the model, view and controller components, there is also a library component containing logic used by several controllers like authentication handling and logging.

The system is implemented using object-oriented PHP. The object instantiation is done by the *router* based on the requested URL using a custom class loader. This approach offers a high degree of modularity and expandability. This flexibility also facilitates providing custom implementations for additional parameter types and result graphs for individual systems by the user. It even allows overwriting the whole logic for defining experiments, deriving evaluations and generating jobs. The necessary PHP scripts can be provided using a git repository by specifying URL and credentials in the UI. There can be one repository per benchmark. However, executing user-provided code is a severe security risk. This function is therefore disabled by default and needs to be activated in the configurations file first. Hence, activating this function requires the same level of access that a malicious usage of this functionality can provide. Furthermore, modifying the repository related settings of an adapter is limited to users belonging to the administrators group—even when this feature has been enabled.

[5] https://getbootstrap.com/
[6] https://adminlte.io/

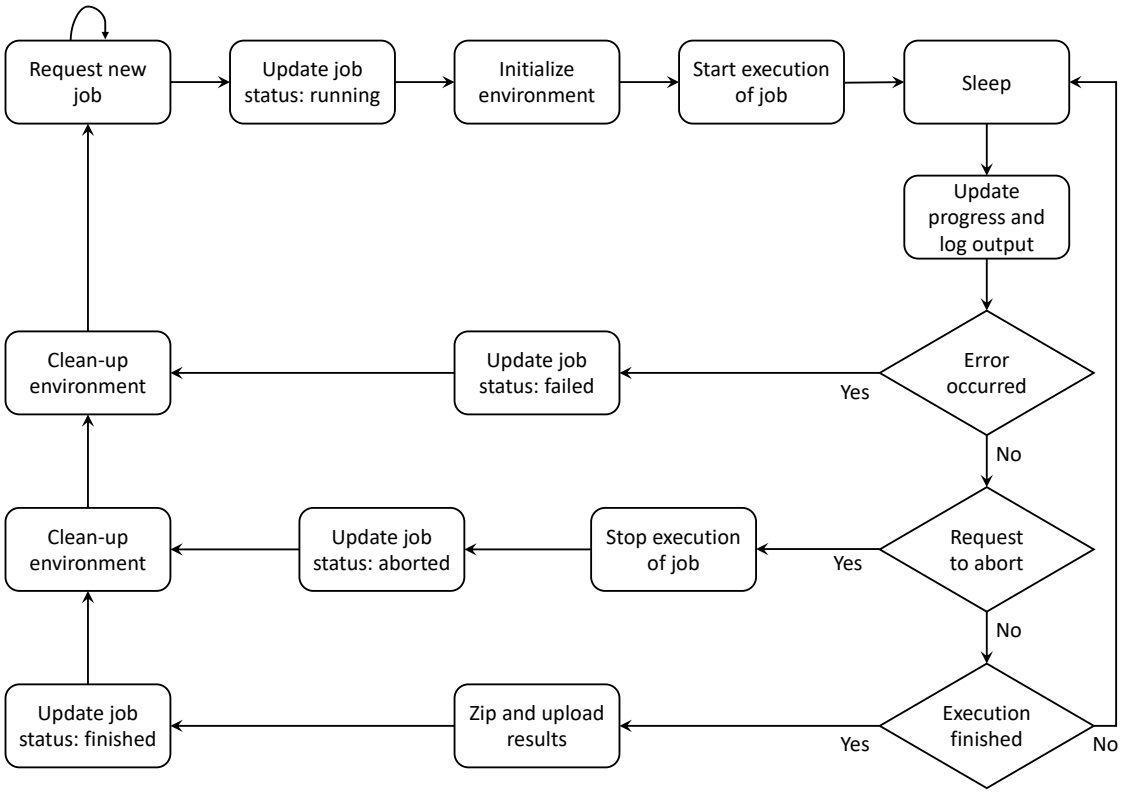


Figure 12.7 Flow Chart visualizing the control loop of Chronos Agent.

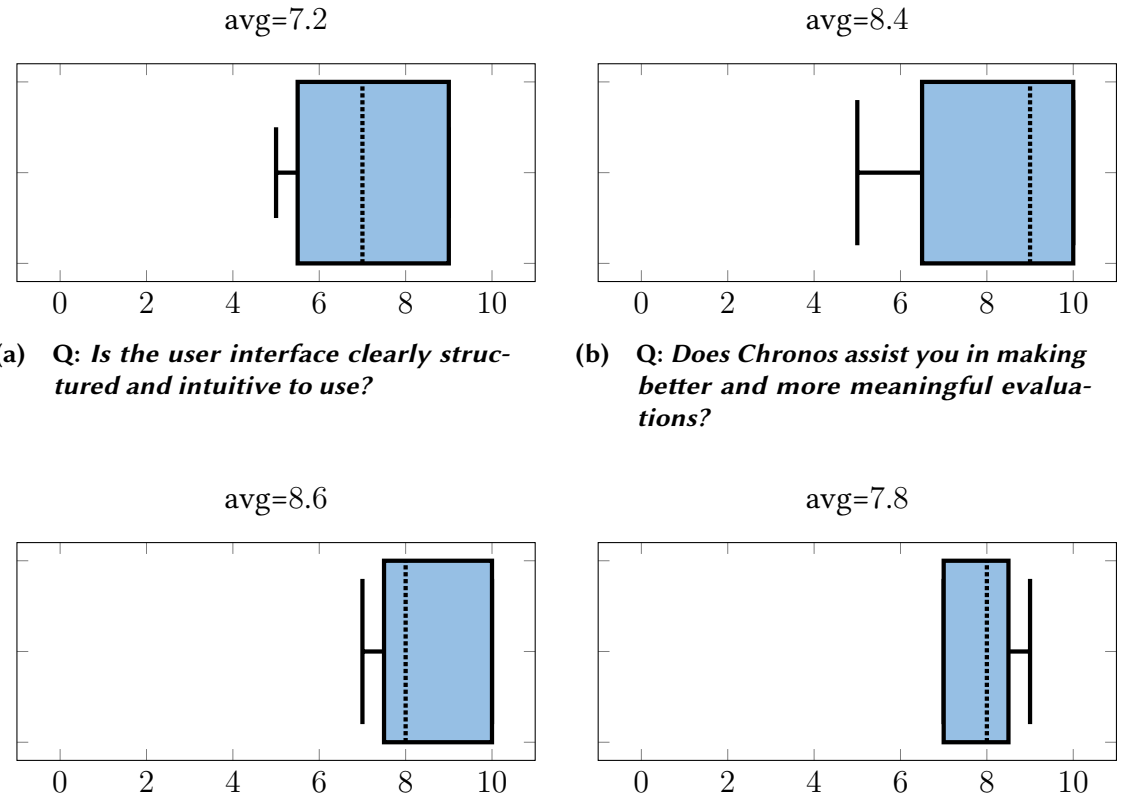
Chronos Control includes an installation script which automates the setup. There is also an integrated mechanism for upgrading Chronos Control to the latest version. Chronos Control is released under the MIT open-source license. The source code and releases are available on GitHub[7].

12.5.3 Chronos Agent

A Chronos Agent is the component of the Chronos stack that runs on every machine used for executing evaluations. It is either integrated into the benchmarker—the software executing the workload and collecting the metrics—or deployed standalone to call existing benchmarking applications. It handles the communication with Chronos Control and steers and monitors the execution of benchmarking jobs.

The flow chart in Figure 12.7 depicts the generic control loop of a Chronos Agent. Implementing a Chronos Agent requires implementing this control loop. Since Chronos Control exposes a documented REST API for communicating with the clients and uses JSON for encoding the messages, implementation should easily be possible in nearly every programming language and on nearly every platform.

[7] `https://github.com/Chronos-EaaS/Chronos-Control`



(a) Q: ***Is the user interface clearly structured and intuitive to use?***

(b) Q: ***Does Chronos assist you in making better and more meaningful evaluations?***

(c) Q: ***Is the behavior of Chronos always transparent to you?***

(d) Q: ***How would you rate the overall usability of Chronos?***

Figure 12.8 Results of a survey among users of Chronos (n=5). Every question has been answered on a scale from 0 (don't agree) to 10 (fully agree).

We created a reference implementation of a Chronos Agent for JVM-based languages, available through Maven Central[8]. This implementation also contains functionality for measuring execution times. The source code is released under the MIT open-source license and is available on GitHub[9].

12.6 Evaluation

In this section, we perform an evaluation of Chronos. As criteria for this evaluation, we take the requirements outlined in Section 12.3. This section is structured accordingly.

12.6.1 Clarity

As outlined in Section 12.3, it can be distinguished between two types of clarity: the *semantic clarity* and the *visual clarity*. In Chronos, both types have been addressed in multiple ways. With eleven distinct parameter types, the semantics of parameters can precisely be depicted. The ability to provide human-readable names for all parameters and options improves the semantic clarity. The event timeline depicted in Figure 12.2(a) and Figure 12.2(b) provide a context-dependent overview of all ongoing activities. This enhances semantic clarity (by providing insights and improving transparency) and visual clarity (by serving the information need in an easy-to-understand form).

Chronos allows to add user-defined descriptions to projects, experiments, evaluations and jobs. This allows to add arbitrary information for further reference. The job name is automatically generated and contains the values of all varied parameters (see Figure 12.2(c)). The design of the UI is based on Bootstrap, providing enhanced visual clarity and accessibility. Pictograms are used in accordance to ISO 9241-210 for improved comprehensibility. The separation between experiments and projects eliminates human errors when repeating an experiment.

To assess the clarity of Chronos, we performed a user-study among a group of developers that used Chronos in the past. We made sure to select a diverse set of participants with different levels of experience in using Chronos. As argued by [NL93] five participants are sufficient for this type of user study. For assessing the clarity-requirement, we have asked four questions. Each question has been answered on a scale from 0 (don't agree) to 10 (fully agree). The results of the survey, including the exact questions, are depicted in Figure 12.8.

The results of the survey show that the behavior of Chronos is transparent to the users. This is very important since intransparency and unexpected behavior can lead to mistakes in the definition of the experiments. Most users also agree that Chronos helps them in defining better and more meaningful benchmarks. However, the survey also shows that the user interface needs further refinements. Nevertheless, with rating of 7.2 it is assessed acceptable for the sake of this evaluation.

[8] `https://search.maven.org/artifact/org.chronos-eaas/chronos-agent`
[9] `https://github.com/Chronos-EaaS/Chronos-Agent`

12.6.2 Flexibility

Chronos is a very versatile tool. We intentionally developed it as a generic benchmarking framework and not as a benchmarking tool purely tailored to Polypheny-DB. The support for eleven distinct parameter types and five result graph types, allows modeling a broad set of benchmarking tools and systems under evaluation.

If the available set of parameter types or result graphs is not sufficient, it is possible to provide additional types. The implementation for these additional parameter types can be provided individually by specifying a git repo containing PHP scripts. This can be done completely within the user interface of Chronos Control. It is also possible to extend or completely override the logic for creating experiments. This allows to model highly sophisticated workflows and complex benchmarking scenarios.

Another aspect of flexibility is that of *integration*. The decoupling of Chronos Control and Chronos Agent minimizes the language and platform dependent parts of the software stack. Integrating support for benchmarking with Chronos into existing benchmarking clients has been simplified by locating the complex logic in Chronos Control. Furthermore, the XML-based job definition and the REST API used for communication between Chronos Control and the Chronos Agents allows for straightforward implementation of Chronos Agents on nearly all platforms and using nearly all programming languages.

Chronos has demonstrated its flexibility as a benchmarking tool for a broad spectrum of projects. Beside Polypheny, Chronos has been used for benchmarking the multimedia retrieval database system ADAMpro [Gia18], the distributed multi-store system Icarus [VSS17], the data replication and partitioning cost model BEOWULF [SFS16], and TDG, a temporal data generation tool for PolarDBMS [BS14]. Furthermore, Chronos has been used in the context of numerous student projects.

12.6.3 Reproducibility

Chronos assists developers in making their results more reproducible in two ways: First, by automating the entire evaluation workflow, Chronos removes the human from the loop and forces all steps to be scripted and second, it archives all parameters, configurations and logs of the evaluation. Our reference implementation of the Chronos Agent also archives various environment parameters.

Another factor that improves the reproducibility is an indirect one: By significantly simplifying the benchmarking process and therefore eliminating or at least drastically

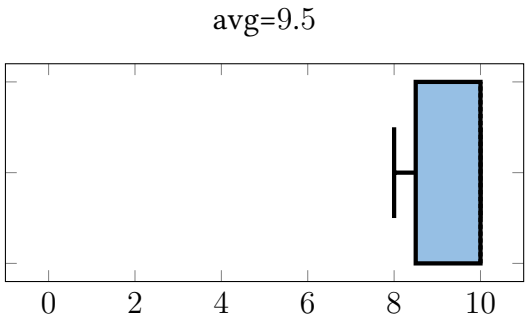


Figure 12.9 Q: *Does Chronos make your evaluations more reproducible?*

reducing the potential for human error's in the benchmarking configuration, the overall reproducibility gets improved.

Chronos does not allow the deletion of evaluations or their results, but it is possible to archive them. This avoids the system getting cluttered, while at the same time, preserving all data that might be important for future investigations. However, Chronos is only a framework for benchmarks. The achievable degree of reproducibility therefore heavily depends on how it is applied.

As part of the user study introduced in Section 12.6.1, we also asked the participants whether Chronos makes their evaluations more reproducible. The results for this question are depicted in Figure 12.9. With an average of 9.5 out of 10, the participants of the survey clearly agree that Chronos is making their evaluations more reproducible.

12.6.4 Scalability

Chronos supports executing evaluations with an arbitrary number of jobs and for an arbitrary number of systems at the same time. There are no artificial limitations regarding the number of users, benchmarks and systems under evaluation. However, there are of course technical limitation depending on the capabilities of the hardware on which Chronos Control is deployed.

Chronos features a role-based rights management using projects. This allows to use one central instance of Chronos for whole groups or departments. Such a setup has proven reliable over several years at the DBIS research group at the University of Basel.

The approach of dividing an evaluation into individual jobs allows parallel execution and therefore a high efficiency. With its distributed architecture, Chronos is tailored towards a distributed and parallelized execution of benchmarks. This approach is also

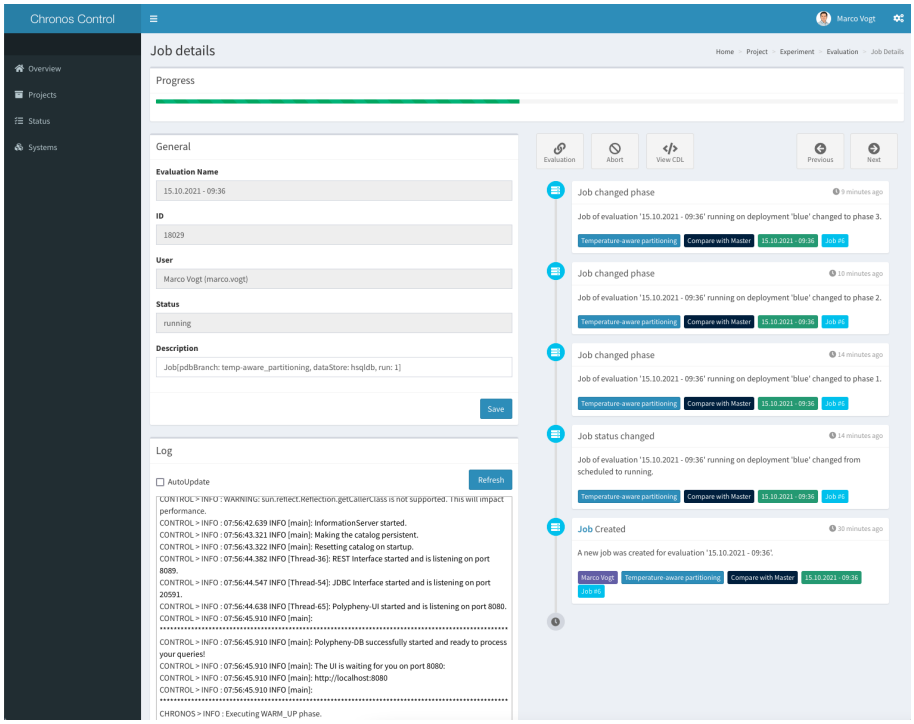


Figure 12.10 Details on the status and progress of a running evaluation job.

very reliable since the parameters and configuration for a job remain persistent and can be verified later.

There are no artificial limitations regarding the number of jobs that can be executed at the same time. Due to the architecture of the Chronos stack, the resources required on Chronos Control for each simultaneous job are very low.

Execution environments allow defining groups of machines with an identical hardware and software configuration. An arbitrary number of environments can be defined for every benchmark and system under evaluation. Jobs belonging to the same evaluation are only executed on machines belonging to the same execution environment.

12.6.5 Monitorability

Chronos allows monitoring the execution of benchmarks within the user interface of Chronos Control. Figure 12.10 shows a screenshot of the detail's page of a job. The progress of the job is calculated by the Chronos Agent and periodically sent to Chronos Control together with other status information.

It is also possible to monitor the log output of the benchmarker live in the user interface of Chronos Control. The log is captured by the Chronos Agent and archived together

with the results of the benchmark. The log view supports ANSI colored log output. This is very useful to quickly identify error messages.

Another powerful feature for monitoring evaluation campaigns are the events displayed as timeline. The timeline displays events associated with the displayed entity (project, experiment, etc.). It also displays the time elapsed since the event. This has proven useful for identifying issues such as jobs stuck in one state.

The user interface of Chronos Control can be accessed using mobile devices (like smartphones or tablets). This allows to conveniently check the progress of ongoing evaluations. This especially comes in handy if there are approaching deadlines and a failing benchmark would require immediate actions.

12.6.6 Transparency

The architecture of Chronos with its separation of Chronos Control and Chronos Agent minimizes impact on the measured performance. Nearly all logic is located in Chronos Control. The Chronos Agent is very lightweight. All compute heavy actions, such as compressing and uploading the result to Chronos Control, happen after the completion of the benchmark. All measurements are done on the machine where the benchmarks are executed. Parallel benchmarks or a high load on Chronos Control therefore has no impact on the measurements. However, it is assumed that Chronos Control is not deployed on a machine used for executing benchmarks.

Nevertheless, impact on the performance is still possible. A critical point is the transmission of log-results to Chronos Control. If there is a large amount of log-output, this can impact the performance. However, a large amount of log-output has a severe impact on the performance in general and should thus be avoided anyway.

12.7 Discussion

Evaluations are an important part of software development and research in computer science. For the former, continuous benchmarks are important throughout the whole development process to identify performance regressions and bottlenecks, and to optimize the system's configuration. In research, reproducible results are necessary to compare the performance and efficiency of different approaches. As researchers developing (prototype) systems for verifying and benchmarking our models, meaningful and reproducible benchmarks are crucial.

Benchmarks provide more than numbers, they provide insights that foster our understanding of systems and concepts. However, good benchmarks are a time-consuming endeavor. With Chronos, we created a framework that automates the entire evaluation workflow. Chronos makes results more transparent, more reproducible and less labor-intensive.

Chronos has been implemented due to the lack of available solutions for automated and distributed benchmarking of systems. It fulfills the requirements outlined in Section 12.3, and has proven itself a stable and reliable tool for benchmarking Polypheny. By releasing Chronos under the MIT license, we made this tool available for other project as well.

As part of future work, we want to further improve the visualization of the resulting parameter configurations. Based on the result of the user study, we also plan to improve and simplify the user interface. Furthermore, we want to transform the concept of Chronos Agents towards a universal piece of software that takes care of setup, execution and clean-up of arbitrary evaluation tasks (e.g. using Docker). This step would fully remove the human from the (benchmarking) loop and further improve reproducibility.

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

— Albert Einstein

13

Verification

Software is written to fulfill a specific purpose [KA08]. *Software verification* is the task of checking if the software *correctly* fulfills this purpose. It is an essential part of software development and a prerequisite for all kinds of performance benchmarks. There are two fundamental approaches for verifying the correctness of software: *empirically* and through a *formal proof*.

The empirical approach can be compared with scientific experiments: the software to be tested is called with various input values. For every input value, it is then checked if it results in the expected output value or behavior. However, an empirical approach can only prove that a software is incorrect.

Formally proving the correctness of software is extremely difficult and not possible in all cases [Pre16]. It is an active field of research and there are also some interesting projects like *DeepSpec*[1]. However, there is not yet a practical approach for proving the end-to-end correctness of whole systems. Hence, in this chapter, we focus on empirical methods.

The most important factor affecting empirical testing methods is the choice of the input values with which the software is tested. They should cover as many cases as possible. In a PolyDBMS with its support for multiple query languages, data models and execution engines, selecting "good" input values is very difficult. For verifying Polypheny-DB, we therefore developed the *Polyfier* approach. It verifies the correctness of a PolyDBMS by comparing the result of randomly generated queries for different storage configurations. In this chapter, we introduce the blueprint for this approach and discuss its limitations. Furthermore, we briefly describe how we have implemented it for testing Polypheny-DB.

[1] `https://deepspec.org/`

13.1 Assumptions

Before we introduce the blueprint of our *Polyfier* approach, we first discuss two assumptions we need to make in order for this approach to work as intended.

Assumption 1. Although database systems are usually well-tested software, they—as probably every system—have bugs. This is also the case for the database systems used as underlying data stores. However, we assume that the execution engines (i.e., the underlying data stores and the engine of the PolyDBMS) do not have the same bug. Thus, if we execute the same query on the underlying data stores, we can identify problems with the integrated engine of the PolyDBMS or with the underlying database systems by comparing the results.

Assumption 2. Every operation supported by the PolyDBMS is supported by at least one of the underlying data stores. Thus, the result produced of the integrated engine of the PolyDBMS can be compared with at least one result produced by pushing down the operation to an underlying data store.

The first assumption is required since we cannot rule out the possibility that there are bugs in the underlying data stores. However, it is unlikely that this will affect all databases supported by the PolyDBMS and that it will result in exactly the same output for all execution engines. However, this may be the case if multiple execution engines rely on the same library. Furthermore, there might also be certain issues with the hardware or the operating system that equally effect all database systems. With the Polyfier approach, we will not be able to identify these issues.

The second assumption is more problematic: to find errors, we compare the results of different execution engines. If all engines return the same result, we assume that the result is correct. However, this assumes that there are multiple engines capable of processing a query. If a function or operation is only supported by the integrated engine, but not by any of the underlying data stores, there is nothing to compare against since the function or operation is always processed by the integrated engine of the PolyDBMS.

This is also the case if a query contains a set of operations and functions that is not supported by a single data store. However, since the queries are generated randomly, there will eventually be a query that only contains functions and operations that are supported by a single data store and is thus able to verify the results produced by the integrated engine of the PolyDBMS. In order to increase the odds for this to happen, the randomly generated queries should differ in their complexity (i.e., in the number of operations and functions).

13.2 Polyfier

The idea of the *Polyfier* approach is to verify the correctness of a PolyDBMS using randomly generated schemas, data, storage configurations and queries. We therefore need generators that randomly produce these inputs:

Schema Generator. This generator randomly generates a schema consisting of n namespaces. The integer n is randomly derived from a normal distribution with a mean of 9 and a standard deviation of 5. It is ensured that $n \geq 1$. The data model of each namespace is picked randomly. For each namespace, a random number of schema elements (i.e., tables, columns, collections, defined fields, graph schema nodes and edges) is being created. The mean and standard deviation of the normal distributions used for randomly generating these schema elements depend on the data model. Where applicable, data types are picked randomly. For relational schemas, primary keys, unique constraints and foreign key constraints are randomly defined as well. It is ensured, that the resulting schema is structurally correct (e.g., that foreign key constraints reference a key and that matching data types are being used). Using the *Query Generator*, the schema generator also generates a random number of views and materialized views within each of the namespaces.

Storage Configuration Generator. The storage configuration defines on which of the underlying data stores the data entities are being stored. This encompasses the configuration of the data partitioning and replication. The generator has a static phase and a random phase. In the static phase, it deploys exactly one data store and allocates all data entities to this data store without any data partitioning. In the random phase, it deploys a random number of data stores and also randomly configures the data allocation and partitioning.

Data Generator. This generator produces data that matches the previously defined schema. It also respects the primary, unique, and foreign key constraints defined for relational schemas.

Query Generator. The heart of the Polyfier system that randomly generates queries that match the schema. In our implementation, the queries are generated as logical query plans by randomly picking possible operations and adding them to the query plan. Using this approach, arbitrary single-model, cross-model, and multi-model queries can be generated. The generator does not only generate query plans that retrieve data, but also query plans that modify data. As discussed in Section 13.1, the generator makes

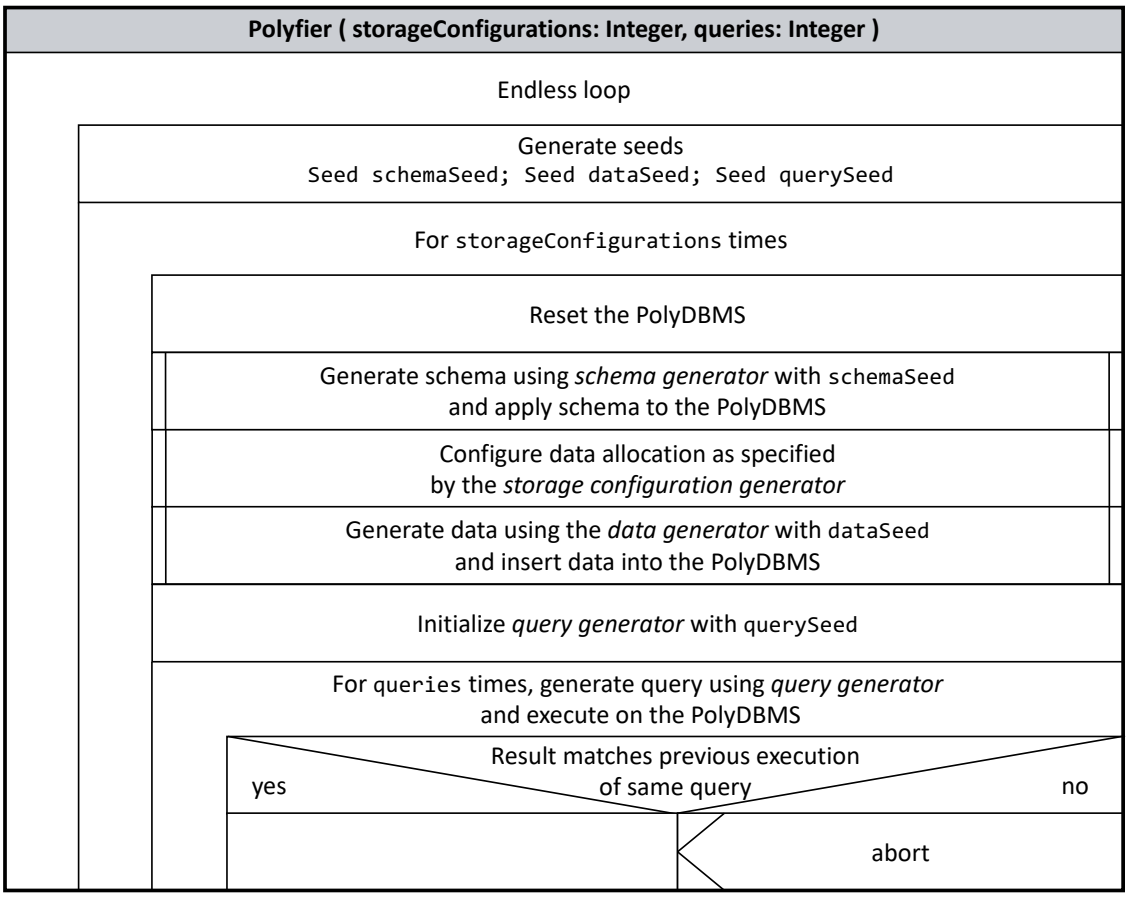


Figure 13.1 Nassi–Shneiderman diagram of the Polyfier approach for verifying the correctness of a PolyDBMS by comparing the result of randomly generated queries for different storage configurations.

sure to produce both small query plans and large query plans (in terms of the number of operations and functions).

Each generator needs to be implemented such that its outcome can be reproduced. In our implementation, we achieve this by using a pseudorandom number generator initialized with a *random seed*. By using the same seed, the outcome can be reproduced.

As depicted in Figure 13.1, the Polyfier system has two parameters: the number of storage configurations that should be verified and the number of queries with which the verification should be performed. Since the generators for the same seed produce the same schema, data, and queries, it is possible to check different storage configurations. At the first execution for a set of seeds, the result of every query is recorded. This allows to compare it with the result of subsequent iterations with a different storage configuration.

Since the first configurations generated by the storage configuration generator allocate all data entities to one data store, and under the assumptions and exceptions outlined in Section 13.1, there should be at least one execution where the result is produced by an

underlying data store. Hence, the result of the integrated execution engine is verified against the result of at least one other execution engine.

As depicted, the Polyfier runs until it finds a problem, i.e., until the result of a query differs from the result of a previous execution of that query under a different storage configuration. By using the same seeds, the situation can conveniently be reproduced in order to identify and solve the issue.

The number of queries with which the verification should be performed primarily depends on the available storage space, since the result of every query needs to be stored. However, a higher number of queries also results in more changes to the data and thus in a better verification of the consistency of data modification operations. Furthermore, the overall stability of the system is tested more extensively.

The number of storage configurations should always be larger than the number of available data stores. If a smaller number is used, only some of the storage configurations allocating all data to a single data store are being checked. At least with Polypheny-DB, this would avoid most of the complex aspects like the query routing and operations in the integrated engine. However, with a high number, it takes more time to thoroughly verify the schema representation and mapping capabilities. Hence, the choice of this parameter is a trade-off between testing schema representation and mapping on one side and query routing and the integrated engine on the other side.

13.3 Discussion

With Polyfier we present an approach for verifying a PolyDBMS. As with any empirical method, it cannot prove the correctness of a PolyDBMS, but it can show with some confidence that errors are rare. Furthermore, it is a valuable tool for resolving issues since the exact conditions can be reproduced.

With the Polyfier approach, we are able to verify multiple aspects of a PolyDBMS. Besides the correctness of the results obtained by executing a query and the consistency of data manipulation operations for various storage configurations, it also verifies the PolyDBMS Requirement 3.3 (*Independence of Storage Configuration*) and the overall stability of the system.

However, Polyfier can only be a piece of a holistic testing a verification concept of a PolyDBMS system. Since our implementation of the Polyfier approach generates query plans based on the internal query representation, the parsing of query languages and

their translation to the internal representation is not tested. In the Polypheny-DB stack, we therefore also use a large set of strategically created integration tests to check the query parsing and translation layer.

Another aspect that cannot fully be verified using the Polyfier approach are cross-model and multi-model queries. These queries contain mapping operations that are always executed in the integrated engine of the PolyDBMS. While the Polyfier can verify the consistency of the results across different storage locations, the actual correctness of the result cannot be verified using this approach. These features therefore need to be thoroughly tested using manually created integration tests.

The extensive verification we have performed has not only allowed us to identify and resolve issues within Polypheny-DB, but also to find and report bugs within the underlying data stores.

Measurements are not to provide numbers but insight.

— Ingrid Bucher

14 Benchmarking

In this chapter, we present the results of the quantitative evaluation of Polypheny-DB, our implementation of a PolyDBMS based on the concepts introduced in this thesis. The goal of these benchmarks is to demonstrate both the effectiveness and the limitations of the introduced concepts based on their implementation in Polypheny-DB.

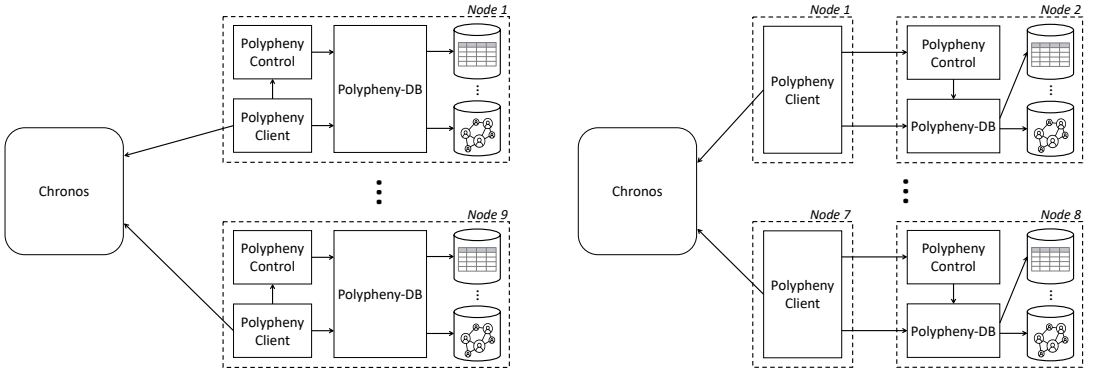
We are committed to the traceability and reproducibility of scientific results. This has not only driven the development of the Chronos system presented in Chapter 12, but also motivated us to make the Polypheny-DB system and all benchmarking tools presented in this chapter publicly available under an open-source license[1]. Furthermore, we also published the raw results, configuration files, parameters and log-outputs of all experiments presented in this thesis in a dedicated GitHub repository[2].

14.1 The Setup

The benchmarking results we present in this thesis were obtained using a distributed and parallelized benchmarking setup with a high degree of automation. At the heart of this setup is Chronos, our Evaluation-as-a-Service system that automates the entire systems evaluation workflow. For executing the benchmarks, we used nine identical nodes (machines). Each node is equipped with an Intel Xeon X5650 24-core CPU with 24 GiB of RAM (out of which 10 GiB are allocated to Polypheny-DB). All machines run Ubuntu 22.04 LTS (with kernel version 5.15.0-37) and the same patch level. As Java runtime environment, we use OpenJDK version 17.0.3.

[1] https://github.com/polypheny

[2] https://github.com/vogti/PhD-Benchmarks



(a) Local: The benchmarking client is deployed on the same node as the system to be benchmarked.

(b) Remote: The benchmarking client is deployed on a separate machine.

Figure 14.1 The machines on which the benchmarks are being executed are arranged in two configurations referred to as *local* and *remote*.

Section 14.1 depicts the benchmarking setup. For the benchmarks, we have arranged the nodes in two different configurations: In the *local* configuration, the benchmarking client is deployed on the same machine as the system to be benchmarked. In the *remote* configuration, the client is deployed on a separate machine. The network connection between the two nodes in the *remote* configuration has a bandwidth of 1 GBit/s. For our benchmarks, we primarily use the *local* configuration. This avoids measuring the network stack that might introduce fluctuations and could be a bottleneck.

All data stores are deployed as Docker containers. We use Docker version 20.10.17, PostgreSQL version 13.2, MongoDB version 4.4.14, and Neo4j community edition version 4.4. The benchmarks are executed by the *Polypheny Client*[3], our versatile benchmarking client for Polypheny-DB. It includes support for multiple industry-standard and custom benchmarks. This is achieved by utilizing OLTPbench [DPC+13], a database benchmarking framework including support for a large variety of database benchmarks. Polypheny Client acts as a Chronos Agent and requests jobs from Chronos Control. Besides benchmarking Polypheny-DB, the Polypheny Client also supports benchmarking other databases. This allows to perform overhead and comparison benchmarks.

To automate the setup and configuration process, we have developed *Polypheny Control*[4], a tool for deploying and monitoring Polypheny-DB. It takes care of pulling the required repositories, executing the build process, and starting Polypheny-DB. This allows us to compare different versions of Polypheny-DB and enables us to perform even complex evaluation scenarios without any manual interaction. Furthermore, it ensures a complete reset between each benchmark.

[3] `https://github.com/polypheny/Polypheny-Simple-Client`
[4] `https://github.com/polypheny/Polypheny-Control`

14.2 The Benchmarks

The results presented in this chapter are obtained using two industry-standard benchmarks and one custom benchmark. In what follows, we briefly introduce the individual benchmarks:

Gavel. The Gavel MultiBench benchmark is a custom benchmark that simulates the workload of the fictive auction house introduced in Chapter 2. It is an extended version of the benchmark we have introduced in [VSS17]. The benchmark creates a relational, a document, and a graph schema and loads randomly generated data. The queries are expressed using SQL, MongoQL, and openCypher and are submitted through different query interfaces.

TPC-C. This is a well-known online transaction processing benchmark that models a supplier, that manages and sells items. The TPC-C benchmark executes a mix of rather complex read and write queries. The specifications for the TPC-C benchmark can be found in [Cou10]. The queries are expressed using SQL.

YCSB. The *Yahoo! Cloud Serving Benchmark* is a benchmark created for evaluating the performance of key-value and cloud serving systems [CST+10]. It consists of five rather simple queries. The benchmark comes with multiple pre-defined workloads that define ratios for these queries. The queries are expressed using SQL.

14.3 The Metrics

In the experiments presented in this chapter, we use two different metrics. The choice of metric depends on the setup of the experiment.

Throughput. Expressed in *transactions per second*, this metric measures the number of transactions (i.e., queries), the database system on average processes per second. A higher throughput can be achieved by a higher degree of parallelization or by reducing the processing time for a transaction.

Latency. The average time an individual transaction takes. It is measured by the benchmarking client, recording the time from the moment a query is submitted and until the result is fully received; thus all result tuples have been read by the client. This metric therefore describes the experience an individual client has.

The results presented in this chapter have been obtained by executing every benchmark 30 times. The presented numbers are the average over these 30 executions.

14.4 Experiment 1: Overhead of Polypheny-DB

A PolyDBMS enables applications that are not possible with single, domain-specific database systems. However, there are also applications where not only these additional features and capabilities are important, but also the performance of the database system; especially the query latency.

Due to its architecture, a PolyDBMS adds an overhead on every query and thus increases the query latency compared to a domain-specific database system optimized for this particular type of workload.

The query latency is influenced by various factors such as

- the complexity of the queries in the workload,
- the optimization of the database for this workload,
- the amount of data,
- the presence of indexes,
- the number of concurrent threads, and
- the concurrency control technique.

Discussing relative execution times or throughputs does therefore not provide any insights. Selecting a data store with an optimistic concurrency control technique, loading a small amount of data, and executing simple queries with a high degree of concurrency might result in execution times being significantly increased by the PolyDBMS. However, if we compare Polypheny-DB with a database system that, like Polypheny-DB, features a strong strict two-phase locking protocol, and choose a benchmark with a large amount of data and complex analytical queries, the absolute execution time will only differ within fractions of a percent.

A meaningful assessment of the overhead introduced by the PolyDBMS therefore requires determining the absolute overhead. The overhead can be seen as the quantification of the impact of using Polypheny-DB if there is no additional benefit for this scenario, thus no capability or feature added by Polypheny-DB.

For this experiment, we therefore select a database that natively supports all queries of a benchmark. By comparing the query latency obtained by directly executing the queries on this database with the query latency obtained by benchmarking Polypheny-DB using this database as its single data store, we are able to isolate the overhead. A serial execution of the workload avoids measuring concurrency control specific differences.

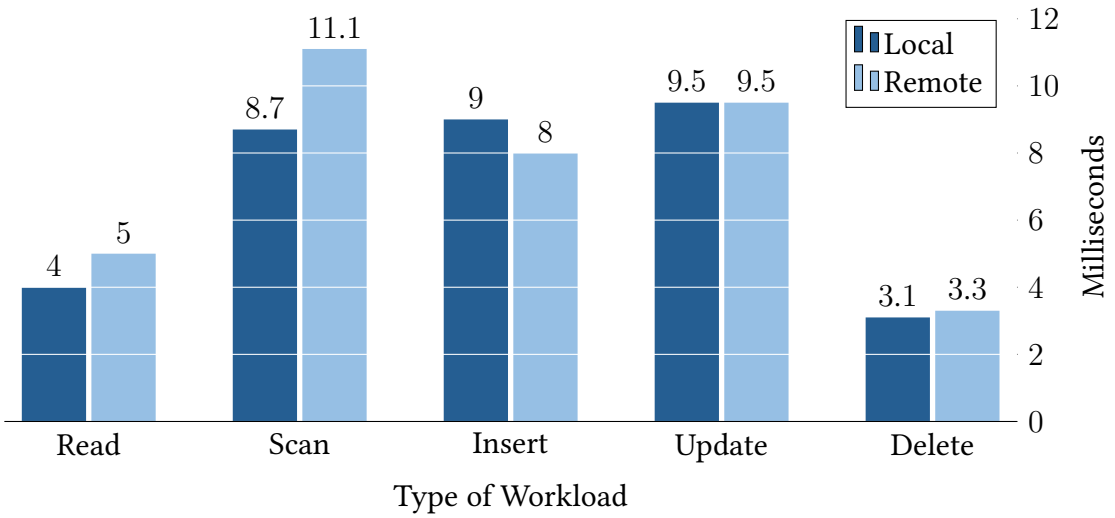


Figure 14.2 Overhead for different types of workloads, measured using the YCSB benchmark. The depicted overhead is the difference between the arithmetic means of the query latencies on Polypheny-DB with a PostgreSQL data store and directly on PostgreSQL.

Due to its popularity, we selected PostgreSQL as database system for this experiment. Furthermore, we selected the YCSB benchmark since it is a well-known and industry-standard benchmark and allows differentiating between different types of workloads. The results of this experiment are depicted in Figure 14.2. PostgreSQL has in both cases been deployed using Docker. The measurements have been performed both with the client on the same node (local) and with the client on a different node (remote). Furthermore, we have compared the query latency for different amounts of data. The difference between the mean query latencies for all queries with a scale factor of 1 and 100 is only 0.05 ms. It can be safely assumed that these are natural variations in the range of the measurement tolerance. The overhead added by Polypheny-DB is thus independent of the amount of data. As mentioned above, the raw results and configuration details can be found in the results repository[5] on GitHub.

The benchmark shows a high overhead for inserts and updates. This is probably due to validation of type constraints. The difference between the overhead of read queries and the overhead of scan queries is expected since there is a larger ResultSet that needs to be serialized and deserialized twice.

Furthermore, the benchmark shows that the JDBC stack of Polypheny-DB, especially the efficiency of the result serialization, is seemingly less efficient than the one of PostgreSQL. If there is data to be serialized (i.e., read or scan) the overhead is higher in a remote deployment than in a local deployment.

[5] `https://github.com/vogti/PhD-Benchmarks/Experiment_1`

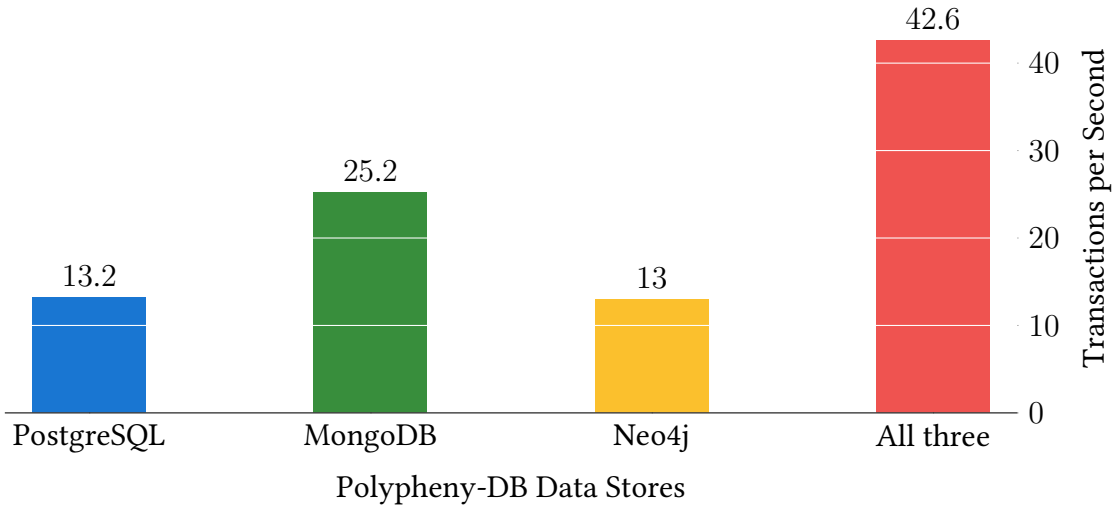


Figure 14.3 Comparison of the throughput in transaction (queries) per second between Polypheny instances with one data store and a Polypheny instance with multiple data stores. For the latter, the data is fully replicated across all data stores. The workload consists of only read queries. Higher numbers are better.

While especially the overhead for inserts and updates could be better, the overall overhead of Polypheny-DB can be considered negligible for many applications. Nevertheless, we consider further reducing the overhead an important part of our future work.

14.5 Experiment 2: Routing

Query routing is an essential part of any PolyDBMS. It plans the decomposition and execution of a query and selects on which of the underlying data stores a query or parts of it should be executed. In Chapter 10, we have introduced a conceptual model for efficiently routing queries in a PolyDBMS. In this experiment, we benchmark this model based on its implementation in Polypheny-DB.

To evaluate the routing model, we compare the throughput of multiple Polypheny-DB instances, each with a single data store, against a Polypheny-DB instance with multiple data stores. Since we explicitly want to benchmark the ability of the system to handle heterogeneous data and workloads, we use the Gavel MultiBench benchmark for this experiment.

This experiment not only benchmarks the effectiveness of the query routing, but also the capabilities of the integrated engine and the mapping between data models for both data retrieval and data modification queries.

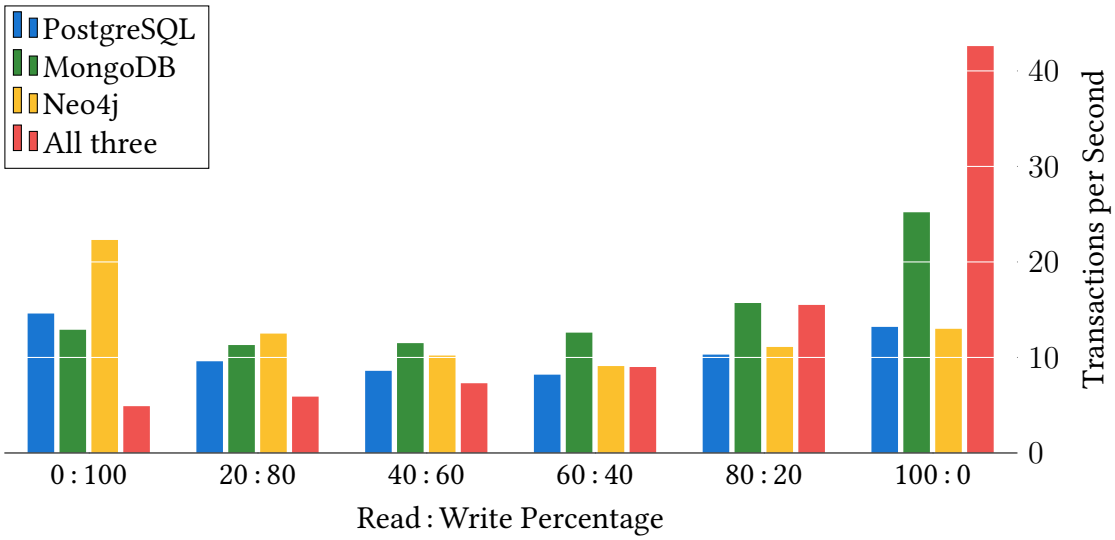


Figure 14.4 Results for different read/write ratios. In the "All three" configuration (red bars), the data is fully replicated across all three data stores. Higher numbers are better.

Figure 14.3 depicts the results of the Gavel benchmark for a workload consisting only of read queries. The benchmark has been executed in a local deployment with four concurrent workers. The first three bars depict the results for a Polypheny-DB instance with a single data store. The fourth bar depicts the result for a Polypheny-DB instance where the data is fully replicated across all three data stores. The system can thus pick the data store with the best characteristics for executing a query. The results show that Polypheny-DB's routing system provides between 2.5 and 5 times the performance obtained with the individual data stores.

However, a full replication of data comes with a price tag: not only does it increases the required amount of storage space, it also adds an additional overhead on data modification queries since all changes need to be performed on all data stores. While the ability to distinguish between eagerly and lazily replicated placements in the form of the freshness aware query processing introduced in Section 8.5.2 might be a suitable solution in certain scenarios, in the context of transactional workloads, however, this is not applicable.

Figure 14.4 depicts the result of the Gavel MultiBench benchmark for different read and write ratios. While in scenarios with high read ratios, Polypheny-DB combines the advantages in terms of query performance provided by the underlying data stores, for update heavy scenarios, a full replication of data limits Polypheny-DB to the performance of the slowest involved data store.

The data allocation should therefore be adjusted such that an entity is only stored on those data stores which are most suitable for processing the queries accessing or modifying

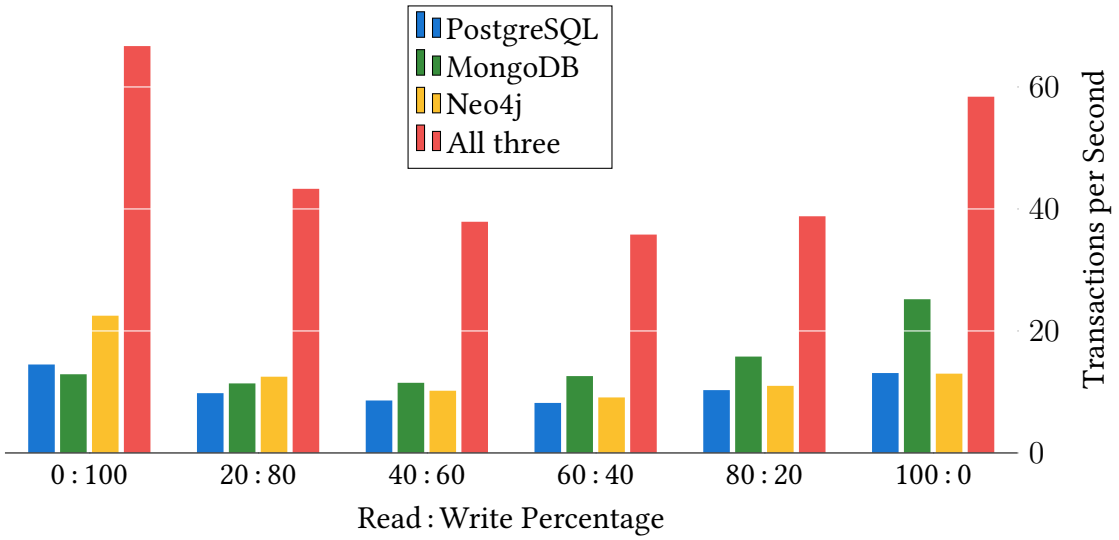


Figure 14.5 Results obtained with a manually optimized placement of data. The storage configuration is optimized for read-workload and is identical for all read/write ratios. Higher numbers are better.

that entity. Data allocation is therefore always a tradeoff between data replication and data partitioning. Replicating entities across multiple data stores can increase query performance for heterogeneous workloads while reducing the number of data placements typically increases the write performance.

Figure 14.5 depicts the results of the same benchmark as depicted in Figure 14.4, however, this time we have manually optimized the data allocation using functionality provided by Polypheny-DB. Compared to a full replication of data across all three data stores, the optimized placements provide superior results for all read/write ratios. The drop in performance for mixed read/write ratios can be explained as an effect of strong strict two-phase locking (SS2PL) used by Polypheny-DB for concurrency control. Since the write queries are on average less complex than the read-only queries generated by the MultiBench benchmark, the exclusive locks prevent parallel execution of read queries. This performance drop can also be observed for the individual data stores.

While Polypheny-DB allows the data allocation to be changed at runtime, the latest release of Polypheny-DB (version 0.7.0) is not yet doing this automatically. Such an automated and self-adaptive optimization of the data placements would especially be beneficial when dealing with changing workloads. We are currently working on implementing this within Polypheny-DB. A first prototype already shows very promising results.

The results of this experiment show that our approach to query routing in a PolyDBMS is efficient and reliably selects the most suitable data store(s).

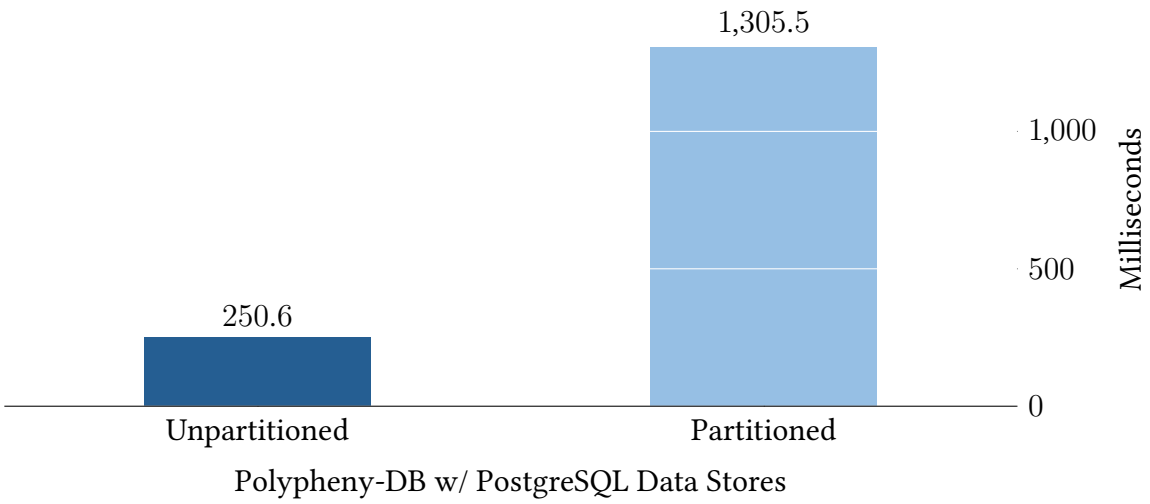


Figure 14.6 Comparing the performance of a Polypheny-DB instance with a single PostgreSQL data store against a Polypheny-DB instance with three PostgreSQL data stores using the TPC-C benchmark. In the three data store deployment, the `item` table is partitioned across the three data stores using the HASH partition function. The other tables are all stored on the same of the three data stores (no replication). The graph depicts the average latency in milliseconds across all transactions. Lower numbers are better.

14.6 Experiment 3: Data Partitioning

The ability to replicate and partition data across underlying data stores is an important part of the data allocation model we introduced in Chapter 8. While data replication has already been examined in Experiment 2, in this experiment we consider data partitioning.

In the conceptual model presented in this thesis, we distinguish between two types of partitioning, which we refer to as *horizontal partitioning* and *vertical partitioning*. The particular mechanics of these partitioning types depend on the data model; however, horizontal partitioning affects only the data, while vertical partitioning also involves splitting the schema. In this experiment, we focus on horizontal data partitioning.

When the data is partitioned among multiple underlying data stores, the system can no longer push down all operations to the underlying data stores, but might need to process some portions of the query in the integrated engine. This usually has a significant impact on query performance since the data must be streamed to Polypheny-DB. The goal of the experiment is to measure the overhead incurred by horizontal data partitioning. We therefore use the relatively complex TPC-C benchmark and compare the average latency for a configuration where the item table is stored entirely on one data store to an implementation where it is partitioned across three data stores. To compare the results,

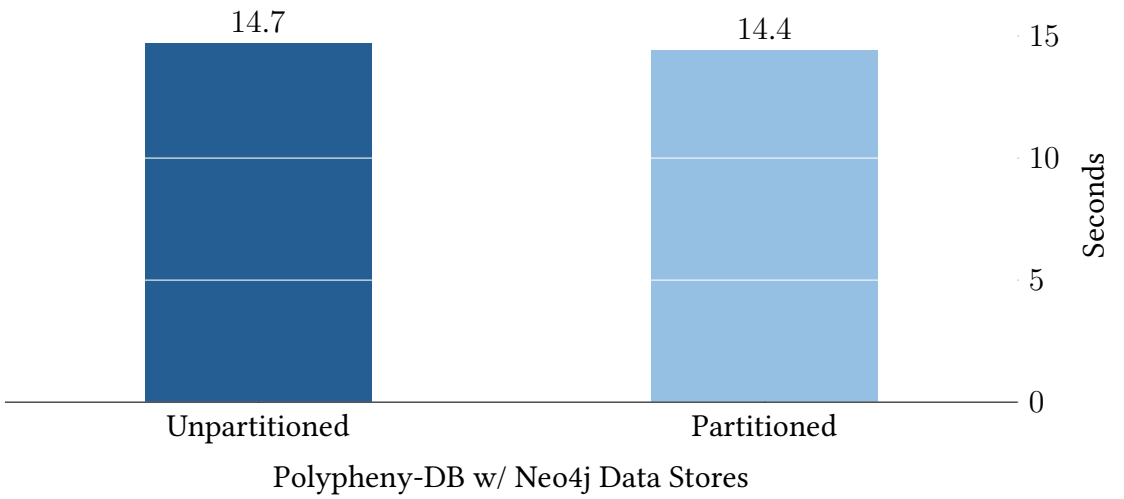


Figure 14.7 Comparing the performance of a Polypheny-DB instance with a single Neo4j data store against a Polypheny-DB instance with three Neo4j data stores using the TPC-C benchmark. In the three data store deployment, the `item` table is partitioned across the three data stores using the HASH partition function. The other tables are all stored on the same of the three data stores (no replication). The graph depicts the average latency in seconds across all transactions. Lower numbers are better.

we use multiple instances of the same database system. The item table is the largest table created by the TPC-C benchmark and is accessed in 96 % of the transactions.

In this experiment, we compare the performance of Polypheny-DB for partitioned and unpartitioned schemas. In the unpartitioned case, Polypheny-DB is used with one data store, and in the partitioned case with three data stores. All data stores are deployed on the same physical node together with Polypheny-DB and the client, thus the partitioned case does not benefit from horizontal scaling.

The results are depicted in Figure 14.6. With PostgreSQL, the mean latency is almost quintupled if the item table gets partitioned across three data stores. The reason for this is, that Polypheny-DB can no longer push down the full query to the data store but needs to process parts of the query in its integrated engine. This integrated engine is not only less optimized than PostgreSQL, it also requires data to be streamed from the data store to Polypheny-DB. This heavily impacts performance.

However, if we execute the same benchmark with Neo4j instead of PostgreSQL, the results are different. As depicted in Figure 14.7, there is only a small difference in performance between the partitioned and unpartitioned configuration. Since Neo4j does not natively support the relational TPC-C benchmark, most of the queries needs to be processed in the integrated engine anyway. The performance is therefore similar for both the partitioned and the unpartitioned case.

14.7 Summary and Discussion

The experiments have shown the potential of a PolyDBMS to enhance query performance in scenarios with heterogeneous data and workloads. While the overhead of using Polypheny-DB might be problematic for applications that require extremely low latencies, for most applications, they should be negligible. By using a learned cost model to simplify routing plan selection, we could further reduce the overhead in the future.

In addition, the experiments demonstrated the cross-model capabilities and performance of the integrated engine for both read and write workloads. Even for complex TPC-C workloads, the integrated engine provides acceptable query performance when the query cannot be pushed down.

However, a PolyDBMS is not (only) about performance: it enables use cases and applications that are not possible with existing solutions. The fact that it also improves performance at the same time is a welcome addition.

Part V

Discussion

Preface to Part V:
Let's Wrap It Up Then

In this final part of the thesis, we discuss and compare our contributions and our implementation Polypheny-DB with existing systems. We also situate these systems in the context of the PolyDBMS concept we have introduced in this thesis. This overview includes both commercial database systems and research prototypes—in particular, polystore systems.

The final chapter of this dissertation then wraps up the presented concepts and discusses whether our work is an answer to the research question that initiated this journey. By discussing the practical relevance of our work, we can come full circle and conclude this thesis. However, we by no means conclude this new branch of research that we have just begun to shed light on. The chapter—and thus this thesis—therefore close with an outlook on how this journey will be continued.

Die gefährlichste Weltanschauung ist die Weltanschauung derer, die die Welt nie angeschaut haben.

— Alexander von Humboldt

15

Related Work

This chapter provides an overview of related research and existing database systems in the light of the PolyDBMS concept. Furthermore, we compare these approaches with our implementation Polypheny-DB.

A PolyDBMS combines aspects from different types of database systems: Polystore and multistore systems, multimodel database systems, and some special services and tools. In this chapter, we present systems from each of these categories.

15.1 Polystore and Multistore Systems

In this thesis, we follow the taxonomy introduced in [TCG+17]. According to this, a multistore is a system that accepts queries through one query interface and utilizes multiple underlying data sores to process a query. Polystore systems, in contrast, accept queries through multiple query interfaces and also uses multiple, heterogeneous underlying data stores to process queries.

There are only a few surveys covering this emerging topic of database systems research. In [BV16], query processing and implementation techniques are discussed. Furthermore, the survey also compares some of the polystore systems that existed in 2016. A survey published by Tan et. al. [TCG+17], compares the BigDAWG, CloudMdsQL, Myria, and Apache Drill. The latest survey of systems can be found in [GKS+22]. In this survey, ten systems, including Polypheny-DB, are covered. The paper also discusses different architecture types. This survey has also been a great help for compiling this overview.

Apache Drill. A distributed query engine for the analysis of large-scale datasets [HN13]. Queries are accepted in ANSI SQL or the MongoDB query language. It supports various

NoSQL systems like HBase, Hive, and MongoDB. Relational databases are supported as well. Similar to Polypheny-DB, data sources can be added at runtime. Designed as a system for data analytics, it does not support data modification queries and its support for schema definition is limited to linking new data sources. It is thus more a platform for data analytics than a database system or PolyDBMS.

AWESOME. A multistore system designed for data analytics applications [DCG16]. Similar to Polypheny-DB, it supports the relational, document, and property graph data models. However, there is no integrated engine. Instead, there is a corresponding underlying data store for every data model. The capabilities therefore depend on the selected data stores. Furthermore, queries have to consider the native data models of the data stores and the individual query capabilities. This violates the PolyDBMS Requirement 3.3 (*Independence of Storage Configuration*).

BigDAWG. In BigDAWG [GCD+16] heterogeneous data stores are organized into "islands" (e.g., relational or array islands). An island can consist of multiple data stores. Each island has a specific data model and query language. A query can include multiple data stores from the same island; however, queries across multiple islands—and thus cross-model queries—are not possible. The BigDAWG system delivers great results [MGS+17] for heterogeneous read-only workloads. However, it does not support data modification or schema definition queries. Hence, the data needs to be loaded into the underlying data stores prior to the start of the BigDAWG system. It is thus more a data analytics platform than a database system or PolyDBMS.

BigIntegrator. Supporting an SQL-like query language, the BigIntegrator [ZR11] system is designed to access BigTable databases. Furthermore, it supports SQL-based relational databases. Similar to Polypheny-DB, the BigIntegrator system has an integrated engine that compensates for missing functionality on the underlying data stores. In contrast to Polypheny-DB and other Polystore systems, BigIntegrator is limited to SQL-based databases and the BigTable system and does not support other data stores. It internally uses a relational data model and does not support data modification or schema definition queries. Both violates the requirements that are specified for a PolyDBMS.

CloudMdsQL. A multistore system that acts as middleware and supports an SQL-like query language for querying heterogeneous data stores. The queries can contain embedded calls to the native query interface of each data store as sub-queries. In [KBV+16], the

authors present an implementation based on a distributed query engine called LeanXScale. The CloudMdsQL system does not support data manipulation operations. However, since it is possible to package native queries for the underlying data stores as subqueries, it may be possible to execute data modification queries. This, however, does not meet the requirements for a PolyDBMS. In addition, there is also no support for schema definition operations. For a PolyDBMS, the support for additional query languages and a logical schema is missing as well.

DBMS$^+$. In [LHB13], the authors propose an approach for query processing on polyglot system landscapes. By specifying requirements regarding the availability, consistency and execution costs, a DBMS$^+$ system would be able to generate multi-system execution plans. Since the proposed concept is only a vision, a concrete comparison is not possible. While the proposed idea of a DBMS$^+$ goes into the direction of what we introduced as PolyDBMS, their approach is focused on data streams. There is also no discussion of essential aspects like schema management or constraint enforcement.

ESTOCADA. A multistore system using an innovative constraint-based query rewriting technique to enable queries over heterogeneous data stores [ABD$^+$19]. Unfortunately, only little information is available on the system itself. ESTOCADA itself is only described as a vision in [BBD$^+$15]. The more recent research paper [ABD$^+$19] focuses on their constraint-based query rewriting technique. According to [GKS$^+$22], the system does not support data manipulation or data definition queries, which is a fundamental requirement for every PolyDBMS (see Requirement 3.7).

FORWARD. The FORWARD system introduced in [OPV15] is a Cloud service that acts as a middleware over heterogeneous data stores. The focus of their work is on the query language SQL++ that extends ANSI SQL and adds native JSON support. The system decomposes the SQL++ queries and translates them to the query languages of the underlying data stores. The underlying data stores are abstracted using (materialized) views. There is only little information on the FORWARD system itself. Furthermore, it seems no longer to be developed. It is unclear, to which extent the proposed concepts are implemented since no evaluation results are available. Since the system only supports one query language, the FORWARD system is to be classified as multistore and thus not fulfill the requirements of a PolyDBMS.

Icarus. The predecessor system of Polypheny-DB, introduced in [VSS17], is a multistore system that replicates data across multiple SQL data stores and supports data modification queries. For each query, the Icarus system selects the data store with the best characteristics for executing it. While its novel query routing strategy which, in a more enhanced version is also available in Polypheny-DB, provided excellent results for heterogeneous workloads, the full replication of data comes with a price tag. The Icarus system is limited to the relational data model (which violates the PolyDBMS Requirement 3.1) and does only support one query language (violates Requirement 3.2).

Myria. This multistore system features a query language called *MyriaL* that allows the expression of complex data analytics tasks and supports the assignment of variables and the construction of loops [WBB+17]. The system uses the *Hadoop File System* to move data between execution engines. It is designed for data analytics applications and does not support data manipulation or schema definition operations. Myria is limited to the relational data model. It does therefore not qualify as PolyDBMS.

Polybase. The add-on for the Microsoft SQL server allows to efficiently read and import data from Hadoop, Azure blob storage, and some other data sources [DHN+13]. It also allows to map these sources as so called "external tables" and query them as part of the relational schema. This is similar to the data sources approach provided in Polypheny-DB. However, our approach also supports the similar concept for the document and graph model. A Microsoft SQL server equipped with Polybase fulfills several of the PolyDBMS requirements. However, since Polybase is limited to the relational data model, it violates the multimodel requirement (see Requirement 3.1).

RHEEM. The RHEEM data processing system [ABBE+16] bridges the gap between data stores and the underlying platforms using a flexible operator mapping. Platform-agnostic operators are mapped to platform-specific execution operators that are executed directly on the data processing systems. An interesting concept in REEM are the so-called conversion graphs which specify the transformation between data models. Similar to Polypheny-DB, the system estimates the execution costs of an operation on the heterogeneous platforms to route queries. However, data is not moved between data stores except for within the scope of a query. According to [GKS+22], the REEM system does not support data manipulation or schema definition queries.

15.2 Multimodel Database Systems

Every PolyDBMS is by definition a multimodel database system. In recent years, many single model database systems have been extended to support multiple data models. This has often been done by some sort of translation layer which maps the additional data model(s) to the native model of the database system. However, there are also native multimodel systems, for instance, ArangoDB.

Furthermore, these systems often lack support for additional query languages. In a recent survey on multimodel databases [LH19], 23 systems have been analyzed and compared: 20 of these systems support only one query language, and the remaining three systems support two query languages. For this section, we selected four interesting multimodel database systems.

ArangoDB. A database system that has specifically been designed to natively support three data models. The key/value, document, and graph data models supported by ArangoDB[1] can be queried using a common query language called AQL (ArangoDB Query Language). It supports CRUD operations, but not schema definitions. These need to be done in the user interface of ArangoDB. The schema consists of two types of collections: document collections and edge collections. Graphs are modeled using both collection types. A node is thus a document connected to other documents as specified by the edge collection. This is an interesting combination of the document and graph data models that might also be interesting for a PolyDBMS. To qualify as a PolyDBMS, support for additional languages supporting cross-model queries would be required.

IBM Db2. IBM Db2[2], exemplary for many traditional, primarily relational database systems, added several interesting multimodel features to their originally purely relational database system. Since version 11.5.4, IBM Db2 also allows representing relational data as a graph using their optional Graph feature. The data is still stored relational; however, it creates a virtual graph that defines each row in a table as either a node or an edge. IBM Db2 thus approaches the functionality we have proposed by the term PolyDBMS using a monolithic architecture. However, with these database systems in their core still being relational databases and relying on relational structures to store the data, the semantics of the data model can be lost in cross-model queries. Furthermore, integrating domain-specific features and achieving the same performance as a native system based on

[1] https://arangodb.com/
[2] https://ibm.com/de-de/products/db2-database

this data model is challenging. On the other hand, the tight integration in one monolithic system might allow a more efficient query processing compared to our hybrid architecture approach. This is especially relevant for short running queries where actual execution time is minimal.

OrientDB. The aim of OrientDB[3] is to unify the document, key/value, graph and object data models. In this unified model, everything is a document. In OrientDB, documents cannot only embed, but also reference other documents. It also supports constraints similar to foreign key constraints in a relational database system. OrientDB only supports one query language, a SQL-like query language without traditional join statements. Instead, joins are expressed using a notation that requires explicit links (i.e., foreign keys).

SAP HANA. A hybrid OLTP and OLAP in-memory database system optimized for real-time data analysis. Originally built as a relational database system, HANA [FCP+12] has been extended to also support the graph data model. However, graphs are internally mapped to relational column storage [RPB+13]. It thus suffers from the same issues as IBM Db2 and other similar multimodel database implementations. With its optimization for different types of workload and the support for multiple data models, SAP HANA is close to a monolithic implementation of a PolyDBMS. However, while with SQL and openCypher, SAP HANA supports two query languages, their application is limited. openCypher can only be used within a "GraphWorkspace". Cross-model queries are not possible using openCypher. This is a violation of the PolyDBMS Requirement 3.4.

15.3 Services and Tools

The systems discussed in this section are not database systems; thus they cannot provide the full functionality of a PolyDBMS. However, these services, frameworks, and tools provide some of the functionality also provided by a PolyDBMS.

Apache Calcite. A framework for adding SQL query support to NoSQL database systems [BCRH+18]. It has a modular architecture and is used by several NoSQL database systems; for example the previously mentioned Apache Drill. It also comes with a very powerful query optimizer based on the Volcano optimizer model [GM93]. In Polypheny-DB,

[3] https://orientdb.org/

we also use a highly adapted version of this optimizer and SQL parser we have forked from Calcite several years ago.

Hackolade. This tool allows to visually model document and other NoSQL databases. It visualizes complex data structures using Entity-Relationship diagrams. It also allows to reverse engineer data structures. The strength of Hackolade[4] are polyglot applications requiring schema management across multiple NoSQL databases.

Hevo. A Software-as-a-Service platform to build ETL pipelines loading data from different sources. Hevo[5] avoids building the required transformation and ingestion pipelines for a data warehouse from scratch. While a PolyDBMS makes traditional data warehouses obsolete, there might still be data sources that require a more sophisticated data transformation process. Integrating such functionality into the PolyDBMS itself would offer even more flexibility.

OctoSQL. A CLI tool that allows to query several file formats and database systems using SQL. OctoSQL[6] also allows to join data from different sources. However, due to its architecture, OctoSQL is only feasible for small amounts of data; especially when joining data. It would be interesting to add an adapter to Polypheny-DB that utilizes OctoSQL to query even more data sources.

[4] https://hackolade.com/
[5] https://hevodata.com/
[6] https://github.com/cube2222/octosql

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

16

Conclusion and Outlook

This final chapter concludes the dissertation by providing three perspectives on our work: a retrospective on our contributions, a section that validates our contributions against the motivating scenario and discusses the practical relevance of our work, and a prospective on future work.

16.1 Retrospective

Our research deals with the question of how the growing demand for a tighter integration of data between different applications can be met without sacrificing the performance for heterogeneous data and mixed workloads. With the growth of data significantly exceeding the growth of computational power, this is a topic of increasing importance. As existing data management solutions are either not suitable for the outlined use case, or are suitable only with significant limitations and performance degradation, we have argued for the need for a new generation of database management solutions, which we named PolyDBMS.

With the specifications presented in Chapter 3, we have defined a concise set of requirements for this new class of database management systems. These requirements are deliberately specified in an implementation-agnostic form that leaves the design and architecture of such a system open. However, while our study of related work presented in Chapter 15 acknowledges the huge achievements and optimizations that empower existing multimodel databases with their monolithic architecture, we also realized the huge potential of the polystore idea.

With our approach of combining the architectural idea of polystores with the capabilities of a "traditional" database management system, we aim to combine the best of both worlds.

This combination of concepts that bridges the gap between polystores and database systems creates a new branch of database system research that requires rethinking established database system concepts. In this work, we have addressed three fundamental research objectives:

- The schema model presented in this thesis enables the combination of different data models beneath a single logical schema that preserves the semantics of the individual models while enabling cross-model queries. Additionally, our model provides tremendous flexibility in defining tailored data replication and data partitioning configurations.
- Our query representation model allows queries formulated in different query languages and based on different data models, to be represented in a single internal representation, while preserving the characteristics and properties of the query languages and their underlying data models, and still allowing cross-model queries.
- The query routing approach presented in this thesis defines an approach for adaptively scheduling the decomposition of queries and their assignment to execution engines. This allows taking full advantage of the heterogeneous execution and storage engines.

Based on these conceptual models, we have developed Polypheny-DB. Starting as a research prototype, this system quickly evolved into a fully functional implementation of a PolyDBMS. The fact that it was selected for the Google Summer of Code for the second time in 2022 is a testament to the maturity of this system and the concepts on which it is built.

From the beginning, our research has been guided by empiricism and a continuous evaluation of our results. As part of this effort, we developed the Chronos system, a platform for automating the entire evaluation workflow. Chronos is also an important building block in our effort to increase the reproducibility of our results.

In addition to quantitative results obtained with custom and industry-standard benchmarks, we also presented an approach for verifying the correctness of a PolyDBMS. While the Polyfier approach presented in Chapter 13 cannot prove the correctness of a PolyDBMS, it eventually finds any inconsistency.

With our research, we have contributed the specifications of a new class of database systems and discussed possible architectural models. The concepts presented in this dissertation lay the foundations and formalize fundamental concepts for this new class

of systems. With Polypheny-DB and the surrounding ecosystem, we have created a fully functional implementation of this new breed of database systems. In addition, we have introduced the tools to thoroughly evaluate it both quantitatively and qualitatively. The obtained results show the potential of this new class of database systems—even in comparison to industry-leading systems.

16.2 Cui Bono?

A PolyDBMS enables a tighter integration of data between different applications and thus allows to draw more value from existing data. Reducing redundantly stored data and eliminating the need for custom synchronization processes significantly reduces the risk of inconsistencies. The ability to always rely on the latest data can be a game-changer for many scenarios. At the same time, a PolyDBMS also enables a more efficient usage of compute and storage resources, since the load can be distributed across multiple machines that previously were only allocated to manage the data of specific applications. By utilizing the optimizations and advantages of heterogeneous storage and execution engines, a PolyDBMS can at the same time improve the overall performance.

The practical relevance of a PolyDBMS is enormous for both scientific and business applications. A PolyDBMS bears the potential to enhance every sort of data processing use case dealing with heterogeneous data or mixed workloads—or both. It combines and provides the optimizations of multiple domain-specific database systems independent of query language and query interface needs.

In Chapter 2 of this thesis, we have introduced a motivational scenario based on a fictive online auction house. A PolyDBMS like Polypheny-DB enables Gavel to consolidate its data while still accessing and manipulating it using the query languages supported by the applications and without losing any semantics. The ability to execute cross-model queries in real-time enables new applications.

However, there are also some limitations. As the evaluations presented in Chapter 14 have shown, Polypheny-DB and probably also similar implementations of a PolyDBMS add an overhead on each query. While this overhead of a few milliseconds should be negligible for a majority of applications, there are indeed applications (e.g., stock trading) where such overheads are significant. Also, for use cases with homogenous data and workloads, the mileage of using a PolyDBMS may vary. It primarily depends on whether the PolyDBMS can add any additional value. If there is a single database system that is well suited for the workload and data, adding a PolyDBMS brings no immediate

advantages for this application. However, on the other hand, it adds the flexibility to support additional applications in the future or to partition the data across multiple machines.

A PolyDBMS, as every database management system, comes with a specific set of supported features and query functions. While a PolyDBMS might support more features than each of its data stores, it is practically impossible to support all features supported by any existing database system. Allowing functionality of an underlying data store to be accessed directly within a query would be a violation of the *Independence of Storage Configuration* requirement. In terms of query functions, this can be solved by the means of user-defined functions. Most other functionality should be emulatable using stored procedures. However, this might be less efficient.

16.3 Prospective

Our work has drawn the research on polystores and database systems more closely together and created new challenges and opportunities—of which we have yet just scratched the surface. Many established concepts in the context of database systems and distributed systems need to be reconsidered when they get applied to a PolyDBMS. In this section, we present an outlook on future work in the PolyDBMS context.

The conceptual models introduced in this thesis focus on data stores under the exclusive control of the PolyDBMS and being mapped in a Local-as-View manner. Our implementation Polypheny-DB goes beyond that by also introducing the notion of a *data source* mapped in a Global-as-View manner. Formalizing the implications of this distinction and creating a proper model for the notion of data sources in a PolyDBMS would be an interesting extension. Furthermore, it would be interesting to consider additional types of adapters, such as a data stream adapter or an API adapter. The latter would also allow the integration of non-database systems like search engines or web services.

The three data models discussed in this thesis are certainly among the most important database data models and already enable a large number of applications and use cases. The extension of the schema model and the query representation model by further data models such as the RDF or the Wide-Column model would increase the number of possible application scenarios even further.

A very interesting aspect of a PolyDBMS we did not discuss in this thesis is the handling of failing data stores. Furthermore, it would also be very interesting to consider the

implications of a distributed PolyDBMS and to formalize the interactions between the two layers of distribution. This also requires extending the routing model to consider data stores available on other instances—while taking the latency and data transfer costs into account. This gets especially interesting if queries are decomposed and the global routing plan includes the sequential processing of multiple PolyDBMS instances. It might even be interesting to consider the implications of distributed data stores. For data stores that allow a distributed deployment, the deployments of a data store at the individual locations of the distributed PolyDBMS can be part of the same distributed setup and are thus then accessed by multiple PolyDBMS instances, resulting in even a third level of distribution.

An important pillar of the PolyDBMS idea is the *Independence of Storage Configuration*. Without this requirement, the ability of the PolyDBMS to partition data and adapt to changes in the workload would be greatly diminished. However, as mentioned before, this requirement also introduces certain difficulties and prevents the PolyDBMS from allowing functionalities of the underlying data stores to be directly accessed in a query. In our future work, we want to reduce this burden by defining a new type of user-defined function in Polypheny-DB which allows specifying how the user-defined function can be pushed-down to an underlying data store, efficiently making use of functionalities of the underlying data stores.

Another very interesting topic in the context of PolyDBMS is that of self-adaptiveness. With the ability to combine data replication and data partitioning across heterogeneous data stores, our schema model already offers plenty of options to adapt to the workload. By automatically creating materialized views or creating indexes, there are even more possibilities to adapt. Our first implementation of self-adaptiveness in Polypheny-DB shows very promising results that motivate further research in this direction.

Another extension for Polypheny-DB we are currently working on is a learned cost model for the query routing. This has the potential to further improve the query performance and reduce the latency of both short-running and long-running queries. In the context of query routing, it would also be interesting to consider the migration of data prior to the execution of a query or the temporary materialization of intermediate results on an underlying data store in order to improve the performance of join operations.

Bibliography

[ABBE+16] Divy Agrawal, Lamine Ba, Laure Berti-Equille, Sanjay Chawla, Ahmed Elmagarmid, Hossam Hammady, Yasser Idris, Zoi Kaoudi, Zuhair Khayyat, Sebastian Kruse, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiane-Ruiz, Nan Tang, and Mohammed J. Zaki. Rheem: Enabling Multi-Platform Task Execution. In *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD/PODS '16, pages 2069–2072, San Francisco, CA, USA. ACM, June 26, 2016. ISBN: 978-1-4503-3531-7. DOI: 10/gqd7gt.

[ASS08] Fuat Akal, Heiko Schuldt, and Hans-Jörg Schek. Toward replication in grids for digital libraries with freshness and correctness guarantees. *Concurrency and Computation: Practice and Experience*, 20(17):1981–1993, December 10, 2008. ISSN: 15320626, 15320634. DOI: 10/d5q3j9.

[ABD+19] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD/PODS '19, pages 1660–1677, Amsterdam, Netherlands. ACM, June 25, 2019. ISBN: 978-1-4503-5643-5. DOI: 10/gpst2j.

[AG08] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys*, 40(1):1–39, February 2008. ISSN: 0360-0300, 1557-7341. DOI: 10/fwnnc5.

[BW06] Sudipta Basu and Gregory B. Waymire. Recordkeeping and Human Evolution. *Accounting Horizons*, 20(3):201–229, 2006. ISSN: 0888-7993. DOI: 10/c4k78z.

[Bat18] Rahul Batra. A History of SQL and Relational Databases. In *SQL Primer: An Accelerated Introduction to SQL Basics*, pages 183–187. Apress, Berkeley, CA, USA, 2018. ISBN: 978-1-4842-3576-8. DOI: 10/ggwc25.

[BAH+19] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD/PODS '19, pages 1757–1772, Amsterdam, Netherlands. ACM, June 25, 2019. ISBN: 978-1-4503-5643-5. DOI: 10/ght34v.

[BCRH+18] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18, pages 221–230, Houston, TX, USA. ACM Press, 2018. ISBN: 978-1-4503-4703-7. DOI: 10/gp49dt.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, February 6, 2019. ISSN: 1433-2779, 1433-2787. DOI: 10/gnpqrd.

[BM04] Paul Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition, 2004. URL: https://www.w3.org/TR/xmlschema-2/ (visited on 02/24/2022).

[Boa11] IEEE-SA Standards Board. IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition). *IEEE P1490/D1, May 2011*, June 2011.

[BAK+18] Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl, and Volker Markl. PEEL: A Framework for Benchmarking Distributed Systems and Algorithms. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking for the Analytics Era*. Volume 10661, pages 9–24. Springer International Publishing, Cham, 2018. ISBN: 978-3-319-72400-3 978-3-319-72401-0. DOI: 10/g8xb.

[BV16] Carlyna Bondiombouy and Patrick Valduriez. Query Processing in Multistore Systems: An Overview. Research Report RR-8890, Inria Sophia Antipolis Mediterranean, March 2016. URL: https://hal.inria.fr/hal-01289759.

[BCC+16] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. A Formal Presentation of MongoDB (Extended Version). (arXiv:1603.09291v1), 2016. DOI: 10/h3gk.

[BCC+18] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and Complexity of MongoDB Queries. 98:9:1–9:23, 2018. Benny Kimelfeld and Yael Amsterdamer, editors. ISSN: 1868-8969. DOI: 10/gphk73.

[BP04] Mokrane Bouzeghoub and Verónika Peralta. A Framework for Analysis of Data Freshness. In *Proceedings of the 2004 International Workshop on Information Quality in Information Systems*. IQIS '04, pages 59–67, Paris, France. ACM Press, 2004. ISBN: 978-1-58113-902-0. DOI: 10/cgrx3t.

[BS14] Filip-Martin Brinkmann and Heiko Schuldt. Towards Archiving-as-a-Service: A Distributed Index for the Cost-effective Access to Replicated Multi-Version Data. In *Proceedings of the 19th International Database Engineering & Applications Symposium*. IDEAS '15, pages 81–89, Yokohama, Japan. ACM Press, 2014. ISBN: 978-1-4503-3414-3. DOI: 10/gnccvn.

[BOB+21] Adam Brumm, Adhi Agus Oktaviana, Basran Burhan, Budianto Hakim, Rustan Lebe, Jian-xin Zhao, Priyatno Hadi Sulistyarto, Marlon Ririmasse, Shinatria Adhityatama, Iwan Sumantri, and Maxime Aubert. Oldest cave art found in Sulawesi. *Science Advances*, 7(3), January 13, 2021. ISSN: 2375-2548. DOI: 10/fqq7.

[BBD+15] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. Invisible Glue: Scalable Self-Tunning Multi-Stores. In *Seventh Biennial Conference on Innovative Data Systems Research*. CIDR 2015, Asilomar, CA, USA, 2015. URL: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper7.pdf.

[Bun97] Peter Buneman. Semistructured Data. In Alberto O. Mendelzon and Z. Meral Özsoyoglu, editors, *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, Tucson, Arizona, USA. ACM Press, May 1997. DOI: 10/dngsf5.

[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A History and Evaluation of System R. *Communications of the ACM*, 24(10):632–646, October 1981. ISSN: 0001-0782. DOI: 10/fdthd4.

[Cha98] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database systems*. PODS '98, pages 34–43, Seattle, WA, USA. ACM Press, 1998. ISBN: 978-0-89791-996-8. DOI: 10/bmxvpw.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model—toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976. ISSN: 0362-5915. DOI: 10/cp2d3z.

[CGM00] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to Improve Freshness. *ACM SIGMOD Record*, 29(2):117–128, June 2000. ISSN: 0163-5808. DOI: 10/fgkx6j.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970. ISSN: 0001-0782, 1557-7317. DOI: 10/dwxst4.

[Cod90] Edgar F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley, Reading, Massachusetts, USA, 1990. 538 pages. ISBN: 978-0-201-14192-4.

[CGP80] Edward G. Coffman, Erol Gelenbe, and Brigitte Plateau. Optimization of the number of copies in a distribution data base. In *Proceedings of the 1980 International Symposium on Computer Performance Modelling, Measurement and Evaluation*. PERFORMANCE '80, pages 257–263, Toronto, Ontario, Canada. ACM Press, 1980. ISBN: 978-0-89791-019-4. DOI: 10/fpsqcm.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10, pages 143–154, Indianapolis, Indiana, USA. ACM Press, 2010. ISBN: 978-1-4503-0036-0. DOI: 10/cxjrfd.

[CMR11] Carlos Coronel, Steven Morris, and Peter Rob. *Database Systems: Design, Implementation, and Management*. Course Technology Cengage Learning, South Melbourne, Australia, 2011. ISBN: 978-0-538-46968-5 978-0-538-74884-1.

[Cou10] Transaction Processing Performance Council. TPC Benchmark C Revision 5.11. Standard Specification, February 2010. URL: https://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[CMM+00] Michel Courson, Alan Mink, Guillaume Marçais, and Benjamin Traverse. An Automated Benchmarking Toolset. In Marian Bubak, Hamideh Afsarmanesh, Bob Hertzberger, and Roy Williams, editors, *High Performance Computing and Networking*. Volume 1823, pages 497–506. Springer, Berlin, Heidelberg, 2000. ISBN: 978-3-540-67553-2 978-3-540-45492-2. DOI: 10/dp398r.

[CHI15] Tom Crick, Benjamin A. Hall, and Samin Ishtiaq. Reproducibility as a Technical Specification. (arXiv:1504.01310 [cs]), June 15, 2015. DOI: 10/h3gg.

[Dad96] Peter Dadam. *Verteilte Datenbanken und Client/Server-Systeme: Grundlagen, Konzepte und Realisierungsformen*. Springer, 1996. 415 pages. ISBN: 978-3-540-61399-2.

[DCG16] Subhasis Dasgupta, Kevin Coakley, and Amarnath Gupta. Analytics-Driven Data Ingestion and Derivation in the AWESOME Polystore. In *Proceedings of the 2016 IEEE International Conference on Big Data*. Big Data 2016, pages 2555–2564, Washington DC, USA. IEEE, December 2016. ISBN: 978-1-4673-9005-7. DOI: 10/gqd5fj.

[DHN+13] David J. DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split Query Processing in Polybase. In *Proceedings of the 2013 International Conference on Management of Data*. SIGMOD '13, pages 1255–1266, New York, USA. ACM Press, 2013. DOI: 10/gqd7hs.

[Dha13] Vasant Dhar. Data Science and Prediction. *Communications of the ACM*, 56(12):64–73, December 2013. ISSN: 0001-0782, 1557-7317. DOI: 10/gdm28r.

[DPC+13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, December 2013. ISSN: 2150-8097. DOI: 10/gnkdbg.

[ESC14] Marcus Edel, Anand Soni, and Ryan R. Curtin. An Automatic Benchmarking System. In *Proceedings of the NIPS 2014 Workshop on Software Engineering for Machine Learning*. SE4ML'2014, Montréal, Canada, 2014. URL: https://www.mlpack.org/static/pub/2014automatic.pdf (visited on 12/05/2021).

[Ell14] Timo Elliott. What is Hybrid Transaction/Analytical Processing (HTAP)? December 15, 2014. URL: https://www.zdnet.com/article/what-is-hybrid-transactionanalytical-processing-htap/ (visited on 04/02/2022).

[EN16] Ramez Elmasri and Sham Navathe. *Fundamentals of Database Systems*. Pearson, Hoboken, NJ, USA, seventh edition edition, 2016. 1242 pages. ISBN: 978-0-13-397077-7.

[Fei03] Dror G. Feitelson. Experimental Computer Science: The Need for a Cultural Change, 2003. URL: https://www.cs.huji.ac.il/~feit/papers/exp05.pdf (visited on 12/01/2021).

[FGG+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, Houston, TX, USA. ACM, May 27, 2018. ISBN: 978-1-4503-4703-7. DOI: 10/gf3jz8.

[FBS12] Juliana Freire, Philippe Bonnet, and Dennis Shasha. Computational Reproducibility: State-of-the-Art, Challenges, and Database Research Opportunities. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12, pages 593–596, Scottsdale, Arizona, USA. ACM Press, 2012. ISBN: 978-1-4503-1247-9. DOI: 10/gkmztw.

[FCP+12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *ACM SIGMOD Record*, 40(4):45–51, January 11, 2012. ISSN: 01635808. DOI: 10/fx3t6m.

[GCD+16] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. The BigDAWG Polystore System and Architecture. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference*. HPEC 2016, pages 1–6, Waltham, MA, USA. IEEE, September 2016. ISBN: 978-1-5090-3525-0. DOI: 10/gqd47x.

[GMUW13] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database Systems: Pearson New International Edition*. Pearson Education Limited, 2nd edition, July 17, 2013. 1140 pages. ISBN: 978-1-292-02447-9.

[GRH+20] Ralph Gasser, Luca Rossetto, Silvan Heller, and Heiko Schuldt. Cottontail DB: An Open Source Database System for Multimedia Retrieval and Analysis. In *Proceedings of the 28th ACM International Conference on Multimedia*. MM '20, pages 4465–4468, Seattle WA USA. ACM, October 12, 2020. ISBN: 978-1-4503-7988-5. DOI: 10/hw9h.

[GWF+17] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. NoSQL database systems: a survey and decision guidance. *Computer Science - Research and Development*, 32(3):353–365, July 2017. ISSN: 1865-2034, 1865-2042. DOI: 10/gpb3xq.

[Gia18] Ivan Giangreco. *Database Support for Large-Scale Multimedia Retrieval*. Doctoral Thesis, University of Basel, Basel, 2018. 291 pages. DOI: 10/g8w6.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 33(2):51–59, June 2002. ISSN: 0163-5700. DOI: 10/dvqdjf.

[GKS+22] Daniel Glake, Felix Kiehn, Mareike Schmidt, Fabian Panse, and Norbert Ritter. Towards Polyglot Data Stores – Overview and Open Research Questions. (arXiv:2204.05779), April 12, 2022. URL: http://arxiv.org/abs/2204.05779.

[GM93] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. ICDE '93, pages 209–218, Vienna, Austria. IEEE, 1993. ISBN: 978-0-8186-3570-0. DOI: 10/fjmk4h.

[Gra12] Dan Graham. The Data Temperature Spectrum. White Paper EB-6599, Teradata Corporation, June 2012. URL: http://downloads.teradata.com/download/cdn/media/whitepapers/The_Data_Temperature_Spectrum.pdf.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, Cannes, France. IEEE Computer Society, September 1981.

[GP87] Jim Gray and Franco Putzolu. The 5 Minute Rule for Trading Memory for Disc Accesses and the 5 Byte Rule for Trading Memory for CPU Time. *ACM SIGMOD Record*, 16(3):395–398, December 1987. ISSN: 0163-5808. DOI: 10/fpzzsb.

[Gre19] Alastair Green. SQL ... and now GQL. September 12, 2019. URL: http://opencypher.org/articles/2019/09/12/SQL-and-now-GQL/ (visited on 02/25/2020).

[GBM16] Danilo Guerrera, Helmar Burkhart, and Antonio Maffia. Reproducible Stencil Compiler Benchmarks Using PROVA! In *Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. PMBS 2016, pages 108–115, Salt Lake City, UT, USA. IEEE, November 2016. ISBN: 978-1-5090-5218-9. DOI: 10/gkmzs3.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 2, 1983. ISSN: 0360-0300, 1557-7341. DOI: 10/cs2zdt.

[HMB+15] Allan Hanbury, Henning Müller, Krisztian Balog, Torben Brodt, Gordon V. Cormack, Ivan Eggel, Tim Gollub, Frank Hopfgartner, Jayashree Kalpathy-Cramer, Noriko Kando, Anastasia Krithara, Jimmy Lin, Simon Mercer, and Martin Potthast. Evaluation-as-a-Service: Overview and Outlook. (arXiv:1512.07454 [cs]), December 23, 2015. DOI: 10/h3gh.

[HG14] Rakebul Hasan and Fabien Gandon. A Machine Learning Approach to SPARQL Query Performance Prediction. In *Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, pages 266–273, Warsaw, Poland. IEEE, August 2014. ISBN: 978-1-4799-4143-8. DOI: 10/gpxkf2.

[HR16] Hadi Hashem and Daniel Ranc. Evaluating NoSQL Document Oriented Data Model. In *Proceedings of the 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops*. FiCloudW 2016, pages 51–56, Vienna, Austria. IEEE, August 2016. ISBN: 978-1-5090-3946-3. DOI: 10/gn3rpc.

[HN13] Michael Hausenblas and Jacques Nadeau. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data*, 1(2):100–104, June 2013. ISSN: 2167-6461, 2167-647X. DOI: 10/gftd9r.

[HM85] Dennis Heimbigner and Dennis McLeod. A Federated Architecture for Information Management. *ACM Transactions on Information Systems*, 3(3):253–278, July 1985. ISSN: 1046-8188, 1558-2868. DOI: 10/bh5s3j.

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007. ISSN: 1931-7883, 1931-7891. DOI: 10/bsnkxv.

[HTT09] Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009. ISBN: 978-0-9825442-0-4. DOI: 10/gdxxq9.

[HKC06] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. Efficient Identification of Hot Data for Flash Memory Storage Systems. *ACM Transactions on Storage*, 2(1):22–40, February 2006. ISSN: 1553-3077. DOI: 10/d3g7hj.

[Hu20] Yingjie Hu. Building benchmarking frameworks for supporting replicability and reproducibility: spatial and textual analysis as an example. (arXiv:2007.01978 [cs]), July 3, 2020. DOI: 10/h3gf.

[HLC+20] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, August 2020. ISSN: 2150-8097. DOI: 10/ghjr2s.

[Hä05] Theo Härder. DBMS Architecture – The Layer Model and its Evolution. *Datenbank-Spektrum*, 5(13):45–57, May 2005. URL: https://www.csd.uoc.gr/~hy460/pdf/Hae05a.DBSpektrum.pdf (visited on 03/14/2022).

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley professional computing. Wiley, 1st edition, 1991. 720 pages. ISBN: 978-0-471-50336-1.

[JLV+00] Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, and Panos Vassiliadis. *Data Warehouse Refreshment*. In *Fundamentals of Data Warehouses*. Springer, Berlin, Heidelberg, 2000, pages 47–85. ISBN: 978-3-662-04140-6 978-3-662-04138-3. DOI: 10/g9j3.

[JBP+] Alistair Johnson, Lucas Bulgarelli, Tom Pollard, Steven Horng, Leo Anthony Celi, and Roger Mark. MIMIC-IV. DOI: 10/hh3x.

[KBT04] Tomas Kalibera, Lubomír Bulej, and Petr Tuma. Generic Environment for Full Automation of Benchmarking. In Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf H. Reussner, and Franz Schweiggert, editors, *Testing of Component-Based Systems and Software Quality, Proceedings of SOQUA 2004 (First International Workshop on Software Quality) and TECOS 2004 (Workshop Testing Component-Based Systems)*, volume P-58 of *LNI*, pages 125–132, Erfurt, Germany. GI, 2004. URL: https://dl.gi.de/20.500.12116/28485 (visited on 12/05/2021).

[KLM+06] Tomas Kalibera, Jakub Lehotsky, David Majda, Branislav Repcek, Michal Tomcanyi, Antonin Tomecek, Petr Tuma, and Jaroslav Urban. Automated Benchmarking and Analysis Tool. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and tools*. valuetools '06, Pisa, Italy. Association for Computing Machinery, 2006. ISBN: 978-1-59593-504-5. DOI: 10/cq5233.

[KA08] Max Kanat-Alexander. The Purpose of Software. Code Simplicity. February 27, 2008. URL: https://codesimplicity.com/post/the-purpose-of-software/ (visited on 05/17/2022).

[Kle17] Andy Klein. Hard Drive Cost Per Gigabyte. The Backblaze Blog. July 11, 2017. URL: https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/ (visited on 03/14/2022).

[KBV+16] Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez, Ricardo Jimenez-Peris, Raquel Pau, and José Pereira. The CloudMdsQL Multistore System. In *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16, pages 2113–2116, San Francisco, California, USA. ACM Press, June 2016. ISBN: 978-1-4503-3531-7. DOI: 10/gnfkjd.

[KLK20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking: For Scientists and Engineers*. Springer International Publishing, Cham, 2020. ISBN: 978-3-030-41704-8 978-3-030-41705-5. DOI: 10/g8w8.

[KP88] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming - JOOP*, 1(3):26–49, August 1988. ISSN: 0896-8438.

[KT18] Pushpendra Kumar and Ramjeevan Singh Thakur. Recommendation system techniques and related issues: a survey. *International Journal of Information Technology*, 10(4):495–501, December 2018. ISSN: 2511-2104. DOI: 10/gp2hxh.

[Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. ISSN: 0734-2071, 1557-7333. DOI: 10/fwcbwn.

[Lec18] Jens Lechtenbörger. Two-Phase Commit Protocol. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 4266–4270. Springer, New York, NY, USA, 2018. ISBN: 978-1-4614-8265-9. DOI: 10/hk3b.

[LLS13] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. Identifying Hot and Cold Data in Main-Memory Databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering*. ICDE '13, pages 26–37, Brisbane, QLD, Australia. IEEE, April 2013. ISBN: 978-1-4673-4910-9 978-1-4673-4909-3 978-1-4673-4908-6. DOI: 10/gpn9zs.

[LHB13] Harold Lim, Yuzhang Han, and Shivnath Babu. How to Fit when No One Size Fits. In *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research*. CIDR 2013, Asilomar, CA, USA, January 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper102.pdf (visited on 06/27/2022).

[LE13] Jimmy Lin and Miles Efron. Evaluation as a Service for Information Retrieval. *ACM SIGIR Forum*, 47(2):8–14, December 21, 2013. ISSN: 0163-5840. DOI: 10/gcpvc5.

[LE14] Jimmy Lin and Miles Efron. Infrastructure Support for Evaluation as a Service. In *Proceedings of the 23rd International Conference on World Wide Web*. WWW '14 Companion, pages 79–82, Seoul, Korea. ACM Press, 2014. ISBN: 978-1-4503-2745-9. DOI: 10/gkmzth.

[LH19] Jiaheng Lu and Irena Holubová. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Computing Surveys*, 52(3):1–38, June 2019. ISSN: 0360-0300, 1557-7341. DOI: 10/gn7pz4.

[LHC18a] Jiaheng Lu, Irena Holubová, and Bogdan Cautis. Multi-model Databases and Tightly Integrated Polystores. 27th ACM International Conference on Information and Knowledge Management (CIKM 2018), Lingotto, Italy, October 26, 2018. URL: https://www.cs.helsinki.fi/u/jilu/documents/CIKMTutorial2018.pdf (visited on 02/02/2022).

[LHC18b] Jiaheng Lu, Irena Holubová, and Bogdan Cautis. Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. CIKM '18, pages 2301–2302, Torino, Italy. ACM, October 17, 2018. ISBN: 978-1-4503-6014-2. DOI: 10/gn7p4s.

[MET+21] Zhiyi Ma, Kawin Ethayarajh, Tristan Thrush, Somya Jain, Ledell Wu, Robin Jia, Christopher Potts, Adina Williams, and Douwe Kiela. Dynaboard: An Evaluation-As-A-Service Platform for Holistic Next-Generation Benchmarking. (arXiv:2106.06052 [cs]), May 20, 2021. DOI: 10/h3gj.

[MSV17] József Marton, Gábor Szárnyas, and Dániel Varró. Formalising openCypher Graph Queries in Relational Algebra. In Mārīte Kirikova, Kjetil Nørvåg, and George A. Papadopoulos, editors, *Advances in Databases and Information Systems*. Volume 10509, pages 182–196. Springer International Publishing, Cham, 2017. ISBN: 978-3-319-66916-8 978-3-319-66917-5. DOI: 10/gkqhs6.

[MGS+17] Tim Mattson, Vijay Gadepally, Zuohao She, Adam Dziedzic, and Jeff Parkhurst. Demonstrating the BigDAWG Polystore System for Ocean Metagenomic Analysis. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*. CIDR 2017, Chaminade, California, USA,

2017. URL: http://cidrdb.org/cidr2017/papers/p120-mattson-cidr17.pdf (visited on 06/27/2022).

[NLF99] Felix Naumann, Ulf Leser, and Johann Christoph Freytag. Quality-driven Integration of Heterogenous Information Systems. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of 25th International Conference on Very Large Data Bases*. VLDB '99, pages 447–458, Edinburgh, Scotland, UK. Morgan Kaufmann, September 1999. URL: http://www.vldb.org/conf/1999/P43.pdf.

[NJ00] Matthias Nicola and Matthias Jarke. Performance Modeling of Distributed and Replicated Databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):645–672, August 2000. ISSN: 10414347. DOI: 10/fbtn77.

[NL93] Jakob Nielsen and Thomas K. Landauer. A Mathematical Model of the Finding of Usability Problems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '93, pages 206–213, Amsterdam, The Netherlands. ACM Press, 1993. ISBN: 978-0-89791-575-5. DOI: 10/bbv823.

[Nye17] Timothy Nyerges. Physical Data Models. In John P. Wilson, editor, *The Geographic Information Science & Technology Body of Knowledge*. 1st quarter 2017 edition edition, February 24, 2017. DOI: 10/hf9z.

[OPV15] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The SQL++ Query Language: Configurable, Unifying and Semi-structured. (arXiv:1405.3631), December 14, 2015. DOI: 10/h3gm.

[OO14] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Philadelphia, PA, USA. USENIX Association, 2014. ISBN: 978-1-931971-10-2.

[OW05] Mikael Onsjö and Osamu Watanabe. Simple Algorithms for Graph Partition Problems. *Research Reports on Mathematical and Computing Sciences*. C: Computer Science, August 2005. ISSN: 1342-2812.

[OST20] Raphael Otten, Martin Schrepp, and Jörg Thomaschewski. Visual Clarity as Mediator between Usability and Aesthetics. In *Proceedings of the Conference on Mensch und Computer*. MuC'20: Mensch und Computer 2020, pages 11–15, Magdeburg, Germany. ACM, September 6, 2020. ISBN: 978-1-4503-7540-5. DOI: 10/gm96qz.

[OH11] Yi Ou and Theo Härder. Trading Memory for Performance and Energy. In Jianliang Xu, Ge Yu, Shuigeng Zhou, and Rainer Unland, editors, *Database Systems for Advanced Applications*. Volume 6637, pages 241–253. Springer, Berlin, Heidelberg, 2011. ISBN: 978-3-642-20244-5. DOI: 10/d563vq.

[OU86] Martyn A. Ould and Charles Unwin. *Testing in Software Development*. Cambridge University Press, 1986. ISBN: 978-1-4503-7540-5.

[OV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer Science+Business Media, New York, USA, 3rd edition, 2011. 845 pages. ISBN: 978-1-4419-8833-1 978-1-4419-8834-8.

[PDN+12] Dongchul Park, Biplob Debnath, Youngjin Nam, David H. C. Du, Youngkyun Kim, and Youngchul Kim. HotDataTrap: A Sampling-Based Hot Data Identification Scheme for Flash Memory. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12, pages 1610–1617, Trento, Italy. ACM Press, 2012. ISBN: 978-1-4503-0857-1. DOI: 10/gpn9zr.

[PD11] Dongchul Park and David H.C. Du. Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters. In *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies*. MSST '11, Denver, CO, USA. IEEE, May 2011. ISBN: 978-1-4577-0427-7. DOI: 10/dnzg5z.

[Pre20] Gil Press. 54 Predictions About The State Of Data In 2021. *Forbes*, December 2020. URL: https://www.forbes.com/sites/gilpress/2021/12/30/54-predictions-about-the-state-of-data-in-2021/ (visited on 12/08/2021).

[Pre16] Ron Pressler. Why Writing Correct Software Is Hard. pressron. July 23, 2016. URL: https://pron.github.io/posts/correctness-and-complexity (visited on 05/17/2022).

[Pro16] Human Progress. Average cost of hard drive storage per gigabyte, September 2, 2016. URL: https://www.humanprogress.org/dataset/average-cost-of-hard-drive-storage-per-gigabyte/. based on data from Statistic Brain.

[Red96] Thomas C. Redman. *Data Quality for the Information Age*. Artech House, Boston, USA, 1996. 303 pages. ISBN: 978-0-89006-883-0.

[RN10] Marko A. Rodriguez and Peter Neubauer. Constructions from Dots and Lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, August 2010. DOI: 10/cbvfdx.

[RPB+13] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, volume P-214, pages 403–420, Bonn, Germany. Gesellschaft für Informatik e.V., 2013. ISBN: 978-3-88579-608-4. URL: https://dl.gi.de/handle/20.500.12116/17334.

[SB08] Ricardo Jorge Santos and Jorge Bernardino. Real-Time Data Warehouse Loading Methodology. In *Proceedings of the 2008 International Symposium on Database Engineering & Applications*. IDEAS '08, pages 49–58, Coimbra, Portugal. ACM Press, 2008. ISBN: 978-1-60558-188-0. DOI: 10/bh7kfh.

[Sec87] ISO Central Secretary. Information processing systems — Database language — SQL. Standard ISO 9075:1987, International Organization for Standardization, Geneva, CH, June 1987.

[SS19] Chandan Sharma and Roopak Sinha. A Schema-First Formalism for Labeled Property Graph Databases: Enabling Structured Data Loading and Analytics. In *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. BDCAT '19, pages 71–80, Auckland, New Zealand. ACM Press, 2019. ISBN: 978-1-4503-7016-5. DOI: 10/hh3z.

[SRJ07] Galit Shmueli, Ralph P. Russo, and Wolfgang Jank. The BARISTA: A model for bid arrivals in online auctions. *The Annals of Applied Statistics*, 1(2), December 1, 2007. ISSN: 1932-6157. DOI: 10/fk72mp.

[SN18] Wiktor Stadnik and Ziemowit Nowak. The Impact of Web Pages' Load Time on the Conversion Rate of an E-Commerce Platform. In Leszek Borzemski, Jerzy Świątek, and Zofia Wilimowska, editors, *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology*. Volume 655, pages 336–345. Springer International Publishing, Cham, 2018. ISBN: 978-3-319-67219-9 978-3-319-67220-5. DOI: 10/hrp6.

[SFS16] Alexander Stiemer, Ilir Fetai, and Heiko Schuldt. Analyzing the Performance of Data Replication and Data Partitioning in the Cloud: the BEOWULF Approach. In *Proceedings of the 2016 IEEE International Conference on Big Data*. Big Data 2016, pages 2837–2846, Washington DC, USA. IEEE, December 2016. ISBN: 978-1-4673-9005-7. DOI: 10/gnccvp.

[SLM+01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., 2001. ISBN: 1-55860-804-4. URL: http://www.vldb.org/conf/2001/P019.pdf.

[Sto15] Michael Stonebraker. The Case for Polystores. ACM SIGMOD Blog. July 13, 2015. URL: https://wp.sigmod.org/?p=1629 (visited on 12/17/2021).

[SÇ05] Michael Stonebraker and Uğur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering*. ICDE '05, pages 2–11, Tokoyo, Japan. IEEE, 2005. ISBN: 978-0-7695-2285-2. DOI: 10/ctkd2n.

[SYF+20] Zhu Sun, Di Yu, Hui Fang, Jie Yang, Xinghua Qu, Jie Zhang, and Cong Geng. Are We Evaluating Rigorously? Benchmarking Recommendation for Reproducible Evaluation and Fair Comparison. In *Proceedings of the Fourteenth ACM Conference on Recommender Systems*. RecSys '20, pages 23–32, Virtual Event, Brazil. ACM, September 22, 2020. ISBN: 978-1-4503-7583-2. DOI: 10/gkmztq.

[TCG+17] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. Enabling Query Processing across Heterogeneous Data Models: A Survey. In *Proceedings of the 2017 IEEE International Conference on Big Data*. Big Data 2017, pages 3211–3220, Boston, MA, USA. IEEE, December 2017. ISBN: 978-1-5386-2715-0. DOI: 10/gnr5rf.

[VJ84] Yannis Vassiliou and Matthias Jarke. Query Languages—A Taxonomy. In *Human factors and interactive computer systems: Proceedings of the NYU Symposium on User Interfaces*, Human/computer interaction, pages 47–82, New York, USA. Ablex Pub. Corp, 1984. ISBN: 978-0-89391-182-9.

[VHS+20] Marco Vogt, Nils Hansen, Jan Schönholz, David Lengweiler, Isabel Geissmann, Sebastian Philipp, Alexander Stiemer, and Heiko Schuldt. Polypheny-DB: Towards Bridging the Gap Between Polystores and HTAP Systems. In Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Tim Kraska, Fusheng Wang, Gang Luo, Jun Kong, and Alevtina Dubovitskaya, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare – VLDB Workshops, Poly 2020 and DMAH 2020*, Lecture Notes in Computer Science, pages 25–36, Virtual Event. Springer International Publishing, 2020. ISBN: 978-3-030-71055-2. DOI: 10/gnxv2h.

[VLG+21] Marco Vogt, David Lengweiler, Isabel Geissmann, Nils Hansen, Marc Hennemann, Cédric Mendelin, Sebastian Philipp, and Heiko Schuldt. Polystore Systems and DBMSs: Love Marriage or Marriage of Convenience? In El Kindi Rezig, Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Tim Kraska, Fusheng Wang, Gang Luo, Jun Kong, and Alevtina Dubovitskaya, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare – VLDB Workshops, Poly 2021 and DMAH 2021*, volume 12921 of *Lecture Notes in Computer Science*, pages 65–69, Copenhagen, Denmark. Springer International Publishing, 2021. ISBN: 978-3-030-93663-1. DOI: 10/gn8qvm.

[VSC+20] Marco Vogt, Alexander Stiemer, Sein Coray, and Heiko Schuldt. Chronos: The Swiss Army Knife for Database Evaluations. In *Proceedings of the 23nd International Conference on Extending Database Technology*. EDBT 2020, pages 583–586, Virtual Event. OpenProceedings.org, April 2020. ISBN: 978-3-89318-083-7. DOI: 10/g8w5.

[VSS17] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. Icarus: Towards a Multistore Database System. In *Proceedings of the 2017 IEEE International Conference on Big Data*. Big Data 2017, pages 2490–2499, Boston, MA, USA. IEEE, December 2017. ISBN: 978-1-5386-2715-0. DOI: 10/g8w7.

[VSS18] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *Proceedings of the 2018 IEEE International Conference on Big Data*. Big Data 2018, pages 3364–3373, Seattle, WA, USA. IEEE, December 2018. ISBN: 978-1-5386-5035-6. DOI: 10/gnxv2j.

[VSB+10] Laura Cristiana Voicu, Heiko Schuldt, Yuri Breitbart, and Hans-Jörg Schek. Flexible Data Access in a Cloud Based on Freshness Requirements. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*. CLOUD 2010, pages 180–187, Miami, FL, USA. IEEE, July 2010. ISBN: 978-1-4244-8207-8. DOI: 10/fqzb48.

[WBB+17] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. The Myria Big Data Management and Analytics System and Cloud Service, 2017. URL: http://cidrdb.org/cidr2017/papers/p37-wang-cidr17.pdf (visited on 06/28/2022).

[WPW+14] Shicai Wang, Ioannis Pandis, Chao Wu, Sijin He, David Johnson, Ibrahim Emam, Florian Guitton, and Yike Guo. High dimensional biological data retrieval optimization with NoSQL technology. *BMC Genomics*, 15, S8, November 2014. ISSN: 1471-2164. DOI: 10/gb3g7k.

[Wan21] Wangde. Tech Insights — Two-phase Commit Protocol for Distributed Transactions. Alibaba Cloud Community. February 15, 2021. URL: https://www.alibabacloud.com/blog/tech-insights---two-phase-commit-protocol-for-distributed-transactions_597326 (visited on 03/13/2022).

[WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, USA, 2002. 853 pages. ISBN: 978-1-55860-508-4.

[Wes11] Matthew West. Some Types and Uses of Data Models. In *Developing High Quality Data Models*, pages 23–36. Elsevier, 2011. ISBN: 978-0-12-375106-5. DOI: 10/bphvjq.

[YGL+17] Noel Yuhanna, Mike Gualtieri, Gene Leganza, and Jun Lee. The Forrester Wave: Translytical data platforms. *Forrester Research*, 2017.

[ZSL+14] Liang Zhao, Sherif Sakr, Anna Liu, and Athman Bouguettaya. *Cloud Data Management*. Springer International Publishing, Cham, 2014. ISBN: 978-3-319-04764-5 978-3-319-04765-2. DOI: 10/hh3w.

[ZR11] Minpeng Zhu and Tore Risch. Querying Combined Cloud-Based and Relational Databases. In *Proceedings of the 2011 International Conference on Cloud and Service Computing*. CSC '11, pages 330–335, Hong Kong, China. IEEE, December 2011. ISBN: 978-1-4577-1637-9. DOI: 10/fxrh7n.

[ZK01] Thomas Zurek and Klaus Kreplin. SAP Business Information Warehouse — From Data Warehousing to an E-Business Platform. In *Proceedings of the 17th International Conference on Data Engineering*, pages 388–390, Heidelberg, Germany. IEEE Computer Society, 2001. ISBN: 978-0-7695-1001-9. DOI: 10/b7dfjz.

Contributors

Polypheny-DB and the entire Polypheny ecosystem shine thanks to many open-source contributors. It is only thanks to their contributions that we have been able to create such a mature and feature-rich system. I would especially like to thank all the students I have had the privilege of supervising over the past few years:

- Silvan Heller and Manuel Hürbin. *A Client for Dynamic Replication and Partitioning of Big Data.* Master's Project, University of Basel, August 12, 2017
- Christian Frei. *Polypheny-KV: A Transactional Key-Value Store for Polypheny-DB.* Master's Thesis, University of Basel, May 14, 2018
- Khushbu Rajendra Agrawal. *Tapmaan: Temperature-aware Data Management in Polypheny-DB.* Master's Thesis, University of Basel, May 19, 2018
- Sein Coray. *Chronos 2.0: A Generic Evaluation Framework.* Master's Project, University of Basel, June 27, 2018
- Nils Hansen. *Entwicklung eines Web UI für Polypheny-DB.* Bachelor's Project, University of Basel, April 26, 2019
- Nils Hansen. *Polypheny-DB UI: An Angular-based User Interface for Polystore Databases.* Bachelor's Thesis, University of Basel, August 29, 2019
- Nils Hansen. *Graphical management of data stores in a distributed database.* Master's Project, University of Basel, January 28, 2020
- David Lengweiler and Isabel Geissmann. *Dynamic Database Exploration.* Seminar Report, University of Basel, Seminar: Modern Human Machine Interaction (MHMI), January 31, 2020
- Jannik Jaberg and Jonas Rudin. *Project Report: Query by Gesture - Polypheny.* Seminar Report, University of Basel, Seminar: Modern Human Machine Interaction (MHMI), January 31, 2020
- David Lengweiler. *Persistent and performant schema management in Polypheny-DB.* Bachelor's Thesis, University of Basel, May 2, 2020

- Isabel Geissmann. *Explore-by-Example: Interactive data exploration in Polypheny.* Bachelor's Thesis, University of Basel, May 2, 2020
- Nils Hansen. *Array type support in a distributed database.* Master's Project, University of Basel, July 17, 2020
- Sebastian Philipp. *Using Indexes to Simplify Queries in a Polystore System.* Master's Project, University of Basel, September 7, 2020
- Marc Hennemann. *Horizontal data partitioning in a polystore system.* Master's Project, University of Basel, October 4, 2020
- Jan Schönholz. *Multimedia retrieval support for Polypheny-DB.* Bachelor's Thesis, University of Basel, December 10, 2020
- Nils Hansen. *Polypheny-MM: Adding Multimedia Support to a Polystore Database System.* Master's Thesis, University of Basel, April 4, 2021
- Marc Hennemann. *Temperature-aware Data Management in Polypheny-DB.* Master's Project, University of Basel, July 25, 2021
- Isabel Geissmann. *Materialized Views in a Polystore Database System.* Master's Project, University of Basel, August 1, 2021
- David Lengweiler. *Integrating support for the document data model in a polystore system.* Master's Project, University of Basel, August 1, 2021
- Cédric Mendelin. *Intelligent routing of queries in a polystore system.* Master's Project, University of Basel, August 15, 2021
- David Lengweiler. *Multi-Model Polystore Systems.* Master's Thesis, University of Basel, April 6, 2022
- Isabel Geissmann. *Self-adaptive Polystore Database System.* Master's Thesis, University of Basel, April 6, 2022
- Marc Hennemann. *Freshness-aware Data Management in a Polystore System.* Master's Thesis, University of Basel, May 7, 2022
- Colin Fingerlin. *An adaptive approach for learning operator costs in Polypheny-DB.* Bachelor's Thesis, University of Basel, July 27, 2022
- Nicolas Odermatt. *A stored procedure based mechanism for modifying views in Polypheny-DB.* Master's Project, University of Basel, September 22, 2022

- Flurina Fischer. *Chameleon Query Interface in Polypheny-DB*. Bachelor's Project, University of Basel, November 2, 2022
- Phaina Koncebovski. *Schema Recognition and Integration in a Polystore System using Machine Learning Methods*. Master's Thesis, University of Basel, March 8, 2023

Furthermore, I would like to thank Google and especially the three students who contributed to Polypheny as part of the Google Summer of Code 2021:

- Vishal Dalwadi. *Support for Contextual Query Language*.
- Shubham Arawkar. *Query the Blockchain*.
- Harshit Sharma. *Quality Check and Assurance*.

A very special thanks also goes to the developer of *Cottontail DB*, Ralph Gasser, who continues to develop and maintain the *Cottontail DB Adapter*, which was originally developed as part of a student project.