

Towards Polyglot Data Stores

Overview and Open Research Questions

DANIEL GLAKE*, FELIX KIEHN*, and MAREIKE SCHMIDT*, Universität Hamburg
 FABIAN PANSE and NORBERT RITTER, Universität Hamburg

Nowadays, data-intensive applications face the problem of handling heterogeneous data with sometimes mutually exclusive use cases and soft non-functional goals such as consistency and availability. Since no single platform copes everything, various stores (RDBMS, NewSQL, NoSQL) for different workloads and use-cases have been developed. However, since each store is only a specialization, this motivates progress in polyglot data management emerged new systems called Multi- and Polystores. They are trying to access different stores transparently and combine their capabilities to achieve one or multiple given use-cases. This paper describes representative real-world use cases for data-intensive applications (OLTP and OLAP). It derives a set of requirements for polyglot data stores. Subsequently, we discuss the properties of selected Multi- and Polystores and evaluate them based on given needs illustrated by three common application use cases. We classify them into functional features, query processing technique, architecture and adaptivity and reveal a lack of capabilities, especially in changing conditions tightly integration. Finally, we outline the benefits and drawbacks of the surveyed systems and propose future research directions and current challenges in this area.

CCS Concepts: • **Information systems** → **DBMS engine architectures**.

Additional Key Words and Phrases: polyglot persistence, multi-/polystore, data management, adaptivity, query processing.

1 INTRODUCTION

After decades of dominance of relational database management systems (DBMS) since the 70's, the appearance of a multitude of new data stores, coined NoSQL (Not only SQL) systems, since 2009 changed the database and information systems landscape immensely. The emergence of ever-growing amounts of (often unstructured) data - particularly in the world of web applications - overcharged the technical capabilities of monolithic relational DBMS [22, 38, 69] that tried to handle the expanding workloads by using highly specialized hardware like Field Programmable Gate Array and Graphics Processing Units [10, 15].

Even though the resulting landscape of diverse stores enables users to select an appropriate system based on their specific requirements, it poses several problems [77]:

- Even as an expert, it is hard to keep an overview of all these stores and their respective sweet spots and limitations (even if some guidelines [38] are available).
- Complex applications may have subcomponents with contradictory requirements so that using a single data persistence solution always ends up in a trade-off between them.
- As applications change and evolve, so do their requirements. These changes are often not immediately visible to the user so that the initially used data store may not remain the best fitting one over time.

The approach of *polyglot persistence* [77] aims to solve these problems by combining the benefits of several data stores (and their underlying techniques) without adopting their drawbacks and to hide the automatic coordination across these (heterogenous-)stores behind single or multiple interfaces.

*These authors contributed equally to this research.

Authors' address: Daniel Glake, daniel.glake@uni-hamburg.de; Felix Kiehn, felix.kiehn@uni-hamburg.de; Mareike Schmidt, mareike.schmidt-3@uni-hamburg.de; Fabian Panse, fabian.panse@uni-hamburg.de; Norbert Ritter, norbert.ritter@uni-hamburg.de, Department of Informatics, Universität Hamburg, Vogt-Kölln-Straße 30, 22846 Hamburg, Germany.

Of course, polyglot persistence comes with a lot of challenges because the system does not only have to orchestrate different stores based on dynamically changing conditions, but also has to deal with mismatches between different data models, and needs to translate and mediate queries.

The objective of this paper is to survey the first generation of polyglot data stores. This includes descriptions and discussions on several aspects such as architecture and query processing approach.

The main contributions of this paper can be summarized as follows:

- A description of three real-world use cases that motivate the use of polyglot data management.
- A list of functional and non-functional features that may be provided by a polyglot data store in order to solve representative real world use cases.
- A description and detailed comparison of currently existing polyglot data stores.
- An overview of open challenges in polyglot data management.

The remainder of this paper is structured as follows: In Section 2, we motivate the use of polyglot data management by different use cases, illustrating different requirements for data management. Given these requirements, we describe and discuss technical features in Section 3, leading to a range of polyglot data management kinds in Section 4. In Section 5 we survey existing systems that can be assigned to any of these system kinds and describes established polystores like BigDAWG [27] or upcoming systems like Polypheny-DB [90, 92]. In detail, we describe each system’s specialization and approach to plan, optimize, and execute queries and what kind of systems are utilized in this process. Section 7 give the complete overview of all considered systems and reveals features commonly used by most of the systems and aspects they are still missing. These missing features or problems are then discussed in Section 8 before Section 10 concludes the paper.

2 MOTIVATION

In this section, we address three different use-cases from which we illustrated relevant requirements in the industry features required in concrete data management design. We consider one use case focusing more on a transactional workload and processing (OLTP), whereas the other examines challenges in processing analytical workloads (OLAP) and another in dealing a hybrid transactional and analytical processing.

2.1 E-Commerce and Customer Management

Consider a well-known online store that targets international markets [36, 50, 98]. Such a system promotes ordinary consumer goods using text descriptions, ratings, short video and image data with non-functional requirement in order to serve purchasing services all the time. From a business perspective, user groups may be interested in changing sales volumes, prices per article or other key indicators. This context represents a well suited transactional processing (OLTP) use case, in which a set of system specifications have to be handled:

- E1. *Availability*: High availability for sales operations is a crucial requirement for e-commerce systems as downtimes lead to a loss in earnings [6] and come to a trade-off between consistency and availability. The decision can be made to occasionally accept downtimes such as customer support or article rating instead of shut-down purchasing features.
- E2. *Flexible Consistency*: On a state-of-the-art, modular e-commerce website, each component requires a different level of consistency (e.g. high consistency for payments, low consistency for product ratings).

- E3. *Read Throughput*: When navigating through the website, users request more and more articles and other data in the system. Therefore, a read-friendly distribution model (e.g. replicating a subset of data node) or a reschedule for better-performing hardware needs to be supported.
- E4. *Complex Analyses*: To enable analytical tasks such as calculating user-specific product recommendations or analyzing the company's recent business numbers, data mining and OLAP querying capabilities have to be provided.

2.2 Agent-Based Simulation and Decision Support System

Another real-world use-case considers the widely used form of a mobility simulation and digitalized cities called digital twins [19, 42, 43] such as representations of Hamburg [94] ecology systems in Africa [61]. Citizens have attributes (e.g. *velocity* and *position*) and move along a road network affected by traffic lights, intersections and other participants. Depending on the scenario's selected time-scale and spatial extension, the simulation produces an extensive set of varying simulation results. The output includes hierarchically structured spatial-temporal data and interrelationships between agents. Due to the multiplicity of results, formats and analytical routines, this simulation is a prime example for analytical processing (OLAP), where the following system properties are required:

- S1. *Write Throughput*: When executing a mobility simulation incorporating many agents such as modelled as citizens on the street, many agent and environmental states are computed in each simulation step. A scaled solution is required to analyze all these states and visualize, for example, all computed trajectories in the world. These solutions have to handle massive write processes in terms of data loaded into the simulation and computed by the simulation.
- S2. *Immutable Data*: Scientific users as the primary target group are convinced that data should never be discarded. Even if data objects are incorrect, their revision should never overwrite old data since previously published works may have used them. Instead, we need a versioning approach that enables users to selectively access individual versions.
- S3. *Support of Spatial Formats*: For a geographic-driven simulation, it is mandatory to define standard spatial formats and allow corresponding queries (e.g. spatial-joins, k-nearest neighbours).
- S4. *Real-time Analyses*: In order to display aggregated indicators as well as the movement of agents on a map, data objects are streamed through the system at runtime, when data has been changed (push-based queries).
- S5. *Complex Analysis*: In addition to simple queries and data browsing along the different dimensions [44], various kinds of complex analyses such as k-means and correlation analysis are beneficial in a scientific context. For the case of traffic simulations where graphs are used to represent the road network, graph analyzes should be possible as well.

2.3 Healthcare Data Management Systems

An often described use case for polyglot data management systems is data in healthcare as provided by the MIMIC II ("Multiparameter Intelligent Monitoring in Intensive Care II") database [29, 54, 78]. It contains detailed information on patients, laboratory and radiology results, physiological data (e.g. electrocardiograms) as well as doctor's and nurse's notes relating observations and treatments. All these different kinds of information have to be gathered and analysed in common modern hospitals.

- M1. *Support of Multiple Data Formats*: Data in this medical use case ranges from structured and unstructured data (patient data, lab results) over images and waveform data (CT and MRI images, electrocardiograms) to graph based data (relationships between patients or blood relations).

- M2. *Complex Analyses*: The analysis of images and waveform data in the case of medical imaging, requires complex operations such as Fourier- and Wavelet-Transformations as well as further image analysis capabilities (e.g. to find regions containing abnormal tissues). Furthermore, text search and graph algorithms may be required (e.g. in order to find similar examination reports or to analyse relationships between patients).
- M3. *Real-time Support*: In order to keep patients in an intensive care unit under continuous surveillance and detect, for example, abnormal heart rhythms in time, real-time monitoring is essential.
- M4. *Availability*: In case of emergencies, it is necessary that requested data such as the patient's allergies or blood type are available at any time.
- M5. *Consistency*: The treatment of patients always have to follow the latest diagnosis. Accessing outdated data might even endanger the patients' lives. Therefore, a high level of consistency is needed for specific parts of the medical data.
- M6. *Security and Privacy*: In a healthcare data management system, a huge amount of personal data is collected and it is highly important, that the data is protected from illegal access. Additionally, it should be possible to render data anonymous in order to use it for research purposes.

3 CONCLUDED REQUIREMENTS AND SYSTEM CHARACTERISTICS

From the use cases in Section 2, we can derive a variety of requirements data management systems have to fulfil in order to provide a sufficient persistence solution. In this section, we assemble a list of these concluded requirements and discuss them in more detail. Here we distinguish between requirements that are directly concluded from the use cases and requirements that are induced from the direct ones.

3.1 Explicit Requirements

From our representative use cases, we derive seven essential, indispensable *functional* features, which needs to be support by a future-proof polyglot data store.

Modeling Power: The most important functional feature is the range of supported data models. This usually includes relational and popular NoSQL data models, such as JSON documents and property graphs, but can also include more specialized ones, such as spatio-temporal, or array-based models (S3, M1). Furthermore, system properties such as availability (E1, M4), flexible consistency (E2) and versioning (S2) depend on the provided data models and systems.

Query Expressiveness: Almost equally important as the range of data models are the expressiveness and the variety of the provided query languages. It has to be distinguished between *data definition* (DDL) and *data manipulation* (DML). Concerning DDL, it is of interest whether or not the system supports the definition of an explicit schema and if integrity constraints, complex domains and proactive concepts (e.g. trigger) are provided. One significant characteristic of a DML is whether the language supports complex *set-based* queries, *point access* to single objects, or allowing to *navigate paths* between linked objects. Apart from classical relational operators, complex queries may include non-relational operators such as graph (E4, M2), array (M2) or spatio-temporal ones (S5). Moreover, some query languages support recursive queries or user defined functions.

Push-based Access: While traditional systems focus on pull-based data access, newer systems, such as Meteor [71], also provide push-based queries which are particularly suitable for frequently changing data (S4, 3).

Data Security/Privacy: For some applications (M6), the support of special mechanisms for data security and privacy, such as encryption [79] and anonymization [96] techniques, is essential and has to be supported.

Flexible Adaption: In order to support the aforementioned features, it is vital to enable the system to find an underlying data store landscape that meets the requirements formulated by the user. Thus, it is necessary that the user has the possibility to annotate individual parts of the data schema with special service level agreements (SLAs). A particular case where such annotations are indispensable is described in the property E2 of Section 2.1. In addition to providing annotations and since the underlying conditions, such as user requirements, workloads, or data characteristics, can change over time, a polyglot data store must be able to adapt flexibly at runtime. There are three types of adaptation:

(1) By changing the parameter setting and/or hardware configuration of a particular data store (e.g. in/decrease of buffer size or quorum-configs). For example, to achieve a higher read- or write-throughput (E3, S1), in which only a subset of data (E2) is affected, to prevent constraint violations. (2) By performing *data or function shipping* between the underlying data stores and/or the mediator in order to execute domain-specific operations (S3, M1, M2) or to *split* workload across the data stores to run operations concurrently (further details in Section 3.2). (3) By changing the system's store-topology (e.g. by starting and interconnecting a new MongoDB instance) and distributing a subset of data according to a well-known distribution model, such as master-slave in order to increase the read throughput (E3) or a multi-master style to handle heavy write-workloads (S1).

Automatic Adaption: To relieve users, the polyglot system should not only be flexible to adapt, but also execute necessary adaptations automatically. The adaptation of a system can be triggered by a wide variety of possible reasons. In contrast to manually triggered changes (*offline*), systems can also react proactively (*online*) by comparing the current workload against specified constraints (SLAs) [77] or by using a prediction model [92].

Transparency: The complex processes of adaption and data (re)distribution should be hidden to users by providing *location, concurrency, distribution* and *mobility* transparency [74, 88]. At the same time, physical and logical data independence should be provided. While physical independence is mainly guaranteed by the underlying stores, logical independence must be ensured by the polyglot system.

3.2 Indirect Requirements

The seven features described in the previous section ensure that the major and unique selling point of polyglot data stores is their transparent and integrated access to multiple heterogeneous data stores without decreasing the overall query performance or – better – even increasing it. Finding the best query execution plan (QEP) in this setup, however, comes with a variety of challenges. Many of these challenges are similar to those well-known from the area of information integration [26]. A big difference, however, is that in a polyglot data store, the data is not only read, but also written. Another difference is that we usually have more control over the individual data stores and know exactly what data is stored in which store and in what form.

Operator Placement: First of all, the query has to be decomposed into sub-queries (query-splitting) that are executed in the underlying stores. This process is based on several conditions such as the query capabilities of these stores (e.g. a spatial join needs to be pushed down to a geo-database (S3)) and the current distribution of data amongst them. Second, the system does not only have to be able to reorder the operations of a query but also have to decide which operations can be pushed down into an associated store. Furthermore, the system has to determine when to move data temporarily from one store to another (data-shipping) in order to perform an operation and if it is possible to replace the functionality of one store by a set of completely different operations of another store (function-shipping, query rewriting). Sometimes, even a permanent migration of data might be the best choice. In this case, it has to be decided when to trigger an expensive migration process and how to perform it (e.g. eager/lazy, online/offline) [86]. Currently, there exists multiple heuristics to address the problem of optimal operator placement [17, 58].

Cross-Store Joins: As soon as all sub-queries return the associated results, it is inevitable to combine them in a higher-level system layer using the best suited operation (e.g. bind-join, hash-join, or skew-join [46]). This combination is a non-trivial process that requires some kind of internal data model such as the relational model, associative arrays, or a hybrid model (e.g. combining relational and JSON). Via the internal data model, it is possible to transform data residing in one store transitively into the model of any other underlying data store while preserving extensibility.

Cost Models: Due to the additional operations described above, the comparison of QEPs becomes more complex and has to be based on location as well as provided functionality. Dynamic as well as cost-based approaches that reduce, for instance, data movement or operations within higher-level layers can be used.

Semantic Conflicts: Apart from problems in planning and optimizing a query, querying data across different data stores leads to a main integration question: How can semantic completeness be achieved by providing the user with the set of all operations available in the underlying stores? Every data store comes potentially with a set of slightly different interpretation of an operation [64]. This causes the problem in mapping these inconsistencies such as in relation to specific data types and missing values (e.g. the use of null values or the discrimination between null [45] and unknown or the exception process when data is not available or interpreted as default) so that it has to be considered if the overall data store should keep different semantics, use a predefined semantic or allow the user to define the applied semantic on a per query basis.

4 SYSTEM TYPES

Integrating different data types, models, and functions is a challenging research field. This section gives a general idea of the different kinds of systems that tackle these problems.

In their work, Lu and Holubová [65] presented two categories of systems. The *multi-model databases* follow the approach to integrate varying data models into one system (e.g. ArangoDB or OrientDB), whereas *multi-modal systems* organize data not by model but by domain and provide access to domain-specific data types such as speech, images, videos or handwritten text.

Another promising approach is to integrate several different SQL, NoSQL, NewSQL [27], or Stream-Processing-Systems into a mediation system, e.g. the mediator-wrapper approach [20]. Underlying data models are not integrated into a single database engine but are handled by selecting suitable data stores if applicable. Based on the work of Tan et al. [87], we distinguish between four different system types: Federated Systems (Fig. 1c), Polylingual Systems (Fig. 1d), Multistore Systems (Fig. 1e) and Polystore Systems (Fig. 1f). The systems we analyzed in this section were classified as one of these four architectures.

The system types differ in the composition of their underlying data stores and the number of query languages/interfaces they offer, as depicted in Figure 1. Federated and Polylingual Systems use a homogeneous set of data stores beneath their mediation layer, whereas Multi- and Polystore systems rely on heterogeneous ones. Furthermore, Federated and Multistore Systems only offer one query language/interface, while Polylingual and Polystore Systems provide many interfaces. In [87], the authors refer to the currently described Polylingual Systems as *polyglot systems*. To avoid confusion with the term *Polyglot Persistence*, which was already introduced by Sadalage and Fowler [77], we renamed the term *polyglot systems* to *polylingual systems*.

Poly- and Multistores provide transparent access to a set of (interconnected) data sources that reside in a static or dynamic topology. This can be accomplished by one or few query interfaces using a virtual global schema mapped to local schemas via views [33, 35, 48, 62] or tuple-generating-dependencies [26].

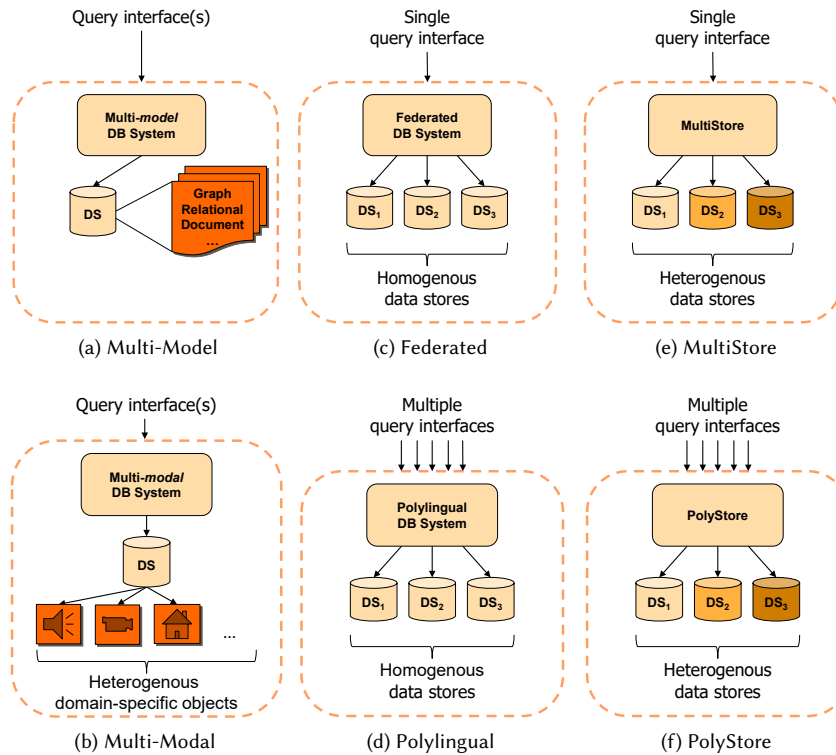


Fig. 1. System Types (Based on: Tan et al. [87])

Furthermore, the architecture of these systems can be classified according to the CAP theorem [41]. Since they can choose and set up their underlying data stores according to the user's current needs, they can theoretically be CA, CP and AP at the same time (never all at once for the same application). Finally, a system's ability to extend to new data stores and provide a dedicated framework/API for integrating them is of utmost importance for its adaptability.

Two other solution families are similar to the *polystore/multistore* concept in that heterogeneous stores are to be accessed. Despite this, they differ in the data sources' scope or the access mechanism. The so-called semantic data lake represents the first kind of solution. They provide uniform access to multiform data but do not consider any data movement across the underlying systems or duplication of data in multiple stores. They are simplifying the mediation to semantic data lakes, often utilizing *Global-as-View* mapping and technologies from the semantic web.

The second kind of system is represented by the solutions mapping relational databases to RDF [82], and Ontology-Based Data Access over relational databases [95], such as Stardog, Ontop, Morph, Ultrawrap, Mastro. These solutions do not consider querying large-scale data sources, e.g., HDFS or NoSQL stores, but integrating multi-variety data from decoupled and remote sources, e.g., from OpenData portals.

The most significant difference to the *polystore/multistores* considered in this paper is the task to find which store or combination of stores answers best a given query (transactional, analytical or hybrid) or which store to move all/part of the data to.

	<i>multistore</i>	<i>polystore</i>
<i>loosly-coupled</i>	PolyBase [25] BigIntegrator [99] FORWARD [73] Apache Drill [8] Apache Calcite [12] TATOOINE [13] QoX [81] QUEPA [67] Odyssey [47] DBMS+ (Cyclops) [63]	Myria [93]
<i>tightly-coupled</i>	RHEEM [3] MuSQL [40] AWESOME [23, 24] HadoopDB [1]	ESTOCADA [4] Polypheny-DB [92]
<i>hybrid</i>	CloudMdsQL [55] SparkSQL [9]	BigDAWG [27]

Table 1. Overview of existing systems for polyglot data management classified as *multi-* or *polystore* tightly utilizing systems for processing or as data integration solution. The stores we discuss and compare in detail are in bold. The italicized systems are out of the scope of this paper and were not discussed further.

5 ANALYZED SYSTEMS

This section introduces a set of different and representative polyglot managed data stores with their functionalities and special features. We selected these systems for their representative Poly-/Multistore approach and general availability, focusing on a differentiated perspective to support multiple use cases. Due to the variety of systems, this survey can provide extensive details of the concepts in terms of query interfaces, query planning, execution, and migration. Other known systems such as [93], HadoopDB [1] and SparkSQL[9] are beyond the scope of this paper. Table 1 shows an the considered systems and their classification as *Multi-* and *Polystore*, respectively. *Loosely-coupled* systems correspond to a network of often autonomous data stores where the mediator has read rights but no rights to write data or reconfigure the individual stores, able to migrate data sets between them. In general, they bear a strong resemblance to virtual integration systems. In contrast, *tightly-coupled* systems were often designed from the beginning as one overall system, capable of applying writes in order to combine multiple processing features instead of simply performing integration of stored data into a common global schema. This leads to a set of individual stores for which data migration is possible and whose individual stores are typically not autonomous so that the mediator can adapt the configurations to actual requirements. Hybrid systems are those in which some stores are tightly coupled, and others are loosely coupled.

5.1 CloudMdsQL

The *CloudMdsQL* multistore system¹ [55] aims at providing a powerful functional SQL-like language as part of the *CoherentPaaS* project [21] and implemented in the *LeanXScale* system [57]. The *CloudMdsQL* system has been developed as an abstraction layer to retrieve data from different stores, keeping the underlying store and query semantics. The system compiles queries into a relational query framework, in which each sub-queries contains the native database query to select the source data. *LeanXScale* executes *CloudMdsQL* query distributed, transforming the result of sub-query into an intermediate table, partitioning them to apply a distributed processing. Figure 2 show the conceptual design of

¹Cloud Multidastore Query Language

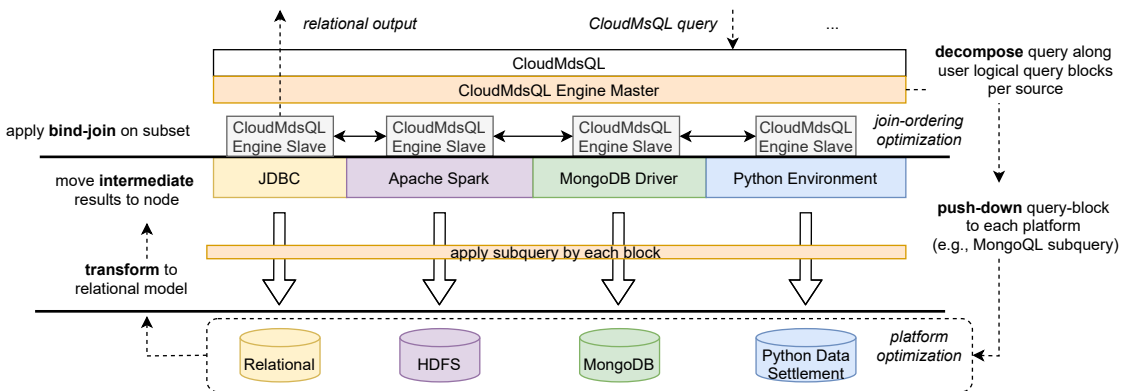


Fig. 2. Conceptual architecture of CloudMdsQL, adopted from [55]

the CloudMdsQL system. The input is a pre-formulated CloudMdsQL query, in which each embedded block already describes the decomposition to access the correct data store. Nodes responsible for the accessed data store execute each native subquery (e.g. user-defined MongoQL query to retrieve documents in MongoDB). The system merges partial results by transforming each result into an intermediate relational format, joining each transformed result from sub-queries by utilising a *bind-join*.

CloudMdsQL provides a familiar query interface to achieve semantic completeness of all associated stores by defining sub-queries with the native data store language (e.g. MongoDB statements or SQL). The base language is SQL oriented and additionally provides Python as an integrated connector to retrieve or transform queried results as an intermediate step in the plan. The base language contains named table expressions, which are wrapped into functional calls for the wrapper, translating calls to the native API of the wrapper without an intermediate integration step. *CloudMdsQL* provides a minimal set of types in order to ensure the most used types in data stores. Including scalar (integer, float, string, binary and timestamps) composite types (array and associative array), complemented by arithmetic, concatenation and access operations.

The language itself is much more restrictive, making the language more practical. When using different native query languages, these sub-queries have to be defined in the context of a so call table named expression, transforming the result into an intermediate table format. In contrast to these native queries, *CloudMdsQL* provides a language extension to incorporate distributed processing systems such as Apache Spark. The extension provides map-, filter- and reduce (MFR) operators applicable for big-data ad-hoc queries. MFR statements consist of a sequence of operating instructions in which the system incorporates datasets by transforming results into a Resilient Distributed Dataset (RDD as the Apache Spark working unit). Each of the MFR operations applies a transformation on the tuples of the dataset and produces a new dataset for subsequently part of the complete *CloudMdsQL* query.

The system applies two specific optimisations for optimal execution of all these different operations and all these blocks of native queries. For efficient merge of two results *A* and *B* from sub-queries, the system applies a bind-join reformulation. In contrast to an equality *JOIN ON* operation, the bind-join allows retrieving data from heterogeneous sources, as long as the acquired source provides a filtering technique. Therefore, it executes and loads one side *B* thoroughly to subsequently select a matching partner by filtering for a key relationship condition on side *A*. However, since the complete side *B* have to be loaded first, the system estimates the cardinality of each result by using the cost

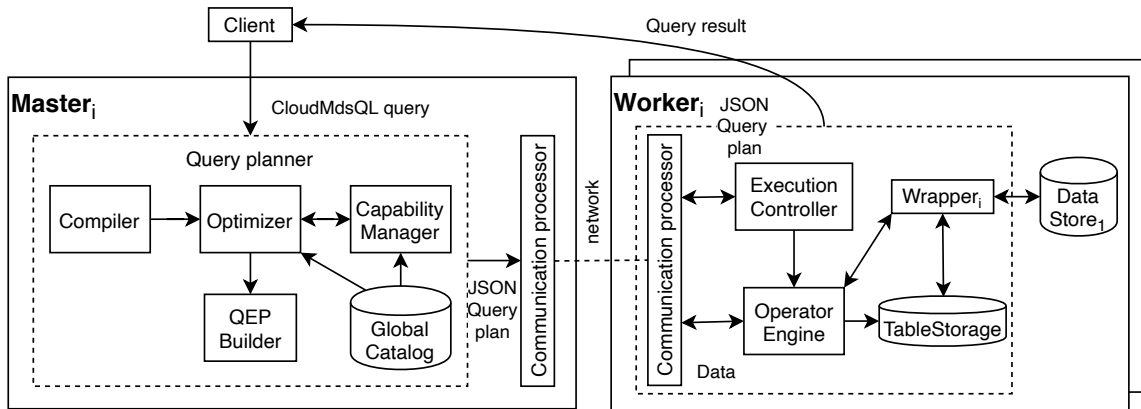


Fig. 3. Distributed architecture of the CloudMdsQL engine from [55]

model of the native's store when available. Otherwise, when no cost models are available, the system looks for exceeding a threshold of join keys to fall back to a hash-join. The other optimisation considers the MFR rewriting to bring filter operations in MFR statements forward as much as possible. Therefore, the query planner controls the planning by three rules: (1) A name substitution replaces column names in predicates of MFR's filter operations with the data reference of the MFR mapping. (2) A *reduce-filter* switch rule changes the order of reduce and filter operations to shrink data object for a reduction tasks, whereas (3) a *map-filter* switch rule change the order for map operations analogously. Each of these rules are applied in the scope of the MFR statements for the execution.

To execute *CloudMdsQL* queries, a query engine called LeanXScale distributed the processing work within a master-worker environment. LeanXScale is a Hybrid Transactional Analytical Processing, bringing OLAP and OLTP together and consisting of a planning controller node and multiple workers, directly linked to exchange query plans and other data. Worker nodes execute the query, and controller nodes plan the query. Generated execution plans are split into multiple sub-plans, each assigned to a respective worker node, collocating the underlying data store and exchanging results. Since the *CloudMdsQL* approach converts sub-queries into intermediate tables, relational operations are available, improving the runtime of LeanXScale by partitioning the base tables for the scan operations among all worker nodes. The partitioning step respects stateful operations such as group-by and reduce.

Planning a query by a controller node produces a JSON-based operator plan transferable to assigned nodes. The plan is an acyclic graph consisting of relational operators in which the leave nodes are named table result references coming from the native sub-queries. These table results can be referenced anywhere in the plan and used as input for other operators. The uppermost operator selects the first worker node to use. Figure 3 shows the technical design for this distributed execution.

Cost model the bind-join are collected by keeping database-specific features under API contract in the wrapper, besides periodically updated database statistics and estimations of sub-queries. *CloudMdsQL* considers reorderings and improvements in the scope of these subsets of operations and not native queries. As to the lack of cost models for some systems, the *CloudMdsQL* engine also allows user-defined cost models and default costs when nothing can be collected.

CloudMdsQL is used in the CoherentPaaS [21] platform to retrieve and process data from heterogeneous data stores for analytical purposes and can be used in conjunction with the distributed LeanXScale query engine, accepting

CloudMdsQL queries. Evaluations were made in [57] showing a significant impact of partitioning results onto multiple workers and merging them back using the bind-join approach.

5.2 BigIntegrator

The focus of the *BigIntegrator* system [99] is to access relational DBMS and BigTable databases, especially cloud-based NoSQL stores. To achieve this, *BigIntegrator* introduces a query interface supporting a SQL-like query language. Incoming queries and generated QEPs contain relational operations with SQL and BigTable database function calls of GQL². Therefore, GQL represents a subset of SQL and sacrifices the full expressiveness of SQL in favour of scalability since only basic filter (selection) predicates are supported. To compensate these restrictions, more complex operations as *joins* or the *like* operator are provided by *BigIntegrator*. *BigIntegrator* tries to *pushing down* as many operations as possible, handled by corresponding stores in the post-query step whose remaining not supported operations are fulfilled by the query language-specific *absorber* plugins, as part of each wrapper. The query engine accesses each *absorber* as a plugin in the wrapper and replaces the source with the respective store access. This access results in a QEP, containing both relational operators and calls to the absorber plugin (*API-calling* module), in which the system moves as many store-specific filter predicates into the plugin implementation. In the end, a *finaliser* plugin receives the plans and executes them. In contrast to other systems such as *BigDAWG*, *BigIntegrator* only works with the BigTable systems and cannot be integrated with other NoSQL systems or models.

BigIntegrator is designating a plugin architecture with expressiveness using GQL for a more straightforward integration of different data stores. A new system needs to provide or use an existing *absorber*- and *finaliser* plugin to grant access to query capabilities of the underlying store. Therefore, *BigIntegrator* transforms the input query into a Datalog [28] query, which can contain both source predicates and non-source predicates (NSPs). The absorber manager takes the Datalog query, which calls and, for each source predicate referenced in the query, calls the corresponding absorber of its wrapper. The manager collects referencing source predicates and replaces them with the native access filters. This abstraction allows decoupled filters and other predicates to produce an algebra expression containing the access to stores and NSPs. Each access is passed toward the call on the corresponding finaliser of its wrapper, which transforms the access into interface function calls.

5.3 ESTOCADA

ESTOCADA [4] is a multistore system to optimise the performance of applications by reducing the cross model problem to a single model problem. It queries data fragments across heterogeneous stores and uses view-based query rewriting with a decoupled result integration.

ESTOCADA supports data in the format as JSON, XML, key-value, graph and nested relations or full-text. Fragments of datasets can be consists of multiple data models and are transformed into an internal pivot model to leverage development and query processing. The internal pivot model consists of a relational model expanded by constraints, specifying the application related data model. It allows representing relational queries expressed in multiple supported languages, splitting a single query into multiple ones using a query tree block approach similar to 5.1. Combinations of subqueries are formulated by an own query interface called *QBT^{XM}*, in which each matching native query language is expressed in an individual tree block.

²Google BigTable Query Language

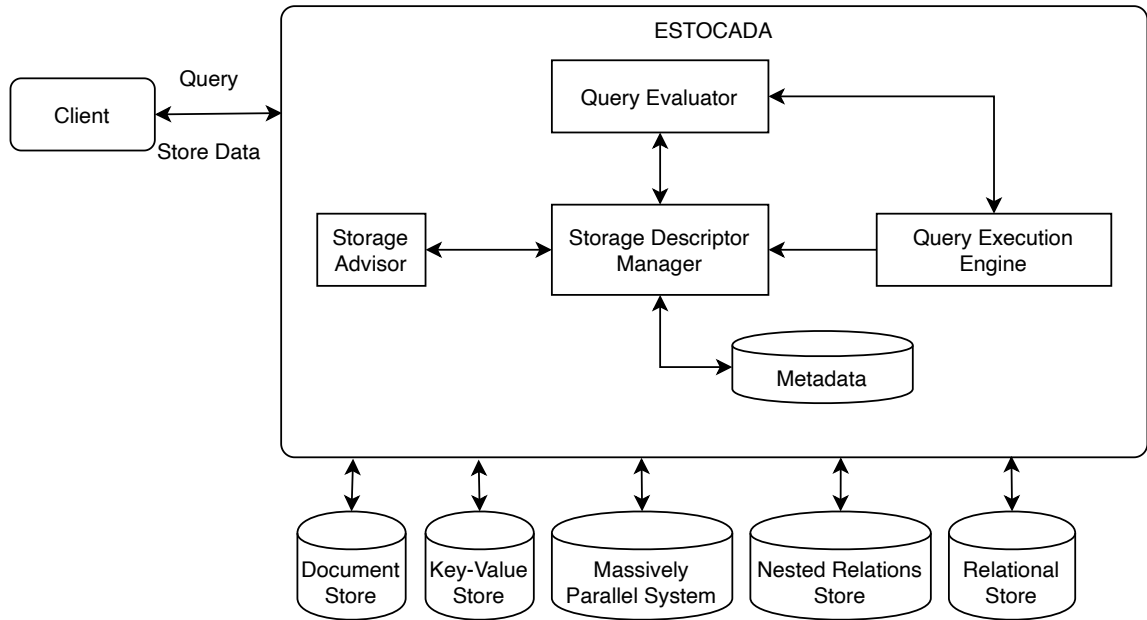


Fig. 4. The architecture of the ESTOCADA system from [4]

The architecture of ESTOCADA consists of two main distinct components. These can be any NoSQL store, a key-value store, a document store, one for nested relations or a relational one. The Storage Advisor module splits each data set into fragments and chooses the backend system for each fragment based on the underlying data model, where to store them accordingly. ESTOCADA decides fragmentation by the amount of application workload, driven according to the exact amount of data accessed as parameterised access pattern or via a heuristic on top of the underlying data model.

The mapping of data to stores is handled through a storage descriptor. These descriptors specify what is contained in the fragment and where it can be found. The containment is defined by the query in the target primary data model, where the source is designated by given a source schema for the database where the outgoing data set is stored. Thus wrapper functionality and mapping information are saved for each fragment, decoupling the system from using extra wrapper components for each tied underlying store. This approach allows the integration of new systems by providing individual store descriptors, respecting the constraints of the new system.

The query interface of this system provides an integration language, coined QBT^{XM} , which is based on the Query Block Trees of System R and provides the properties to express each block in a different data store's matching query language and data model. To achieve the best possible performance from the available data stores, ESTOCADA automatically distributes and partitions the data across the different data stores, which are entirely under its control and hence do not have any autonomy. Therefore, it is a tightly-coupled multistore system.

Reformulation of relational queries (pivot) on views for each data store, holding constraints. ESTOCADA solves the reformulation problem using the Provenance-Aware C&B algorithm, respecting conditions in relational algebra.

ESTOCADA's generic model approach and their architecture are motivated by large-scale e-Commerce and open data warehousing systems for digital cities, dealing with the requirements in Section 2.1.

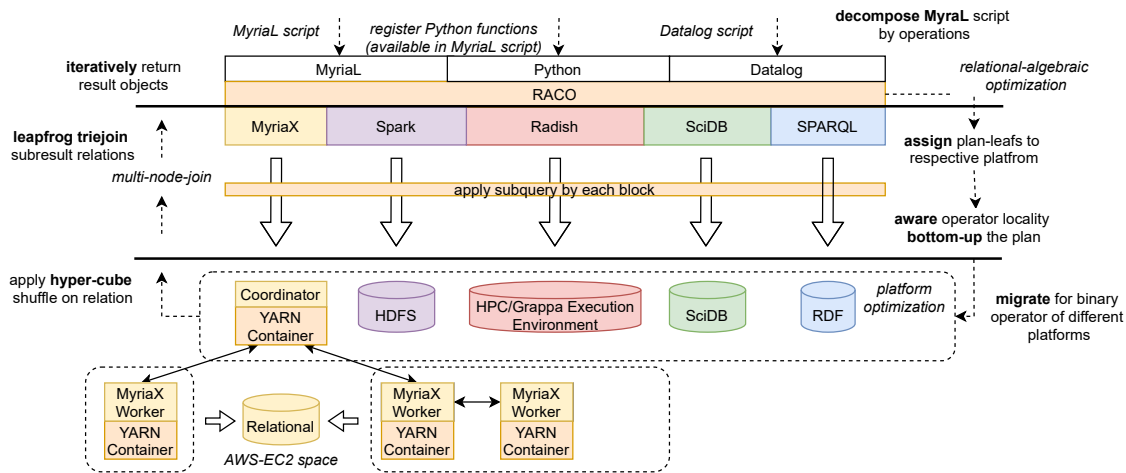


Fig. 5. Conceptual architecture of the Myria Big-Data Management System, adopted from [93]

5.4 Myria

Myria [93] is a federated data-analytics system with an imperative-declarative hybrid language called *MyriaL* and optional support for Datalog [28]. It facilitates the expression of complex data analytic tasks and wraps each declarative statement with imperative constructs, such as variable assignments and iterations. Myria implements support for user-defined functions and aggregates exposed by a Python API for full acceptance. Myria uses shared-nothing architecture and enhances the query engine (MyriaX) with iterative processing focusing on horizontal elasticity with scale-in and scale-out. It provides a data-ingestion for HDFS and multiple cloud storage, using YARN container [89] in a distributed setting.

Myria uses a relational-algebraic optimisation with a dedicated relational algebra compiler called (RACO). RACO is a relational, local-aware, algebraic optimiser using a rule-based approach and extends it by imperative constructs to capture the semantics of array, graph and key-value data. Figure 5 shows the architecture containing the outer view top-down. Inputs in MyriaL processed by RACO can produce federated execution plans considering computation and data movement across the subsystems. Extensibility of to new backend system can be provided by mapping the relational operators to the new API, AST or supported query language concepts. The database administrators must implement their functions to retrieve metadata for newly coupled backend systems. Therefore the new system needs to provide rewriting rules for the RACO compiler.

RACO extends the model in multiple ways. It enables iterative processing, which is a recurrent requirement in performing machine-learning- and graph-analytics task. Therefore the RACO supports a *do-while* loop, in which the content is executed each time and checking a loop boundary. The termination conditions is synchronized by a relational sub-query, those result contains exactly *one* tuple with a *boolean* attribute. The other loop extension is a *do-until* loop, in which asynchronous processes are enabled. Moreover, RACO integrates a flat map operator [18], in which non-1NF values are formed into multiple ones and the support of a so-called *stateful apply* operator. This *stateful apply* is used to provide window function, making sliding window operations on datasets available.

Myria's query execution is based on RACO and translates the inputs formulated by MyriaL into a logical algebra. Query plans are in the form of *operator-graphs* and can contain cycles for the iterative extensions. A query consists of

multiple individual query fragments, in which a subset of operators within the graph are grouped and executed by an individual shared-worker thread. The query federation assigns each leaf of the operator tree to those platforms where the data exist. Minimizing data movement is an open challenge and considered by integrating a data-movement operator into the plan when data lies on a different platform for binary operation.

This operator set consists of mentioned loop extensions and the known relational operators such as aggregates and joins. Especially joins are supported in MyriaX by using *HyperCube* [11] and *Shares* [2] data distribution algorithm, which is leveraged in a decoupled operator called *Tributary Join*. These multi-joint algorithm tries to process the conjunctive query in one communication round within the distributed setting. It builds a balance of joins among the servers to avail from their parallel execution and reduces the amount of additional data-movement operations across engine boundaries.

Intermediate query results are exported as CSV tabular data and stored in HDFS. This file approach offers higher connectivity for external systems but results in lower transfer performance when coupled engine provides do not provide task-optimized imports. Since the exponential growth of implementing custom operators for each engine combination, MyriaX provides a data movement component called PipeGen [51] in order to facilitate the import and export functions for CSV. PipeGen is an extension for DBMS to speed up the import and export, using Apache Arrow [7] with their network protocol and compressed message format.

The Myria big-data management project is successfully used in multiple domains, such as natural language processing, neuroscience, astronomy and oceanography.

5.5 Polypheny-DB

Polypheny-DB is a research system designed by the database group at the University of Basel from a vision in [92] towards a first version to support hybrid transactional and analytical processing (HTAP) workload, a combination of OLTP and OLAP requirements. The project aims to create a self-adaptive polystore (in a cloud environment) that provides access to a heterogeneous set of data while incorporating the characteristics of the system's workload to balance out and improve data distribution and underlying data stores choices. In contrast to other systems presented and discussed in this paper thus far, *Polypheny-DB* is - in its current state - a vision and not a fully functional system [92]. Nonetheless, several smaller projects exist in the research group's portfolio intended to fulfil the roles of components of the *Polypheny-DB* system. In the following, we present the vision of *Polypheny-DB* and the components the developers want to use in their endeavour.

A major design decision for *Polypheny-DB* is its distinction between two data management layers (Figure 6): (1) A global data partitioning and replication layer in the Cloud, which is based on user requirements (e.g. consistency level or availability quota) and the resource optimization of Cloud providers. (2) A local level with polystores located in individual Cloud data centres. These stores are envisioned to leverage the strengths and sweet spots of different data models, data stores, and storage media.

The data partitioning is supposed to be handled (1) explicitly by incorporating user-defined flags on attributes or (2) implicitly by analysing the system's workloads. The partitioning is built on previous work on Cumulus [31], supposed to be handled by QuAD [32], an adaptive quorum-based replication protocol that dynamically selects an optimal quorum configuration based on sites' load and network latency. While QuAD can adapt its quorum strategies by considering the site's properties and therefore outperforms static approaches, it cannot adapt to user-defined requirements (SLAs) or the system's actual workload.

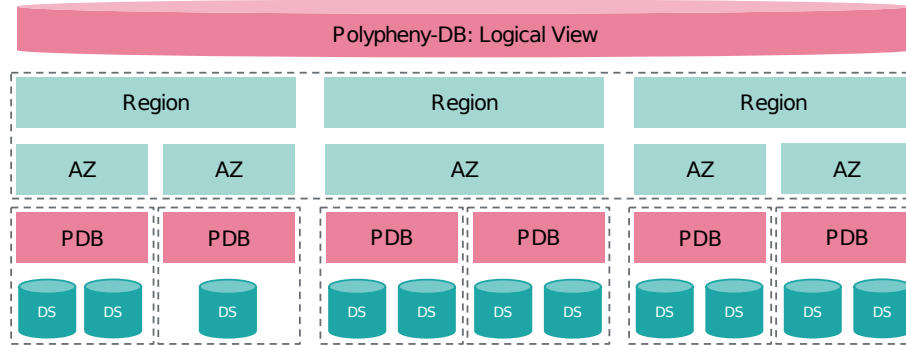


Fig. 6. *Polypheny-DB*'s layered architecture consisting of Regions, Availability Zones (AZ) and *Polypheny-DB* instances with their underlying datastores [92]

Polypheny-DB utilises data partitioning and replication by employing a cost model derived from an extended version BEOWULF [83] cost model. The extensions include the specification of user-defined requirements as service-level objectives (SLO)s, compiled from SLA agreements. Regarding the translation from SLAs to SLOs, *Polypheny-DB*'s developer refers to PolarDBMS, which included an approach for this [30].

*Polypheny-DB*s query interface incorporates multiple query languages and data models. Besides a CRUD interface to point-access, it includes at least a minimal SQL dialect and OpenCypher, an open query language for property graphs, built upon subsystem Icarus [91]. In [92], the authors aim to extend Icarus's ability to connect to different SQL based, relational databases by adding support for document and graph stores. For the internal data representation, incoming queries are translated into an algebraic tree to use mathematical models unifying relations, graphs and matrices like associative arrays. In order for *Polypheny-DB* to provide ACID guarantees for transactions, the system will use a two-phase commit protocol for coordination, knowing the danger of deadlocking in case of failures.

On the local polystore level *Polypheny-DB* dynamically distribute and replicate data to add or remove datastores if needed. Besides the difficulties in dealing with the automatic management of stores, a sound way to estimate current and future workloads have to be implemented to change the system accordingly. For this, workload predictions from Cumulus and BEOWULF [83] are intended to be incorporated. For its query planning needs, the system builds on efforts done on Icarus [91]. Icarus's query planning component estimates the execution time of an incoming query (1) by analysing its used operations (2) and comparing its structure, entities, functions and operators to similar queries already encountered by the system. The estimation defines an upper time limit for further analysis and the generation of queries execution plans³, which are assigned a cost. The plan with the lowest cost is then given to the query execution engine.

In addition to the polystore functionalities, the developers of *Polypheny-DB* want to include additional functionalities like temporal data management based on their system ARCTIC [16], an index structure for searching versions of data items by archiving queries, and multimedia retrieval provided by their system ADAM_{pro} [39] in conjunction with the multimedia retrieval engine Vitivr [76].

Conclusively, *Polypheny-DB* is one of the few systems aiming at a self-adaptive ability but is very vague in how that is to be achieved. The vision paper's primary component is the cost model, which determines almost every decision the system has to make. Considering all the variables and inputs (SLAs/SLOs, current/future workloads, per-Transaction

³Methods for generating the plans include: query-splitting, and execution on different datastores, data migration

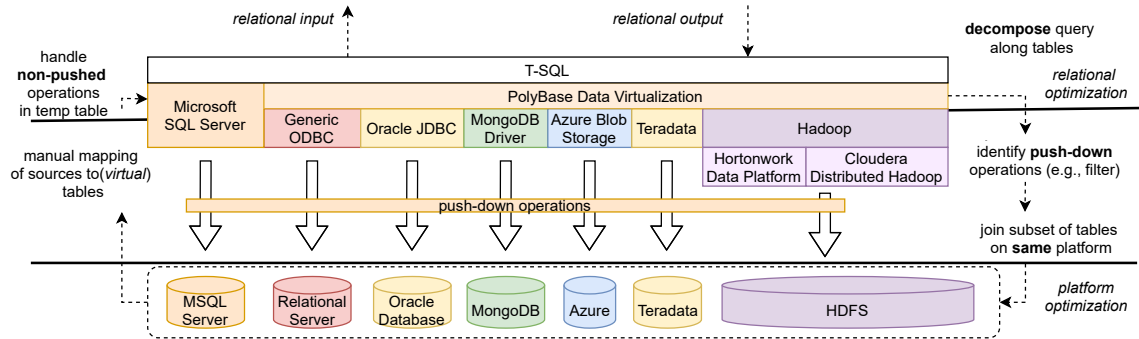


Fig. 7. *PolyBase* conceptual architecture adopted from [25]

overwrites, global and local data partitioning/replication), the cost function's scope becomes quite ambitious and poses one of the most significant difficulties. Additionally, the amount of different systems which are supposed to be included as components within *Polypheny-DB* is very sizable and can be a problem on its own if everything is combined into one. In addition, a system like *Icarus* has to be extended and cannot be incorporated as-is.

5.6 PolyBase

As part of Microsoft's efforts to improve data virtualisation for SQL Server, Microsoft introduces *PolyBase* [25] as an active component to retrieve data from a remote source and to shift access workload into an acquired system of the polyglot setup. Figure 7 shows the conceptual design of the data virtualization. Users can query data directly from SQL Server, MongoDB, Teradata, Oracle, Hadoop clusters, and Cosmos DB using T-SQL without installing special client connection software. The system uses a generic Open Database Connectivity (ODBC) standard to connect to additional providers via third-party ODBC drivers. The main approach is to keep different data sets in a relational format. Therefore, the user must provide a mapping via tables from the inner *PolyBase* scope to the external data provider. For example, if data from MongoDB is to be queried using T-SQL, a so-called external table must be created manually in *PolyBase*, describing the mapping of the documents (possibly hierarchical) structure to the relational format. The manual mapping into columns with corresponding types must be explicitly defined for fields from the documents. The main goal of *PolyBase* is to provide a solution to two use cases: (1) PDW requires data from Hadoop and returns the result to the user/application or (2) PDW requires data from Hadoop and materialises the result as an output file in Hadoop for later usage by PDW or MapReduce.

PDW basic architecture comprises a control node and a variable amount of worker nodes. Figure 8b show the technical design of *PolyBase* with its distinct PDW and interceptor component append to the SQL server.

For this task, HDFS data can be referenced in *PolyBase* as external tables, which makes the correspondence with the HDFS file on the Hadoop cluster possible. By providing an external relational *view* as well as their current execution costs, PDW allows manipulating both the native and external tables expressed by location-transparent SQL queries over a GAV schema. These are executed according to a distributed SQL execution plan, in which the query optimiser decides to *split* the query and to push SQL operators to the external store by building jobs for projection and selection on the external tables or when asymmetric join is applied on two external tables.

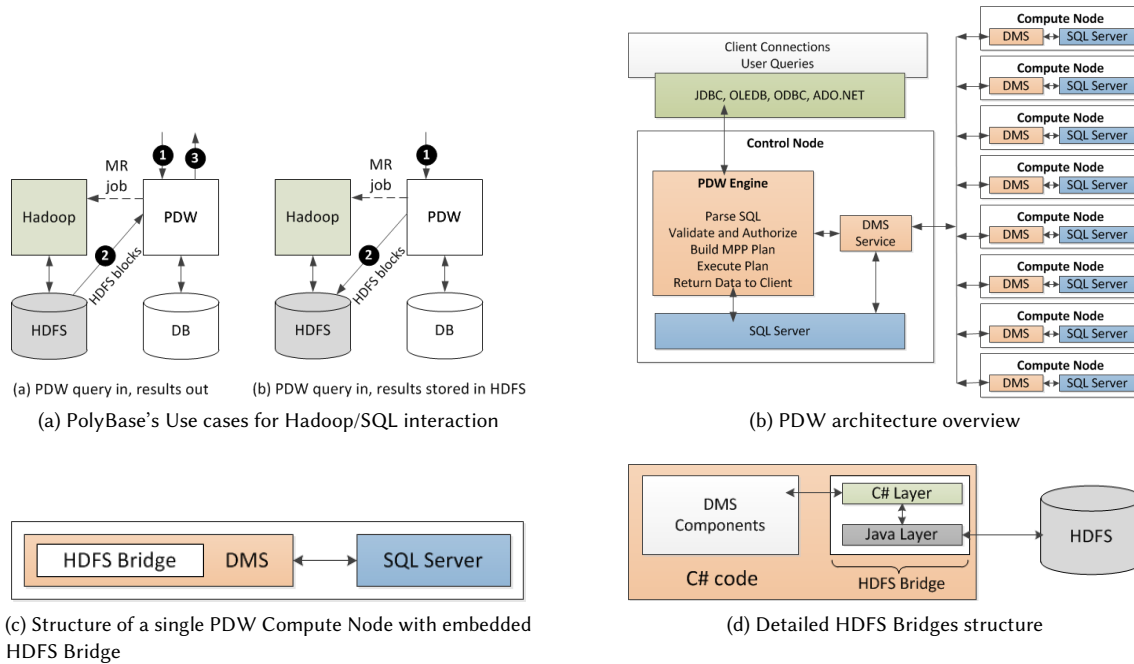


Fig. 8. The architecture overview of PolyBase as part of Microsoft's SQL Server PDW v2 [25]

In many cases, PolyBase can simplify the pushdown of the JOIN operation to improve query performance. When a join is performed on an external source, this reduces the amount of data movement and increases query performance. Without JOIN available, the data to be linked is moved locally from the tables to a temporary data source and only then linked. Joins, projections, aggregations, filters, and statistics determinations are forwarded to the data source via the connection using ODBC. For Hadoop Cluster and MongoDB, on the other hand, filters and statistics calculations are only partially available, while JOINS are not possible with them.

The overall objective is to minimise the data transfer amount between HDFS and PDW. Data imported/exported to/from PDW is processed in parallel and integrated into the same PDW service that shuffles PDW data among compute nodes.

5.7 BigDAWG

*BigDAWG*⁴ is an extensible polystore system with support for multiple data models, real-time streaming analytics, and different visualisation interfaces [27, 34]. The main goals of the system are: (1) the support of location transparency and prevent the user from managing an own mapping of data to store, (2) providing complete semantics of each query concept, and (3) enable the user to convert data from one query interface to another. Figure 9 and Figure 10 shows the technical design of BigDAWG, consisting of four distinct layers: database and storage engines, so-called *islands of information*, middleware and API and the applications themselves.

⁴Big Data Analytics Working Group

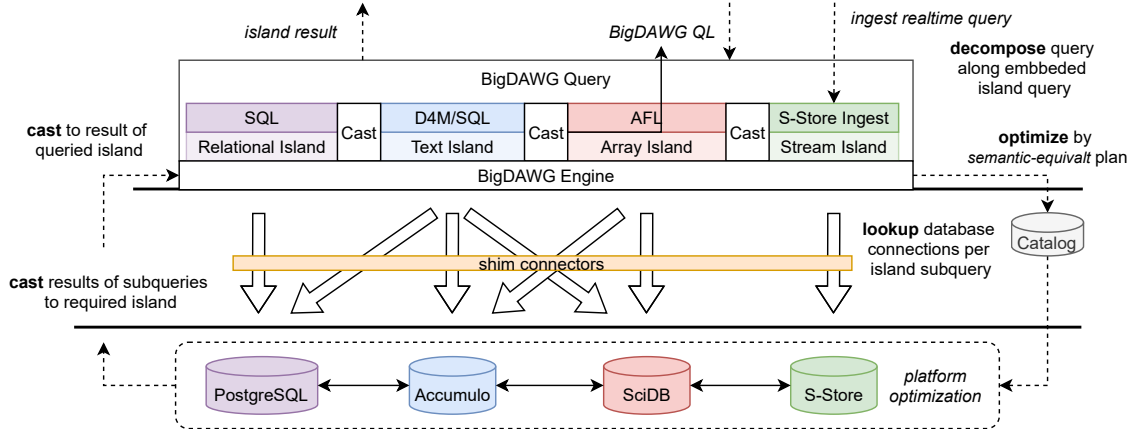


Fig. 9. Conceptual architecture of the BigDAWG system, adopted from [27]

Currently, PostgreSQL, Apache Accumulo, SciDB and S-Store are integrated into *BigDAWG* and can be accessed using different *islands of information*. An island consists of a data model, a query language and a set of storage engines. It enables the user to query a collection of these storage engines with a single query language. So-called *shims* provide the mapping between the data model of the island and the underlying data models in different stores, allowing the extension to other databases by implementing the *shim* contract. For this reason, *BigDAWGs* support multiple islands other polystores such as *Myria* (5.4), *D4M* as processing systems accessing Accumulo, SciDB and employing other semantics build onto of associative arrays. These islands do not procure the complete functionality of the underlying data stores. Therefore, in order to achieve semantic completeness, *BigDAWG* initially comes with islands (*degenerate islands* that provide the complete functionality of one data store. Additionally, it is possible to define new or existing islands by writing a shim for a new data store. Working with *BigDAWG*, the user is also enabled to write queries accessing multiple islands by using so-called *cast* and *scope* operations.

The *BigDAWG Layer* is a middleware responsible for receiving queries, query planning, query optimisation and benchmarking. Query processing works as follows: first, the query planning module parses the query, creates possible query plan trees, and assigns sub-queries to storage engines. The query plan trees are sent to the performance monitoring module, which determines the best tree based on information gained from benchmarking. At last, the query execution module uses the data migration module to identify the best method to combine the results and execute the query.

Query optimisation in *BigDAWG* is based on a black-box approach and continuous monitoring of the performance of each query. The basic idea of optimisation is to move data only in case of expensive operations where a high-performance gain (at least an order of magnitude) can be expected. In order to minimise data movement, all local computations are performed first. Then, the results are combined. Apart from this basic optimisation, *BigDAWG* collects information on the duration of (sub-)queries on various data stores. The resulting preference matrix is learned during three different modes: training mode, optimised mode and opportunistic mode. Every possible query plan is run in training mode on every possible storage engine. In optimised mode, *BigDAWG* chooses the most promising execution plan for the (sub-)query based on the preference matrix. A random plan is selected if the matrix does not contain a similar query. The system also chooses an execution plan based on the preference matrix in opportunistic mode. Additionally, (sub-)queries might be evaluated further during times of low system utilisation or if new storage engines become available.

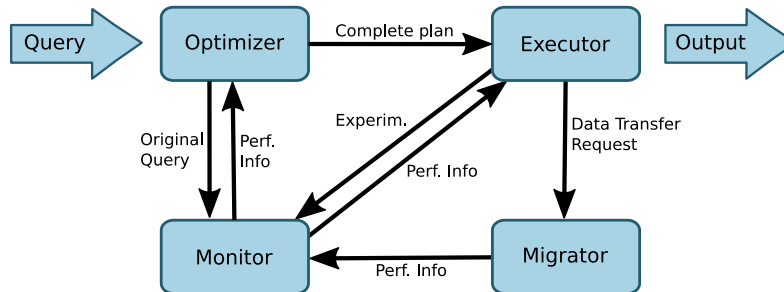


Fig. 10. Technical architecture of the BigDAWG system, adopted from [27]

An additional optimisation aspect, *BigDAWG* considers is the placement of data. Objects can be moved for load-balancing and optimisation purposes but only to data stores with *shims* connected to all islands of source one. The movement of data is triggered by the comparison of the runtimes to similar queries over time, whose access all are going through the *BigDAWG* middleware.

BigDAWG has already been implemented and is available on GitHub⁵. Furthermore, the usefulness of the system has been evaluated for several use cases such as medical application[29] an ocean metagenomic analysis [68]. In the medical application, we already discussed in Section 2.3, *BigDAWG* used PostgreSQL and Myria for clinical data, Apache Accumulo for text data, SciDB for historical waveform time-series data, and S-Store for streaming time-series data [70]. A proposed polystore benchmark made evaluations in [52] showing the flexibility of the island concept and runtime improvements compared to Spark. Improvements reduce latency and benefits for migration tasks by moving data to an applicable store instead of using a reformulation of desired semantics in Spark.

5.8 FORWARD

FORWARD is a federated query processor that hybridises the different data models. The generic query language SQL++ [73] implements a superset of SQL and consolidates different NoSQL query concepts within a semi-structured data model that integrates both JSON and the relational data model.

Two major components compromise the middleware: the integrated view management provides virtual and materialised images. The SQL++ middleware uses the view concept to transfer the virtual view representation from the various stores into the data model of SQL++. Wrapper translates data queries on virtual views into the native query language. Materialised views preserve data states, besides virtual-only views, reduce the translation. Each update of these materialised views is employed by *change* sets, for which FORWARD implements a different view-maintenance approach. The approach, called *idIVM* for ID-based view updates, determines state changes between the underlying schema and the materialised view in FORWARD algebraically [53]. Changed objects are formally defined by their key attributes from the original data sets. Conventional tuple-based comparisons are more compact and calculated more efficiently since not the entire tuple is compared, but solely things according to the key attribute.

⁵<https://bigdawg.mit.edu/get-bigdawg>

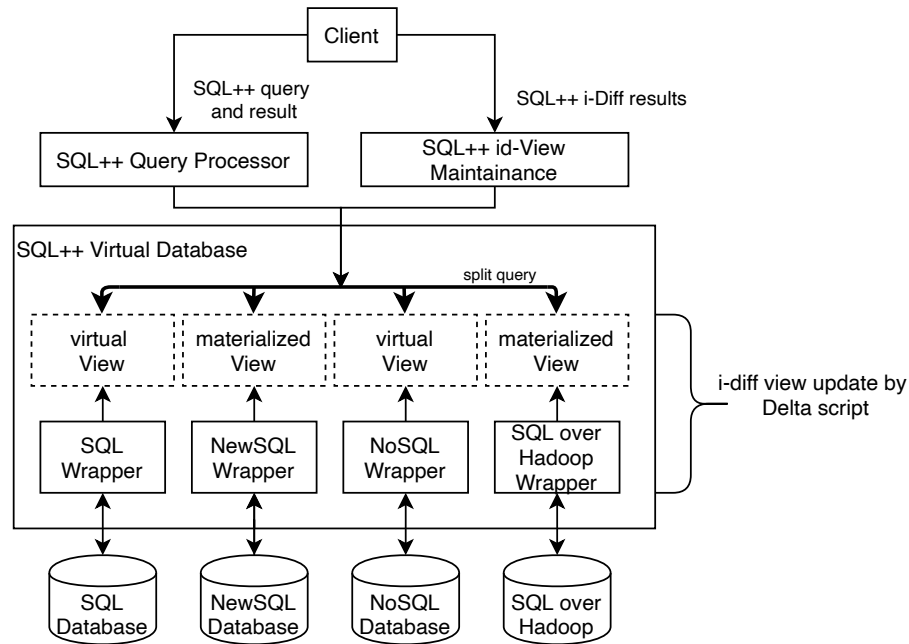


Fig. 11. Placeholder for the FORWARD system according to [73]

idIVM considers four separate steps: within a δ script generation, it first determines the key attributes held in an i-diff schema for the original tables. Incoming changes are transferred to the i-diff values using entries from the modification log. This i-diff propagation determines the actual state change for which a series of DML operations are generated from the standard SQL in the script. Change operations such as DELETE/INSERT/UPDATE affect the materialised views and are optimised in a final step by performing a set of rewriting rules for each query. Semantic minimisation eliminates inefficiencies, composing individual operator rules to gain more performance.

As a hybrid approach to creating a uniform access layer for heterogeneous data, SQL++ considers variable semantics per query. Thus, various semantics can be defined as annotations to consider divergent processing of *null* or equality operations. Throwing of an exception, the processing as zero or using a default value, e.g. with *false* for equality, are conventional solutions.

Internally, SQL++ queries are decomposed into individual sub-queries and use the maintained views. Partial results are integrated within the FORWARD Engine. A description of the translation into the native queries or the kind of decomposition is not provided.

SQL++ is currently implemented in *AsterixDB* as well as *AWESOME* and provides SQL++ runtime system. It provides read requests and a schema definition language defined on SQL++, which allows defining individual types according to the JSON data model, in which a custom type-system implementation makes consolidation around different types.

5.9 MuSQLE

MuSQLE is a distributed SQL query execution, targeting multi-engine environments [40].

MuSQL is a SQL engine to execute relational queries on multiple platforms, able to move data between them. It allows the reformulation of SQL queries and assignment of subqueries on a fixed set of supported engines. In doing so, MuSQL pays attention to reducing the overhead of data movement by keeping operations on data as much as possible together and selecting a join order afterwards according to the locality heuristic. The query engine currently supports PostgreSQL, SparkSQL and MemSQL, all with native support for SQL dialects. It implements partial aspects with schema administration, query planning and execution, strictly based on the relational model. The support by MuSQL is the distribution of query execution and federation according to 4 over this heterogeneous landscape. MuSQL optimises the SQL execution by intercepting the Spark interfaces via a proposed API contract for extending the SQL distribution on other systems.

System optimisation is performed on the logical plans, allowing local physical optimisation by engines and taking intermediate result movements into account. Although each engine operator and cost model do not need to be integrated, only specific API calls must be implemented. The changes made to engines allow the creation of virtual tables and estimate execution time for queries.

MuSQL consists of three central components. The *Metastore* component stores and manages the schema and mapping information for each table. Tables are stored distinctively in one of the supported SQL engines as a whole. The *SQL Parser* reads and validates the query and generates the corresponding object model in the form of a query graph. The multi-engine optimiser uses a cost model defined explicitly for MuSQL, and the *Engine API* acts as a wrapper layer in which the communication with the engine is abstracted, or the query is delegated further.

Separated into two categories for execution and estimation, the API for MuSQL offers the possibility of calling up the actual execution of a given SQL query and time estimation for each connected engine. The query planning uses this engine API, coupled with the concrete SQL engines and expects five functions that must be implemented individually in each wrapper. In the current implementation, MuSQL checks queries in PostgreSQL and MemSQL by utilising their available EXPLAIN statement, while for SparkSQL, multiple individual cost models are implemented for each SQL operator.

In addition to cost estimation for query execution, estimates for migration within the system are also determined. The *emphEngine* API provides a function in which the loading of tables, encapsulated in MuSQL by an engine-spanning Spark DataFrame, can be determined.

MuSQL's optimisation focuses on the JOIN order and uses the *DBSize* dynamic programming planning algorithm already known from DB2 with its extension as *DPhyp* in [72].

DPhyp generates a linear JOIN graph (left-deep-join tree) which is iteratively enlarged. The choice for the next JOIN partner is examined by counting the linked subgraphs. Such a connection exists if, for two induced sub-graphs of the JOIN graph, each with a disjoint set of nodes, at least one edge exists which connects the two sub-graphs. These are called connected sub-graph pairs (*csg-cmp-pairs*). MuSQL extends this JOIN order planning by a location-based optimisation and the inclusion of cost and statistics estimates. In location-based optimisation, the properties of the distributed tables and the transfer of intermediate results are taken into account. This is achieved by determining for the *DPhyp* algorithm a plan for precisely one existing engine and all available ones.

The enhancement achieves the inclusion of costs and statistics using the available information from the Engine API in *DPhyp*. Therefore each possible execution plan is checked with its engine combination and required data movement operation, utilising the table loading cost function provided by the API. Estimated costs are determined for each join using the identified *csg-cmp-pairs* of the extended *DPhyp*. Besides, MuSQL's query planning checks to shift the query

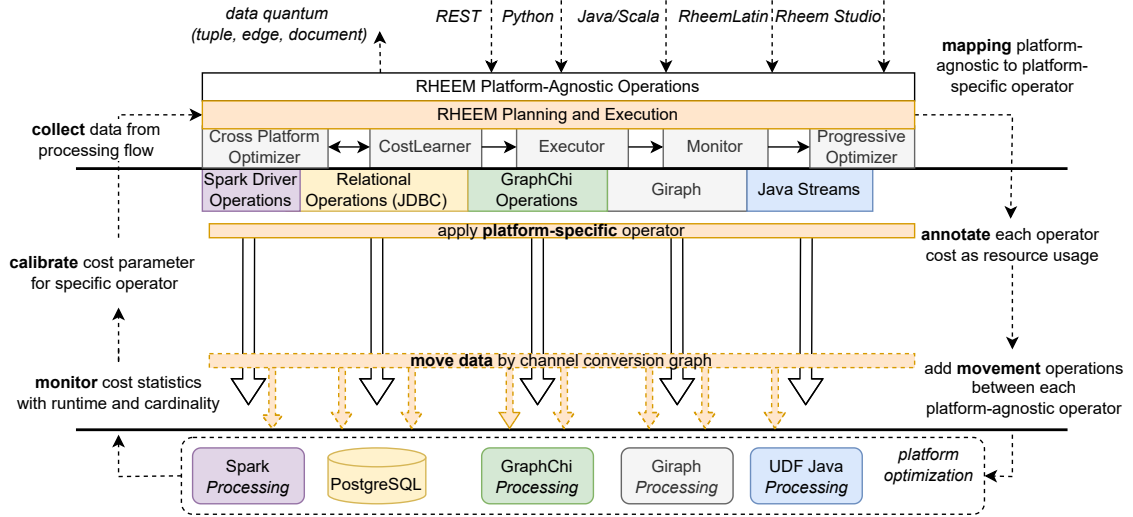


Fig. 12. Technical architecture of the RHEEM processing system, adopted from [3] with data storage and processing systems.

processing to another SQL engine that is cheaper execution. Considering the costs for loading intermediate tables into the engine and preparing temporary tables, MuSQL checks all possible execution plans.

Since MuSQL provides individual cost estimations of relational operations to integrate SparkSQL, new systems must implement the MuSQL API engine contract. This contract requires the cost estimation for relational processes and the interpretation of the SQL language.

Additional effort in query planning is required due to the extra search dimension coming through each engine owned performance for a specific query.

Experimental evaluations with MuSQL successfully transfer whole intermediate table results and benefits in location-oriented planning for the TPC-H benchmark. However, it is prohibitive for using it with large datasets required in big-data analytics.

5.10 RHEEM

RHEEM is general-purpose cross-platform data processing system (DBMS, MR, NoSQL) objecting to decouple the application by providing a multi-platform execution and data storage independence [3]. The goal of *RHEEM* is efficient utilisation of so-called tasks in connection with the migration of dependent workload within a distributed system. Due to the high variance of data models and database systems through their API, *RHEEM* propagates a 2-tier model. On the upper layer, *RHEEM* uses platform-agnostic operators for mediation and migration operations, whose accessing application makes use of them and provides an abstraction to the data. On the lower layer, *RHEEM* uses platform-specific operators that are executed directly on the dynamic data processing systems. For this purpose, the system maps platform-agnostic operators to platform-specific execution operators, including analytical operators such as map, reduce, group-by, and reformulated equality and inequality joins. *RHEEM* uses specific platform implementations and connects them through the internally used Java runtime platform. The mappings can be either direct (1:1) to an underlying platform-specific operator, extending (1:n) to several operators or reducing (n:1) if one operator already

fulfils all functions in combination. The set of possible combinations of input and output operators is the set of all available executions, in which *RHEEM* calls an inflated plan. This inflated plan contains connected operations and the partitioning of required data.

In contrast to other systems such as ESTOCADA [4] or FORWARD [73], *RHEEM* calculates no materialised views and only keeps the data where it has initially been stored, moving data objects temporarily for the scope of a query. In particular, the migration between platforms in *RHEEM* is possible by utilising a so-called *conversion graph*, whose vertices contain the different data formats/structures and directed edges consider the transformation from one model to another [59]. Self-defined or existing operations implement the so-called conversion channels as model transformations steps and are reusable (e.g. service or mapping functions) or not (e.g. streams when they are at the end). The direction of the edges describes clear possibilities to translate from one model to another. The goal of *RHEEM* migration is to create a minimum conversion tree (MCT) from the possible conversions in which the root format leads to one or multiple consumers. The concrete problem is optimising and reducing the conversion cost, considering the reusability through the directed channels. In [59], it was proven that the problem in finding the MCT is NP-hard.

To produce an efficient execution plan from the inflated plan, *RHEEM* estimates the cost for each platform-agnostic operation and cardinality with necessary migration steps. The system annotates the inflated plan with these estimates, extended by the costs of necessary migrations, which arise from the MCT and concretely determines the resource utilisation from a genetic cost model. This cost model considers a platform-agnostic operation with estimated cardinality and calibrates parameters for different acquired resources. By monitoring and collecting execution statistics of a given plan, *RHEEM* differs the estimated cost from the actual one, calculating a geometric loss for an agnostic operation. The loss improves the prediction by calibrating weighting factors for multiple used resource features such as CPU cycles, consumed storage and outer resource utilisation (memory, disk, network). Calibrating these low resource cost factors requires an abstraction of processing to a single kind of data object. For this reason, *RHEEM* introduce an additional abstraction level, given for concrete data objects, so-called *data quantum*. Data quantum can represent multiple formats such as database tuples, graph edges, and document content, helping to trace the resource consumption when processing individual objects.

When extending *RHEEM*, newly supported systems must provide a mapping of the abstracted *RHEEM* operators to their local operations or operation sequences. These types must be integrated into the conversion graph if new systems support new models, respecting the input conversion channels.

RHEEM provides a development environment *RHEEM Studio* [66] for non-experts in order to create *RHEEM* plans, using their processing language *RHEEMLatin* and other interfaces such as Java, Python or simple REST. According to Agrawal et al. [3], the ecosystem supports Postgres, JavaStreams, Apache Flink, Apache Spark, GraphChi and Giraph.

6 RELATED SYSTEMS

In our research on polyglot persistence, we encountered multiple systems which fall into or near the category of polyglot managed systems but did not fit our criteria of a data store very well. These systems focus more on other aspects of data management that we do not consider in the scope of this paper (such as pure OLAP systems, query engines or frameworks). In this section, we want to give them some spotlight and briefly present their idea and purpose but will not go into detail for our analysis and comparison further on (see Section 7).

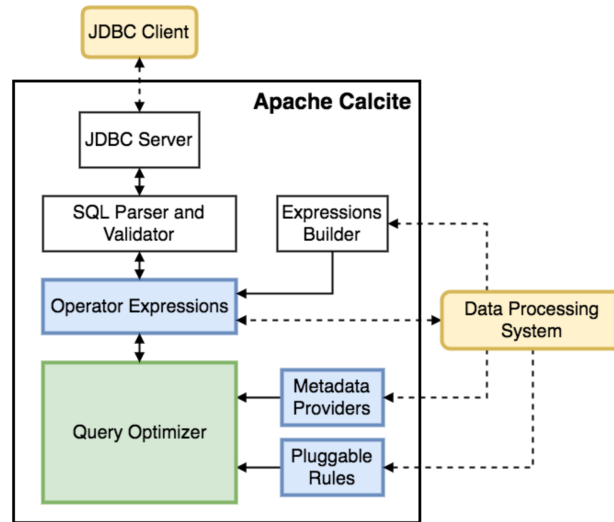


Fig. 13. System integration architecture for query planning and optimization (based on [12])

6.1 Apache Calcite

Apache Calcite [12] is a Java-based software framework that provides query processing, optimization, and support for the SQL query interfaces, granting uniform access to multiple data processing systems.

Calcite's architecture consists of a modular and extensible query optimizer with built-in optimization rules. The query processor handles a variety of query languages, utilized by an adapter architecture designed for extensibility and support for heterogeneous data models. This adapter approach allows the user to connect data stores containing a subset of data objects. Internally, Calcite represents its queries via a relational operator tree, implementing the base SQL language with a subset of extensions. This subset concerns stream-based queries such as sliding windowing expressions and geospatial queries. A dedicated GEOMETRY data type enables geospatial queries. Stream-based queries are implemented internally by a dedicated window operator, and a new map and array type allow to access semi-structured data.

Apache Calcite is available as a framework, and the framework design is required to define its mediation system with configuration concerning connected stores with Local-as-View mapping of source to the global one. Figure 13 shows the integration architecture of Calcite with query processing components and required user interaction to couple external data systems.

The central component of Apache Calcites is its query optimizer, consisting of planning rules, metadata providers and a planning component. The set of planning rules includes transformations of the query tree, in which possible transformations preserve the semantics of the query and resolve via pattern matching on the operator tree. Calcite includes multiple optimization rules for respective connected sources, containing precise query reformulations whose set can be extended manually for each connected system.

The interaction between the framework and a system is possible in Calcite in several ways. When using Calcite for heterogeneous access, an available common compliant JDBC interface allows expressing the SQL statements directly, whereas the support for LINQ4J language extension in Java abstracts completely from SQL queries. LINQ4J encapsulates

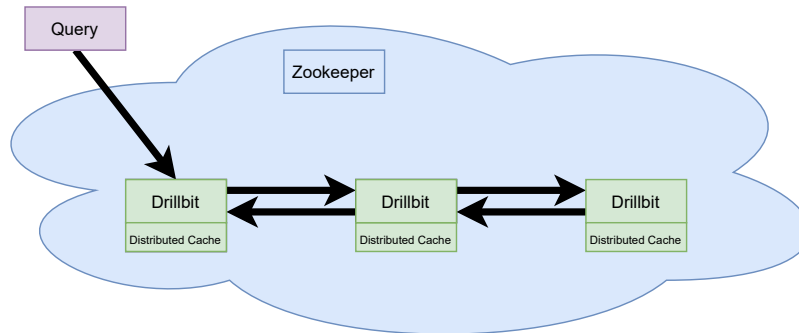


Fig. 14. High level representation of the flow of a Drill query and the system's architecture (based on [8])

the query formulation behind Java functions and delegates, letting the user work within a single programming environment.

In addition, Calcite allows to link Or the framework can be linked over so-called Calcite adapters with new systems, in which a closed converter sub-component is implemented as an interface. This converter is responsible for the transformation into the native interface and the forwarding of the relational expression.

The framework is under active development and is currently being used in multiple public projects such as Apache Beam, Flink, Drill, Solr, Phoenix, Apex, Storm and Samza.

The framework omits some key components, e.g., storage of data, algorithms to process data, and a repository for storing metadata. These omissions are deliberate: it makes Calcite an excellent choice for mediating between applications having one or more data storage locations and using multiple data processing engines. It is also a solid foundation for building bespoke data processing systems. The Calcite architecture is not only tailored towards optimizing SQL queries. In contrast, data processing systems commonly use their parser for their query language.

6.2 Apache Drill

Apache Drill is part of the Apache Software foundation's software project portfolio and aims at providing an open-source schema-free SQL query engine for Hadoop, NoSQL data stores and cloud storage solutions [8]. As stated in their initial proposal for the project, the team behind Apache Drill took inspiration from Google Dremel to provide an open-source solution capable of scaling to 10.000 or more servers and processing petabytes of data⁶. Thus, Drill supports a variety of NoSQL databases and file systems, including HBase, MongoDB, MapR-DB, HDFS, MapR-FS, Amazon S3, Azure Blob Storage, and Google Cloud Storage, Swift, NAS, and local files [8].

The main component of Drill is the Drillbit service. This service is part of Apache Drill's concept of a distributed processing engine using multi-level serving trees [69, p. 58–59]. Every node running a Drillbit service is capable of (1) accepting a query and (2) performing a query rewriting process.(3) Generating an optimized execution plan and (4) run the plan on appropriate nodes⁷, (5) collecting the results of all involved nodes and finally (6) sending the client the queried result. Figure 14 illustrates the flow of a Drill query and the system's architecture on a high level.

⁶<https://cwiki.apache.org/confluence/display/incubator/DrillProposal> (last accessed: 06.12.2020)

⁷A Drillbit service consults a Zookeeper instance to check for suitable and available Drillbit nodes

Apache Drill can join data from multiple datastores in a single query without the client's need to manage metadata. This information is automatically derived from Drill's storage plugins to integrate the different datastore into its system. These plugins also enable Drill's datastore-aware optimizer, which is necessary for the query optimization process of a Drillbit service, in which Apache Drill considers data-locality for their Drillbit service, colocating the service on the same node where the datastore is running.

6.3 AWESOME

AWESOME (Analytical Workbench for Exploration of Social MEDia) [23, 24] is a multistore system focusing on data investigation and analytics. Currently, *AWESOME* supports relational, semi-structured and property graph data as well as a set of computational structures for matrices and vectors. The data stores which are integrated into the *AWESOME* system at present are PostgreSQL, AsterixDB, Neo4j, Apache Solr Cloud and a Spark computation engine. Replacing these data stores is possible if the new data store has the same model. Then, a couple of query translation rules must be redefined according to the new data store.

Together, the different data models and data stores allow *AWESOME* to handle two kinds of data heterogeneity: (1) applications that have to store and query heterogeneous data sets and analyze them by combining different data models in arbitrarily complex ways and (2) applications where data sets have a mixed model.

Data placement and computational processes of raw and derived data are defined by the data ingestion language ADIL. This *AWESOME* specific language allows specifying complex ingestion workflows, ingestion filters and complex query statements. For instance, it enables the user to define store specific data acquisition processes and route data into appropriate storage. Furthermore, the construction of *analysis environments* and data augmentation processes is possible. The developers of the *AWESOME* system differentiate between *dictionary-based* and *computational* data augmentation. In the case of dictionary-based data augmentation, a polymorphic join operation is performed periodically between terms in a dictionary and any data set residing in one of the underlying data stores. On the other hand, computational data augmentation is achieved by passing a data item through a function that returns the augmented form of the item as a result.

The architecture of the *AWESOME* system consists of nine different components. Unfortunately, the authors do not provide an architectural overview of the system's components or further information on their query processing workflow. Nevertheless, some components are mentioned and will be described below.

One of the main components is the *Data Ingestion Module*. It receives data from various sources (e.g. file sources, database sources, data APIs and streaming APIs). The connection between the system and the different data sources is realized by adapters that can gather and filter data and perform manipulations. Furthermore, the adapters are responsible for populating data based on a decision table into the appropriate store. The *Data Derivation View Manager (DDVM)* uses database views and model transformations to manage the data derivation process. It can be considered as a repository for derived views that compiles view definitions into workflows of tasks. Besides, the *DDVM* ensures that the materialization processes of every view continue to be executed if updates (insert-only) occur in the primary repository. The *Query Processing Module (QPM)* parses, plans, optimizes and executes queries. These queries may either be pass-through queries that are expressed in the native query language of an integrated data store or *AWESOME* queries that are processed through a data federation layer using a hybrid data model. The *Data Computation Module (DCM)* is responsible for the computational processes and manages the lifespan and availability of results derived from complex ingestion workflows. In case of user-defined functions that cannot be computed inside any store, the *DCM* manages necessary data exchanges and returns or stores results.

Apart from these main components, the *AWESOME* system consists of an internal data store that is used for objects that cannot be stored in the integrated data stores, an additional cache for data that can only be stored and retrieved but not queried and a *System Catalog (SC)* that keeps track of all data sources and derived views. The integrated *Performance Monitoring Module (PMM)* monitors the system resources such as memory, number of threads and IO operations so that the ingestion and query processing modules are able to incorporate this information. Furthermore, third-party analytical libraries provide access to text processing, graph, statistical, and machine learning methods.

Currently, complex ingestion processes and queries have to consider the native data models of the integrated stores, the different query and ingestion capabilities, and the different computational costs for data insertion, fragmentation, filtering, index creation, data movement, and query processing and computations. Here, the authors see room for improvement so that, in their future work, they plan to enhance the inflow optimization, the content optimization and the dataflow optimization. Inflow optimization decides where to store the data and how often they need to be replicated. In contrast, content optimization covers the elimination of content-free, insignificant or redundant data, cross-store information deduplication, integrity constraints and clustering. The authors plan to use a data flow graph to optimize the data flow and adjust their computed plans to the overall system's load (burst and quiet periods).

6.4 DBMS+ (Cyclops)

DBMS+ (Data Management System for Multiple Systems)[63] is a conceptual approach proposed in conjunction with the existing Cyclops platform to enable distributed or centralized query processing on polyglot system landscapes. The approach provides the user of a domain to specify its application, including availability, consistency, changeability, and cost requirements for the DBMS+. In request processing, these restrictions are taken into account, generated, and evaluated as so-called multi-system execution plans by the DBMS+. In terms of content, the thesis illustrates its approach by employing continuous data stream queries on several analytical example scenarios related to interactions between users and web pages. All scenarios are covered in the concept by continuous data stream queries. In its concept of continuous stream-based execution, DBMS+ first distinguishes between centralized processing, where exactly one processing system is used, and distributed processing, where several different systems are used. In terms of execution types, the plans distinguish between *incremental* and *non-incremental* execution plans. Non-incremental execution plans extract data objects from a request window on the stream and process them directly via a group-by aggregation. On the other hand, incremental processing caches aggregations that have already been executed and pass them on to the next operation only when the result is complete. Furthermore, DBMS+ distinguishes between *imperial* and *federated* processing in a distributed execution. *federated* processing uses the native query language of each connected system, leaving independent parsing and local optimization to the individual databases. This independence makes it easier to connect the systems to the higher-level mediation and exploit local, potentially costly optimizations for each system. However, semantic differences due to different languages must be reconciled. On the other hand, the *imperial* processing called in DBMS+ avoids considering a native query language and skips parsing, generating local plans, and optimizing. It interacts directly with the execution and storage component of the respective system (for example, in the case of the public-access storage component in MySQL), giving it complete control over all storage operations. This direct access avoids constraints on the respective systems and semantic inequalities. Open questions consider implementing and integrating the proposed DBMS+ approach into the existing Cyclops system. Especially which plans should be identified and selected for these connected systems and how the above requirements must be structured and described to get an interpretable format. Furthermore, problems from distributed data management have been discussed that concern the provision of storage space and how related data can be stored in a distributed manner.

6.5 Odyssey

Odyssey [47] system is a multistore for handling multiple analytical processing systems for big-data queries in Hadoop and parallel OLAP. It enables the storage and retrieval of data within HDFS and RDBMS using opportunistic materialized views based on MISO [60]. MISO is a method of tuning the physical design of a multistore system (Hive/HDFS and RDBMS) by deciding to place data in multiple data stores, thereby minimizing necessary data shifts between intermediate result stores. The results of query execution are treated as opportunistic, materialized views that can then be located in the underlying stores to optimize the evaluation of subsequent queries. Thereby MISO builds a sort of "caching" for partial results of outbound analytical calculations. MISO aims to maximize the performance of ad-hoc processing of large data queries by deciding where data is best stored. Therefore, it must maintain and calibrate its cost functions for estimating the cost of operations. As a result, there is an integration effort to generate the corresponding cost estimates for each new engine added.

This optimization problem is equivalent to the Knapsack problem, which is intensified by two dimensions. Each Knapsack has two-dimensional capacities with current storage capacity and transfer quotas. In addition to considering the current workload, future transfer costs must also be taken into account through subsequent queries, which are optionally refined by user-defined constraints. For example, some data can be persistent only to a particular format without changing the semantics. Specifically, MISO refers to this problem as a Multistore Design Problem (MDP), where an observed stream of queries on the current multistore design is used to determine a new one that meets the specific constraints and reduces workload costs. In practice, a so-called MISO Tuner component performs a periodically recurring (online) optimization of the current workload distribution.

6.6 QUEPA

QUEPA is a tool for data exploration and augmentation within polystore setting [67]. It shows a possible contribution to this problem by introducing augmented search and augmented exploration, two new methods to access a polystore based on the automatic enrichment of data extracted from a database with data that belong to the rest of the polystore. Data augmentation search considers the stepwise expansion of search results over a local database with relevant data for this query stored elsewhere in the polystore. The exploration exploits the same infrastructure and provides an interactive way to access data, similar to searching the Web.

QUEPA proposes the following three components: A repository, implemented as a Neo4j database, stores relationships among related data objects in the polystore. The repository is an undirected graph in which the vertices represent the same queried conceptual model. References among these objects are represented as edges in the graph equivalently. Each database serves one or many collections (e.g. tables in a relational database) containing the undirected references between those objects, restricting the data types only to those systems whose items are identifiable by a name and a key.

The collector must identify the relationships between data objects stored in the repository. Due to inherent heterogeneity given by the polystore with their partially consisting schema-less databases, QUEPA combines various unspecified entity resolution approaches from literature in an offline manner. For schema-bound data and information, the system follows proven techniques. Therefore QUEPA considers an alignment- and a record-linkage phase (for resolving entities and removing duplicates) on an existing schema.

The third component is responsible for request processing based on data augmentation, in which incoming requests are validated and rewritten. QUEPA first checks the type of query in which aggregate-oriented queries are executed entirely without augmentation. The query processing adds new data objects based on the identified links with the

same reference level from the repository. This query method makes it able to traverse the data and retrieve them in an exploratory way.

This feature distinguishes QUEPA from the middleware approach in that no additional abstraction layer is created using a mediator. It uses a gentle method for data integration and stores data objects in native format.

QUEPA currently exists as a prototype implementation with multiple stores covering the most used data models by MongoDB, MySQL, Neo4J and Redis database. The paper demonstrates the QUEPA system functionally with four different scenarios: (1) simplicity of configuration and startup of QUEPA, (2) query result comparison with and without augmentation, (3) query result comparison with and without augmentation, (4) a simple exploratory query and (5) adding new system types and promotion of new relationships. The evaluation does not cover non-functional aspects such as consumed resources and query performance.

6.7 QoX

QoX [81] is a particular type of workflow-driven analytical engine that integrates data from relational databases, Extract-Transform-Load (ETL) tools and varying execution engines (such as MapReduce). QoX specifies data flow in an ETL-like workflow, allowing it to combine structured and unstructured data. It provides generic data flow operations such as join, aggregation and filtering, user-defined functions, and pre-defined analysis procedures (e.g. sentiment analysis or product identification). The system's primary focus is to optimize these integrating workflows in performance, freshness, latency, cost, and scalability. To enable the user to define optimization goals for their workflows, QoX provides the feature to annotate the corresponding queries.

6.8 TATOOINE

TATOOINE is a data integration tool that has been developed in collaboration with french journalists [13]. The resulting tool can combine several different sources and enables the user to analyse web and social network communication.

TATOOINE uses a GAV data integration approach to store static as well as dynamic information from structured, semi-structured and unstructured databases or social feeds. Apache Solr, a set of relational databases and RDF data sources, is currently integrated. Since TATOOINE does not use mappings or an integrated global schema, queries are evaluated over so-called *mixed data instances*. *Mixed data instances* consist of a set of data sources and an application-dependent RDF graph that contains domain-specific knowledge about the data within each store and the connections between data residing in different stores. Information and structures from the RDF graph can be combined with any data source using the join opportunities provided by repeated values such as names or hashtags. The resulting *mixed queries* combine sub-queries expressed in the data store's native language with RDF querying. RDF query languages such as SPARQL provide, amongst others, operations like disjunctions, pattern matching, aggregations and the construction of triple results, as well as the derivation of implicit triples of the RDF graph.

Even though *mixed queries* are a powerful and flexible tool, a vast amount of knowledge and expertise is necessary to use them. Therefore, TATOOINE also provides keyword-based querying. For each source, digests are derived, containing the schema and a representation of a set of atomic values associated with positions in the schema. The digests can be viewed as directed graphs. Given two or more keywords, the engine looks them up in the digests and identifies the shortest paths between them afterwards. The shortest paths contain additional information that is closely related to the keywords.

TATOOINE is strongly tailored to the specific usage patterns of journalistic research. In favour of more general approaches, it has not been analysed further.

7 COMPARISON

Based on their different design objectives, the systems presented in Section 5 offer different properties, advantages and limitations. In this section, we categorize them according to the system types we presented in Section 4 if applicable and propose a comparison based on the concluded requirements we identified in Section 3. The results of our categorization and comparison efforts are displayed in Table 2, Table 3 and Table 4, featuring information about system types, the systems' architectures, functional features and query processing capabilities as well as their adaptivity properties.

7.1 Architecture

The comparison in Table 2 shows significant differences in system design for all considered systems, themselves stated as polyglot or multistore database or data management. Despite the wide acceptance of applying polyglot design in supporting use-case by a mix of database systems, there is still no common sense in designing such system kind overall. Only the classification by [75] in separated designs of linked-data, mediator-wrapper- or simply data warehouse-systems were being approved. Systems suited more public access result in linking multiple heterogeneous system by given a Local-as-View mappings and common ontology with unified access via a set of *classes* and *relationships*. Data access is applied by reasoning on the ontology, expressed with Description Logic what is called *Ontology-Based Data Access*. Building polyglot systems from scratch results in concerning a distinct mediator using one or multiple wrappers in order to query data from underlying data stores. In contrast to the linked-data approach, in which (web-)data bases on ontology's, stored in RDF format, queries are formulated against a single query interface or in this native language where data is located. A formulated query is either being executed directly without concerning any capabilities or planned and optimized, minimizing essential migrations steps. Given the single intermediate model, results are represented in this way how store communication is managed (e.g. exchanging complete relations or single tuple data objects). Besides abstract design concepts, Table 2 shows that the common sense in distinctive concerning these described system types in Section 4 is mostly approved. The majority of systems can be categorized as multistore. Besides the simplicity of managing a single query interface, this approach allows to include capabilities of the connected databases in one's own query planning. In contrast, prominent references of BigDAWG [27] and Polypheny-DB [92] do not aim at merging the respective capabilities of the databases into a higher-level model, but rather at advancing the respective available language. While BigDAWG does not include any data location in the planning and this must be managed and formulated by the user, Polypheny-DB has the overall goal of being able to apply all available query languages to the entire set of all managed data objects. A special focus is represented by the commercial solution from Microsoft in the form of PolyBase [25], linking, external data sources such as AS HDFS cluster provider and CosmosDB with its own SQL server T-SLQ language and operation. Despite its simplified use case as a wrapper for reformulating SQL queries and evaluating them on mappings into external systems, this system is classified as polyglot.

The first evaluated dimension captures functional features. Polypheny-DB and AWESOME are the only data stores which support write operations, whereas all the other stores only provide read functionality and have their focus only in an optimized and integrated multi-engine analytics, which does not operate on a global schema. Only the AWESOME system is using their *ADIL* language as a form of explicitly defining a schema. Furthermore, the lack of a DDL in most of the systems results in the non-existence of data constraints, which corresponds to the read-only character. Advanced features such as data privacy, security, as well as push-based queries, are provided by none of the systems.

FORWARD and Polypheny-DB support the most extensive amounts of varying data models, in which the latter is planning to enable the user to specify data requirements (e.g. a minimum level of consistency) in forms of annotations.

Table 2. Categorization and comparison of the systems from section 5
 (✓ - existent, ✗ - nonexistent, ⊕ - partially fulfilled, ⊖ - unknown)

	CloudMdsQL	BigIntegrator	ESTOCADA	Myria	PolyPhenyDB	Polybase (HDPS-Bridge)	BigDAWG	FORWARD	MusOLE	RHEEM
System Type	Multistore	Multistore	Multistore	Multistore	Polystore	Multistore (HDPS-Bridge)	Polystore	Multistore	Multistore	Multistore
System Availability	research system	research system	research system	research system	vision	commercial	research system	research system	research system	research system
System Evaluation	[56]	✗	[4]	[49, 93]	✗	[25]	[52, 97]	✗	[40]	[3]
Architecture	Component connection Mapping Extensibility CAP Orientation	loosely-coupled LAV ✓(++) CA	tightly-coupled LAV ✓(++) CA	loosely-coupled LAV ✓(+) CA	⊖ ⊕ ⊕ CP	loosely-coupled GAV ⊕ AP	hybrid-coupled Hybrid ✓(+) CA	loosely-coupled GAV ✗ CA	tightly-coupled GAV ✓(+) CA	tightly-coupled LAV ✓(+) CA
Currently supported data stores	PostgreSQL (DBC), MongoDB, Apache Spark, Derby, Python ^a	BigTable, standard-SQL supporting DBMS	relational systems, Spark, AsterixDB, Redis, Solr, Saxon	SciDB, HDPS, MyriaX, SFARQL supporting stores	MariaDB, PostgreSQL, VoltDB, MonetDB ^b	Microsoft SQL Server, Oracle, Teradata, MongoDB, Generic ODBC, Hadoop, Azure Blob Store	PostgreSQL, SciDB, Accumulo, S-Store, Myria	AsterixDB, MongoDB, CouchDB	PostgreSQL, MemSQL, SparkSQL	JavaStreams, PostgreSQL, Flink, Spark, GraphChi, Graph
Transparency	Location Transformation / Migration Concurrency	✓ ✓ ✓	✓ ✓ ✓	✓ ✓ ✓	✓ ✓ ✓	✓ ✓ ✓	✓ ⊕ ^c ✓	✓ ✓ ✓	✓ ✓ ✓	✓ ✓ ✓

^aNo data store

^bSet of DBMS currently supported by Icarus implementation

^cOnly for single-island queries.

Table 3. Categorization and comparison of the systems from section 5
 (✓ - existent, ✗ - nonexistent, ◐ - partially fulfilled, ? - unknown)

	CloudMdsQL	BigIntegrator	ESTOCADA	Myria	PolyPhenyDB	Polybase	BigDAWG	FORWARD	MusQL	RHEEM
Query Language	CloudMdsQL (using native query languages), Python	SQL, GQL	No common query language; pivot language; relational algebra	Datalog, MyriaL, Python	SQL dialect (e.g. PolySQL [91]), CRUD, openCypher, vitivr	T-SQL	SQL, AFEL, MyriaL, SPARQL (island-dependent)	SQL++	SQL	RheemLatin, TheemStudio, Python, Scala, Java, REST
Definition Language	✗	✗	✗	✗	SQL dialect ⁴ , CRUD interface, openCypher, vitivr	T-SQL	✗	SQL++ with JSON related Types and Datasets (in AsterixDB)	✗	✗
Manipulation Language	Data changes defined as native queries	✗	✗	✗	possible in all available query languages; updates can be made in-place or versioned	T-SQL	✗	SQL++ with INSERT/DELETE/UPDATES of datasets (in AsterixDB)	✗	✗
Functional Completeness	Provided by native query languages	only by providing extending query execution plugins	Native queries allow native feature access	Allows user-defined functions and aggregates constrained imperative	Provided by native query language	Subset of T-SQL operations, provided by DB bridge	Provided by degenerate islands	Allows the definition of own functions; no direct access of native store functionality	✗	Fixed but extendable by user-defined functions
Input Models	Depends on native query language used	Relational Model	Nested relational model, key-value, graphs, document	Relational Model	Depends on native query language used in subquery block	Relational Model	Island-dependent	Hybrid Model (Relational + JSON)	Relational Model	Data quanta (abstraction of tuples, graph edges, document content)
Adaptivity	✗	✓	✓	✓	✓	✓	Only within islands	✓	✓	✓
Logical Data Independence	static	static	static	static	horizontally and vertically	static	static	static	vertically and horizontally fragmented	static
Cross-Model Persistence	✗	✗	✗	✗	Estimation of current and future workloads to adapt by adding/removing databases and migrate/replicate data	✗	✗	✗	✗	Cost model learner adapts cost function for query processing
Adaption Approach	✗	✗	✗	✗	GDPR compliant data distribution, data safety aware replication	✗	In conjunction with S-Store	✗	✗	✗
Data Privacy and Security	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Push-based Query Execution	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Custom Optimization Targets	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

⁴Tearus uses self-developed dialect PolySQL [91]

Table 4. Categorization and comparison of the systems from section 5
 (✓ - existent, ✗ - nonexistent, ⊕ - partially fulfilled, ⊗ - unknown)

	CloudMdsQL	BigIntegrator	ESTOCADA	Myria	PolyPhenyDB	PolyBase	BigDAWG	FORWARD	MuSQL	RHEEM
Semantic Completeness	Semantic of native stores transformed into provided CloudMdsQL typesystem	fixed	Distinct semantics treated differently. Encode the semantics same as much as can	Relational semantics extend by iterative processing constructs such as do-while and while loop	⊕	PDW filters for unsupported types and handles them itself	Island-dependent (defined during development of island)	varying semantics bypassed by annotations per query	Standard SQL support extended by store-specific functions	fixed
Cost Estimation	cost-based	⊗	dynamic		heuristic	cost-based	heuristic, benchmarking	⊗	cost-based	adaptive cost-based
Optimization Approach	heterogeneous join (bind join), operator reordering	heterogeneous join (bind join), pushing-down operations	query-rewriting minimization (respecting conditions), sub-query minimization	data-movement minimization, operator reordering, heterogeneous join (tributary-join)	⊗	(classical) query planning, pushing down operations	minimization of data movement, pushing-down operations, benchmarking for semi-equivalent query	caching-like approach based on materialized views	join operator reordering (finding subplans for all possible locations), data-movement minimization	data-transformation minimization, data-movement minimization, operator-reordering, adaptive cost model
Query Decomposition	data location dependency, according to native query fragment in global query, according to parallel processing	data location dependency, data store capability	Native queries translated into relational pivot language	data location dependency, according to parallel processing (splitting by shared-worker thread)	data location dependency, data store capability	data location dependency	according to capability, according to data island dependency	⊗	data location dependency	data location dependency
Internal Query Model	Relational	Relational	Relational (views with constraints)	Relational	Associative Arrays	Relational	Island-dependent	Hybrid model consisting of JSON and relational	Relational	Data quantils (abstraction of tuples, graph edges, document content)
Migration	✓	✗	✓ ^a	✓	✓ (not specified further)	✓	✓	✗	✓	✓
						On-Demand		✗	On-Demand	On-Demand

^aFragments stored as materialized views.

In terms of query processing, the adopted approaches vary strongly. One part of almost all query execution systems (except for *CloudMdsQL* and *ESTOCADA*) is query splitting. The optimization of queries is in the majority of cases cost-based, but heuristics and benchmarking are also applied. A common data model is used by all stores except for *BigDAWG*. Most of the systems use the relational model but, especially when *NewSQL*-stores are enclosed, associative arrays and documents are also common. An aspect which seems to be slightly neglected is the semantic of the systems or rather the semantic mapping between different data stores. For almost all cases, the semantic mapping is fixed or not even considered at all (also referred to as fixed. *FORWARD* offers with its *SQL++* query language the most complex approach where the semantic mapping between the underlying data stores can be defined for each query by the extended relational algebra. Another possibility to achieve semantic completeness is provided by systems such as *CloudMdsQL* and *BigDAWG* that allow using multiple query languages and their semantics in combination.

Overall, the characteristics of the different system architectures are similar. Almost all systems prefer a *CA*-approach. *PolyBase* is, caused by the fail-safe structure of the underlying *HDFS*, the only system that favours availability and partition tolerance over consistency. Solely, *Polypheny-DB* is the only system that enables the user to choose between consistency and partition tolerance. As a form of data integration, most systems follow a *LAV* style or at least a hybrid variant. Only *FORWARD* and *PolyBase* are using the *GAV* approach. In *FORWARD* the extensibility of new data stores is handled by dedicated interfaces and already approved for *AsterixDB*. Only *Polypheny-DB* and *PolyBase* do not take extensibility into account. All other systems propose individual solutions, but are as a result more complex.

The adaptivity dimension is considered by none of the existing systems. Only aspects such as providing concurrency and location transparency are shared among all systems. The former is caused by the systems' read-only nature. Logical independence is provided by each system except *CloudMdsQL*. *Polypheny-DB*, *QoX*, *PolyBase* and *BigDAWG* are able to migrate data during query execution at least temporarily. Only *Polypheny-DB* is able to change the used data stores and distribution model (replication/partitioning), intends to orchestrate hardware and (geo-wide) data placements if changes are required.

In summary, we observe that a large amount of requirements is not suitably realized by the presented set of polyglot data stores. The system which seems to achieve the most of them is currently *Polypheny-DB*, but no coherent prototype or implementation is available yet. According to the authors, some of the necessary components have already been realized as standalone applications or as parts of other systems.

7.2 Transparency

Given by the underlying design principles of *Poly-* and *Multistores* they are complex software structures composed of multiple heterogeneous data stores in order to utilize the different advantages of the underlying stores. As we established in Section 3, it is beneficial to hide the adaption and data distribution processes so that the user can focus on working with the data itself. Therefore, the focus of this subsection is on Transparency⁸, especially w.r.t. location transparency, transformation/migration transparency and concurrency transparency. These three key areas were chosen because of their importance for the seamless distribution and processing of data over multiple integrated data stores.

Based on the evaluation of the different *Poly-* and *Multistore* systems it became very clear that all systems hide the inner workings of their polyglot systems. No system communicates any information of the data that is linked to the location or migration of the data. For any user it is not directly distinguishable whether the processing is done on a

⁸Clarification: A process is considered as transparent if the inner working of the process are not inherently visible from the outside.

single store system or a Poly-/Multistore. Furthermore, the systems don't show how the data is processed, how the data retrievals or migrations are handled or how they may be done concurrently.

The only notable exemption is BigDAWG. This system only provides some level of transformation/migration transparency if the query is focused on a single island. In any case where multiple islands are part of a BigDAWG query a casting mechanism has to be formulated.

7.3 Query Interface

One of the most defining features of Poly- and Multistores are their query interfaces. They not only specify the type of the system but also characterize the properties that this system is able to offer to its users.

The first contact the user has with the system is usually with the offered query languages. It decisively determines which data models will be used externally and how powerful the descriptive tools are the users may utilize for working with the data. An analysis of the implemented query languages of all evaluated systems shows two main trends: The usage of a SQL derivative/dialect and the implementation of an original query language. Systems which use their own language are mainly CloudMdsQL with its query language of the same name, Myria with MyriaL and RHEEM with RheemLatin. Furthermore, MyriaL finds usage in the Polystore BigDAWG, too, as the query language of one of BigDAWG's islands. Nonetheless, many of these system offer further ways for the user to interact with the database. For example, all three integrate Python interfaces. In addition to this, Myria implements Datalog. Likewise, RHEEM extends its offering by a multitude of systems and APIs, like RheemStudio, Scala, Java and REST.

The datastores incorporating SQL-based languages cover a wide variety of dialects. The Google product BigIntegrator uses both SQL as well as the in-house developed Google Query Language, which has a SQL-like syntax for operating on their Google Cloud Plattform. Microsofts PolyBase employs Transact-SQL (T-SQL), developed by SAP and Microsoft. Polypheny-DB plans for at least one SQL dialect for its query interface and proposes PolySQL, which was developed as part of the ICARUS multistore by the same research group [91]. In addition to SQL, Polypheny-DB allows interaction via simple CRUD operations, an openCypher implementation and vitivr, a multimedia retrieval engine[76]. A quite unique approach is the query language used in FORWARD, SQL++, which is an extension of SQL to include NoSQL data models and JSON. Additionally, the database system AsterixDB uses SQL++, too, to enable users to work with its own data model *ADM* (Asterix Data Model), which is based on a superset of JSON [5].

ESTOCADA is a notable exception as it does not offer a common query language at all. Instead, it offers a pivot language based on relational algebra which encapsulates the query languages of its underlying data stores. ESTOCADA offers a query languages in a similar fashion by incorporating the native query languages. Additionally, its language portfolio is extended by the Python scripting language.

An observation that is very apparent after a while comparing and evaluating poly- and multistores is the lack or omission of a Definition or Manipulation Language in the sources. In the case of a *Definition Language*, only Polypheny-DB, PolyBase and FORWARD are explicitly offering any kind of possibility to define schemata. All three stores use their query interface languages for this (e.g. PolyBase → T-SQL, FORWARD → SQL++).

For *Manipulation Languages*, these three stores are joined by CloudMdsQL. The data manipulation is done by using the native languages of the underlying stores that are offered by the poly- and multistore through their query interfaces or - in the case of FORWARD - by SQL++. Additionally, Polypheny-DB's design should allow to make the updated in-place or versioned.

One important factor when dealing with a mediation layer like a poly-/multistore is the capability of the store to preserve the Functional Completeness of Incorporated data stores through their query interfaces. Three of the ten systems we evaluated provided the functional completeness by the native query languages of underlying stores, namely CloudMdsQL, ESTOCADA and Polypheny-DB. BigDAWG uses a similar tactic by offering so-called degenerate islands, which are only which encapsulate only a single store and its query language. BigIntegrator, Myria, FORWARD and RHEEM solve this issue by allowing user-defined function, thus offering the possibility to manually extend the multi-/polystore by the desired functional capabilities. It should be noted, that FORWARD's query language SQL++ does not enable *direct* access of the native store functionality. In PolyBase a subset of T-SQL operations (e.g. provided by the Hadoop DB Bridge) offers a portion of the functional completeness of the stores connected to the SQL server. MusQLE lacks any mechanism to preserve its stores' functionalities.

The *Input Models* the stores expect are very dependant their implemented query languages. Accordingly, it not very surprising that the prominence of relational languages also translate to a high number of stores (BigIntegrator, Myria, PolyBase, MusQLE) that work with a relational model. In addition, FORWARD uses a relational model which is extended to provide JSON-support. CloudMsdQL and Polypheny-DB are flexible on their input model as it depends on the used query language. Similarly, the expected input model in BigDAWG is dependent on the used Island. ESTOCADA used a fixed defined list of four data models, which include a nested relational model, the key-value model, graphs and documents. Finally, RHEEM uses a unique approach in the form of data quantas, which is an abstraction of tuples, graph edges and document contents.

7.4 Adaptivity

In order to enable the polyglot system to react to changing requirements impose by the user directly or by changing workloads it has to adapt itself. In Section 3.1 on page 5 different levels of adaptivity have already been presented. Each level corresponds to a different kind of system adaption with a severity regarding costs and extent of necessary system changes. In this section we want to focus on three topics that we consider important for adaption in a Poly- or Multistore: Logical data independence, cross-model persistence and used adaption approaches.

If we look at DBMS design, one key feature of most database systems is data independence. This feature ensures that changes in lower layers won't affect the higher ones. If we deal with adaptivity the question of how adaptive a system can be quickly correlates with the questions on how data independent the Poly-/Multistore layer is to the underlying data stores. Most of our evaluated systems imposed logical data independence between the polyglot system layer and the incorporated data stores but there were two notable exceptions: *CloudMdsQL* and *BigDAWG*. While *BigDAWG* technically ensures data independence, it only does this on its island level (so within an island). As soon as the island context is left, no data independence is warranted. *CloudMdsQL* doesn't implement any form of logical data independence what so ever between its mediation layer and its data stores.

In order to store data in these Poly- and Multistores in a polyglot way, these system employ generally a static cross-model persistence approach. This means that the location of data and the sued data model is mostly predetermined/defined and then appropriately distributed on run-time. This distribution can be given in the query statement itself like in *CloudMdsQL* where the used native query language implicitly determines the used store. Alternatively it can be given by a explicit schema mapping, a method i.e. Polybase employs. However, these static methods do not distribute whole data entities by e.g. fragment the data horizontally or vertically. Nevertheless, the horizontal

and vertical distribution of data is by no means something that does not find application. In our evaluation both Polypheny-DB and MuSQLE are system which incorporate these strategies in their cross-model persistence approach.

During our evaluation one aspect of currently available Poly- or Multistore systems became quite apparent: Nearly no system incorporates any real form of adaptivity. Given the sheer amount of challenges the development a Poly-/Multistore has on its own, the addition of a (sophisticated) adaption system on top may not be the focus for the involved developers. Nevertheless, research projects such as *Polypheny-DB* show that systems potentially will develop in this direction in the future. This system is the only system which is incorporating adaption into its system design. Triggered by estimations of current and future workloads *Polypheny-DB* should adapt by adding or removing data stores to its running system as well as migrate and replicate data to adjust data locality and distribution. At this point, however, it must unfortunately still be pointed out that *Polypheny-DB* has so far only described this function in a vision paper but has not implemented it. A system which implements a practical and functional form of adaptivity in our evaluation is *RHEEM*. It is the only other system which describes adaptivity as part of its systems functionality. *RHEEMs* adaptivity shows itself in the form of a cost model learner, which adapts the cost function for the query processing by tracing the resource consumption when processing data objects (cf. Section 5.10).

7.5 Additional Features

Additional features describe extra functionalities to the actual data processing that are supported by the system *independently*. These functionalities are relevant for the security operationalization within the e-Commerce use cases in Section 2.1 to restrict access for the users. Our system comparison shows that systems such as BigDAWG, CloudMdsQL, ESTOCADA, Myria do not provide native support for data security or privacy to store or control access to data securely. This access must be specified externally by the deployment or network systems. Myria's documentation describes how to deploy in a cloud environment for Amazon Web Service to control a virtual firewall for inbound and outbound traffic using a *security group*. However, internal access controls are missing. Polypheny-DB focuses on implementing the General Data Protection Regulation for geographic distribution in its paper but does not describe any specific action. PolyBase connector for Microsoft SQL Server uses the security features of the intercepted SQL server. In addition to production relevant security, push-based queries are relevant for stream-based processing. None compared system supports this feature in which queries cannot be registered with the system and returned to the caller as the response when data states changes. This lack in query registration has its reason for not updating existing datasets on the level of the mediating system like BigIntegrator or MuSQLE, where the focus lies in integrating various sources in a read-only manner. Any comparative systems do not offer the ability to define a schema. Myria allows the sending and reading of *small* data sets, for which an input schema with *attribute names* and *type* can be specified. Further annotations are not possible or used by the mediating system.

7.6 Query Processing

For the most part, query processing in the systems is viewed very differently. Starting from the 4 table, we consider five distinguishing criteria in processing. Semantic closure describes how the system handles the diverging semantics of the data models and underlying query languages, such as the behaviour with NULL values or missing attributes. While BigIntegrator and RHEEM have their semantics for predefined operations fixed for RHEEM's RHEMLatin language, among others, FORWARD allows changing the operational semantics. Employing possible annotations to the SQL++ queries, the responsibility is shifted to the user, who chooses the interpretation of operators or above mentioned NULL values. This adaptability on the user level also supports MuSQLE. While a standards-compliant SQL interface defines

many operations upfront, it extends them with store-specific capabilities in the form of available functions. CloudMdsQL and BigDAWG refine the use of native query languages by embedding them in a language for parts of the grammar this specific language can contain. This non-agnostic system approach means that individual semantics must be dealt with within the system interaction since the BigDAWG and CloudMdsQL do not translate anything more. The behaviour of a query is dependent on the store executing it. The input and output are defined restrictively using migration from a language or semantic environment (for example, an Island within BigDAWG). CloudMdsQL proposes a fixed input and output typing via relations, whereas BigDAWG needs an additional migration step by the user. Complete semantic transparency is achieved in PolyBase. Any operations and queries are defined as Transact-SQL query, whose language semantics is already specified and executed by SQL server mediator in case of differences if needed. PolyBase avoids structural, semantic mismatches such as NULL value handling or missing attributes because the mapping of the external sources must be defined before. The user maps external sources to relations, resolves incompatibilities of types and structures (e.g. flatten hierarchical sources), and deals with missing values in the previous step.

When comparing the query processing feature of the considered multi/polystore system, our analysis shows that multi/polystores have different planning and execution approaches and leave some planning to the user. They all have in common that the system decompose incoming queries into sub-subqueries, each assigned to a different platform. The partial results are merged into a uniform format using a preferred join approach. BigIntegrator, ESTOCADA, Myria, PolyBase, FORWARD, MuSQLE, and RHEEM decompose a global query to those systems where data is located at the time of the query. Knowledge of current data locality, however, is not managed by everyone. CloudMdsQL shifts the knowledge of which platform contains the desired data to the user. Primary, the user already described the decomposition by the CloudMdsQL query language, and the system merges each subset, using the bind-join into a relational format. Compared to this, BigDAWG delegates the query to a planning component. The Planner optimizes for semantically equivalent operations. Previous benchmarks from a training phase allow the Planner to choose the right engine within the island language capsule (e.g. SQL) and limit migration only on these language and API levels. PolyBase also looks for a good execution plan for sub-branches of a query. The goal is to leave as large a branch of the query tree as possible in the databases, pushing down the operations in the tree into systems where the data resides. However, compared to BigDAWG, PolyBase does not have the approach of moving data between platforms. Data is loaded as needed in the PolyBase Mediator when a particular operation is impossible (e.g. a JOIN operation within a MongoDB). For planning in the SQL server where PolyBase settled, all data is managed in a relational format such as MuSQLE and ESTOCADA, requiring a previous mapping configuration. Instead of only considering migrations within a subset of engines sharing the same API or language as BigDAWG, RHEEM provides additional missing migration between platforms within the complete query. In combination with the unit cost of each operation, RHEEM determines necessary migration between platforms and looks up cost-efficient transformation paths within an available conversion graph and tries to select those platforms based on learned workload from previous executions. This learning-based and dependency resolution approach makes data placement dynamic on multiple platforms. RHEEM uses an extra conversion graph to transform data objects from one platform to another, even if a conversion is only available by shifting them to an intermediate platform first. Completely without model transformations, ESTOCADA, Myria and MuSQLE achieve processing. MuSQLE builds on the relational model and makes use of relational optimization. MuSQLE enumerates and selects the cost-efficient relational query. A graph search on the relational plan tries to find a suitable join plan, with connected subgraphs, selecting the platforms where these relational subqueries shall be applied. Each platform selection is determined by comparing potential data locality with the actual data placement on a platform. MuSQLE selects those platforms for a subquery by utilizing static cost functions or querying the cost estimation of

the underlying platform. Since the context remains in the relational model, the effort in the transformation is low but mainly characterized by the transfer costs. Comparing MuSQLLE with FORWARD and ESTOCADA, data is moved between platforms, but all of them keeps data within a relational context. FORWARD and ESTOCADA extends the relational model, planning and executing queries on views. The mediator maintains the state of the views on each platform or incrementally updates their materialization. Migrations within query processing do not take place.

7.7 Migration

Migration moves and transforms data objects from one or more sources to more targets, preserving the information content. In the context of Multi-/Polystores, data movement is supported in CloudMdsQL, Myria, Polypheny, PolyBase, BigDAWG, MuSQLLE and RHEEM. FORWARD and BigIntegrator are the only systems without any described migration task. They focus on closing the gap of the semantics differences between models by empowering query language or restricting expressive. FORWARD extends the syntax of the query language, considering the semi-structured facets, whereas BigIntegrator uses GQL and reduces queries only to use filters. In contrast, other systems concern the migration of datasets directly within the system and planning phase or have to be addressed by the user. These levels differ significantly in the transparency of migrations in the systems, conducted in multiple ways. BigDAWG forces the user to describe migrations within a query to manually resolve model dependency and differences. The available CAST operator can embed the original question, defining the translations from a given schema to another selected platforms/islands (PostgreSQL schema, SciDB schema, and text schema). In comparison, RHEEM allows an automatic transformation across multiple models. They use the conversion graph, which contains vertices of data models and edges with necessary transformations. The graph allows resolving single source multiple target paths to convert a source into required targets of an operation and their applied platforms, enabling to switch from one platform to another. PolyBase and MuSQLLE pursue this in on-demand strategy and query planning. While RHEEM adds the additional migration steps into the query plan afterwards, MuSQLLE does not calculate any migration costs but the execution within the plan using a given cost model. MuSQLLE moves data, contrary to transfer object for an operation completion such as a JOIN, coming from preceding steps (e.g. a selection). Since MuSQLLE mediates data on multiple relational systems, data model transformations are unnecessary compared to RHEEM. PolyBase, on the other hand, transfers data only when adding or as needed when the push-down of an operation is not possible.

8 OPEN CHALLENGES OF MULTI- AND POLYSTORES

As our comparison in Section 7 shows, there is a whole bunch of remaining challenges. A rich set of functional features is required. Thus, full read and write support is indispensable. However, query expressiveness (e.g. complex predicates, full joins, or aggregations) may be subject of a trade-off with the level of flexibility/adaptivity, the polyglot data store supports in terms of dynamic changes of the underlying system's topology. Users must be able to specify a (database) schema for their application requiring an appropriate DDL. Furthermore, it should be possible to use the system in the traditional pull-based way but also in a push-based style with sufficient support for data privacy and security.

Concluding, we see that the main challenges will be the following:

- (i) How can initial user/application requirements be mapped to capabilities of data stores and be derived as an appropriate initial topology?
- (ii) How to maintain a mapping with a changeable data store topology and schema?

- (iii) When should an adaption process be triggered and to which extent (e.g. Temporary or persistent data migration, Adding or removing data store to/from the topology)?
- (iv) How can the individual store capabilities be incorporated into the query processing while keeping their semantic completeness?
- (v) How can we monitor a polyglot system to enable autonomous decisions regarding adaption?

We consider it highly interesting to find out, whether or not the performance gains, which can be reached by topology changes, can be significantly higher than the overhead which must be spent in order to provide full adaptivity. Thus, we aim at the conceptualization, realization and evaluation of a polyglot persistence mediator supporting full adaptivity.

9 RELATED WORK

The emergence of NoSQL data stores since the late 2000s led to a multitude of database systems with their respective sweet spots and limitations. This landscape offers a wide range of opportunities to gain performance and functionality by choosing the right store for their needs. However, the amount of stores makes it difficult to keep an overview of all of them. For this reason, efforts were made in categorizing them, specifying their properties and providing a decision guide [38]. Unfortunately, this paper only focuses on single data store environments and do not cover the recent developments in the context of integrated polyglot settings.

The general idea of the previously valid *One size fits all* paradigm has been discussed by Stonebraker and Çetintemel [85] and extended with future data management solutions and research directions in [84] (e.g. using abstract data types or data federation).

Efforts to categorize the landscape of the emerging polyglot data stores have only been done by a few researchers so far. Tan et al. [87] focus on examining properties such as heterogeneity of query interfaces or data stores, flexibility regarding schema or extensibility, the autonomy of the underlying data stores, transparency and optimality. The work of Bondiombouy and Valduriez [14] targets query processing techniques and query interface capabilities of multistore systems. Properties such as adaptivity, system changes during runtime or topology adjustments have not been covered before.

10 CONCLUSION

In this paper, we addressed the issue of polyglot data stores for two representative use cases. Based on these, we outlined the need of systems capable of tackling a multitude of (even conflicting) requirements, which can not be provided by using only a single store technology in the backend. Therefore, we presented a selected set of Poly- and Multistores designed to incorporate a collection of heterogeneous data stores and evaluated their design/capabilities regarding our outlined requirements (Tables 2-3).

We concluded, that due to their static underlying data store landscape current Poly- and Multistores show a lack of adaptivity in their systems. In addition, most of the systems are not able to exploit the unique capabilities of their incorporated data stores. In our opinion, adaptivity is the key feature to guarantee future-proof, long-term usability of polyglot data stores in a world of ever changing application and data requirements. This results in a set of challenging research question regarding adaptivity as outline in Section 8.

Based on our expertise in the fields NoSQL database technology [38] and scalable data management [37], we aim at an automated polyglot persistence mediator which propose a declarative approach allowing the application to

initially specify an annotated data schema [80]. It should be possible to annotate schema elements by continuous non-functional (e.g. a certainly acceptable write latency), binary non-functional (e.g. atomic updates), or binary functional (e.g. consistency level read-your-writes required) requirements to express SLAs. Depending on the functional and non-functional capabilities of (NoSQL) systems, a smart algorithm is supposed to rank available systems and derive the best possible topology including a routing model which defines the mappings from schema elements to databases.

REFERENCES

- [1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. 2009. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB* 2, 1 (2009), 922–933.
- [2] Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing Joins in a Map-Reduce Environment. In *Proceedings of the 13th International Conference on Extending Database Technology (Lausanne, Switzerland) (EDBT '10)*. Association for Computing Machinery, New York, NY, USA, 99–110. <https://doi.org/10.1145/1739041.1739056>
- [3] 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. 2016. Rheem: Enabling Multi-Platform Task Execution. In *Proceedings of the 2016 International Conference on Management of Data (San Francisco, California, USA) (SIGMOD '16)*. Association for Computing Machinery, New York, New York, USA, 2069–2072. <https://doi.org/10.1145/2882903.2899414>
- [4] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. 2019. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *Proceedings of the 2019 International Conference on Management of Data (Amsterdam, Netherlands) (SIGMOD '19)*. Association for Computing Machinery, New York, New York, USA, 1660–1677. <https://doi.org/10.1145/3299869.3319895>
- [5] Sattam Alsubaiee, Yasser Altowim, Hotham Altwajry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. arXiv:1407.0454 [cs.DB]
- [6] Joseph H Anthony, Wooseok Choi, and Severin Grabski. 2006. Market reaction to e-commerce impairments evidenced by website outages. *International Journal of Accounting Information Systems* 7, 2 (2006), 60–78.
- [7] Apache Software Foundation. 2020. *Apache Arrow*. Retrieved June 22, 2020 from <https://arrow.apache.org/>
- [8] Apache Software Foundation. 2020. *Apache Drill*. Retrieved December 06, 2020 from <https://drill.apache.org/>
- [9] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (Melbourne, Victoria, Australia) (SIGMOD '15)*. Association for Computing Machinery, New York, New York, USA, 1383–1394. <https://doi.org/10.1145/2723372.2742797>
- [10] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*. 94–103.
- [11] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication steps for parallel query processing. *Journal of the ACM (JACM)* 64, 6 (2017), 1–58.
- [12] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data (Houston, Texas, USA) (SIGMOD '18)*. 221–230.
- [13] Raphaël Bonaque, Tien Duc Cao, Bogdan Cautis, François Goasdoué, Javier Letelier, Ioana Manolescu, Oscar Mendoza, Swen Ribeiro, Xavier Tannier, and Michaël Thomazo. 2016. Mixed-Instance Querying: A Lightweight Integration Architecture for Data Journalism. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1513–1516. <https://doi.org/10.14778/3007263.3007297>
- [14] Carlyna Bondiombouy and Patrick Valduriez. 2016. *Query Processing in Multistore Systems: an overview*. Research Report RR-8890. INRIA Sophia Antipolis - Méditerranée. 38 pages.
- [15] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*. Springer, 1–35.
- [16] Filip-Martin Brinkmann and Heiko Schuldt. 2015. 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 (Yokohama, Japan) (IDEAS '15)*. Association for Computing Machinery, New York, New York, USA, 81–89. <https://doi.org/10.1145/2790755.2790770>
- [17] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 69–80.
- [18] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (Chicago, Illinois, USA). IEEE, New York, New York, USA, 1789–1792.

- <https://doi.org/10.1109/IPDPSW.2016.138>
- [19] Thomas Clemen, Nima Ahmady-Moghaddam, Ulfia A Lenfers, Florian Ocker, Daniel Osterholz, Jonathan Ströbele, and Daniel Glake. 2021. Multi-Agent Systems and Digital Twins for Smarter Cities. In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 45–55.
 - [20] Sophie Cluet, Claude Delobel, Jérundefinedme Siméon, and Katarzyna Smaga. 1998. Your Mediators Need Data Conversion!. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) (SIGMOD '98). Association for Computing Machinery, New York, New York, USA, 177–188. <https://doi.org/10.1145/276304.276321>
 - [21] CoherentPaaS. 2014. *CoherentPaaS Project*. Retrieved March 30, 2020 from <http://coherentpaas.eu/>
 - [22] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. 2011. Relational cloud: A database-as-a-service for the cloud. (2011).
 - [23] S. Dasgupta, K. Coakley, and A. Gupta. 2016. Analytics-driven data ingestion and derivation in the AWESOME polystore. In *2016 IEEE International Conference on Big Data (Big Data)* (Washington, DC, USA). IEEE, New York, New York, USA, 2555–2564. <https://doi.org/10.1109/BigData.2016.7840897>
 - [24] Subhasis Dasgupta, Charles McKay, and Amarnath Gupta. 2017. Generating polystore ingestion plans – A demonstration with the AWESOME system. In *2017 IEEE International Conference on Big Data (Big Data)* (Boston, Massachusetts, USA). IEEE, New York, New York, USA, 3177–3179. <https://doi.org/10.1109/BigData.2017.8258297>
 - [25] David J. DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flaszka, and Jim Gramling. 2013. Split Query Processing in Polybase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) (SIGMOD '13). Association for Computing Machinery, New York, New York, USA, 1255–1266. <https://doi.org/10.1145/2463676.2463709>
 - [26] AnHai Doan, Alon Halevy, and Zachary Ives. 2012. *Principles of Data Integration*. Elsevier Science, Amsterdam, Netherlands.
 - [27] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. *ACM SIGMOD Record* 44, 2 (2015), 11–16.
 - [28] Thomas Eiter, Georg Gottlob, and Heikki Mannila. 1997. Disjunctive datalog. *ACM Transactions on Database Systems (TODS)* 22, 3 (1997), 364–418.
 - [29] Aaron J Elmore, Jennie Duggan, Michael Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. 2015. A demonstration of the bigdawg polystore system. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1908–1911.
 - [30] Ilir Fetai, Filip-Martin Brinkmann, and Heiko Schuldt. 2014. PolarDBMS: Towards a cost-effective and policy-based data management in the cloud. In *2014 IEEE 30th International Conference on Data Engineering Workshops* (Seattle, Washington, USA). IEEE, New York, New York, USA, 170–177. <https://doi.org/10.1109/ICDEW.2014.6818323>
 - [31] Ilir Fetai, Damian Murezzan, and Heiko Schuldt. 2015. Workload-driven adaptive data partitioning and distribution – The Cumulus approach. In *2015 IEEE International Conference on Big Data (Big Data)* (Santa Clara, California, USA). IEEE, New York, New York, USA, 1688–1697. <https://doi.org/10.1109/BigData.2015.7363940>
 - [32] Ilir Fetai, Alexander Stiemer, and Heiko Schuldt. 2017. QuAD: A quorum protocol for adaptive data management in the cloud. In *2017 IEEE International Conference on Big Data (Big Data)* (Boston, Massachusetts, USA). IEEE, New York, New York, USA, 405–414. <https://doi.org/10.1109/BigData.2017.8257952>
 - [33] Marc Friedman, Alon Y Levy, Todd D Millstein, et al. 1999. Navigational plans for data integration. *AAAI/IAAI* 1999 (1999), 67–73.
 - [34] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. 2016. The BigDAWG polystore system and architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (Waltham, Massachusetts, USA). IEEE, New York, New York, USA, 1–6. <https://doi.org/10.1109/CSC.2011.6138543>
 - [35] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom. 1997. The TSIMMIS approach to mediation: Data models and languages. *JGIS* 8, 2 (1997), 117–132.
 - [36] Felix Gessert and Norbert Ritter. 2015. Polyglot persistence. *Datenbank-Spektrum* 15, 3 (2015), 229–233.
 - [37] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Erik Witt, Eiko Yoneki, and Norbert Ritter. 2017. Quaestor: Query Web Caching for Database-as-a-Service Providers. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1670–1681. <https://doi.org/10.14778/3137765.3137773>
 - [38] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. 2017. NoSQL database systems: a survey and decision guidance. *CSRD* 32, 3 (2017), 353–365.
 - [39] Ivan Giangreco and Heiko Schuldt. 2016. ADAM_{pro}: Database Support for Big Multimedia Retrieval. *Datenbank Spektrum* 16 (2016), 17–26. <https://doi.org/10.1007/s13222-015-0209-y>
 - [40] V. Giannakouris, N. Papailiou, D. Tsoumakos, and N. Koziris. 2016. MuSQL: Distributed SQL query execution over multiple engine environments. In *2016 IEEE International Conference on Big Data (Big Data)* (Washington, DC, USA). IEEE, New York, New York, USA, 452–461. <https://doi.org/10.1109/BigData.2016.7840636>
 - [41] Seth Gilbert and Nancy Lynch. 2002. Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (2002), 51–59.
 - [42] Daniel Glake, Fabian Panse, Norbert Ritter, Thomas Clemen, and Ulfia Lenfers. 2021. Data Management in Multi-Agent Simulation Systems. *BTW 2021* (2021).
 - [43] Daniel Glake, Norbert Ritter, and Thomas Clemen. 2020. Utilizing spatio-temporal data in multi-agent simulation. In *2020 Winter Simulation Conference (WSC)*. IEEE, 242–253.

- [44] Daniel Glake, Norbert Ritter, Florian Ocker, Nima Ahmady-Moghaddam, Daniel Osterholz, Ulfia Lenfers, and Thomas Clemen. 2021. Hierarchical Semantics Matching For Heterogeneous Spatio-temporal Sources. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 565–575.
- [45] Georg Gottlob and Roberto V Zicari. 1988. Closed World Databases Opened Through Null Values.. In *VLDB*, Vol. 88. 50–61.
- [46] Ankush M Gupta, Vijay Gadepally, and Michael Stonebraker. 2016. Cross-engine query execution in federated database systems. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.
- [47] Hakan Hacigümüş, Jagan Sankaranarayanan, Junichi Tatemura, Jeff LeFevre, and Neoklis Polyzotis. 2013. Odyssey: a multistore system for evolutionary analytics. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1180–1181.
- [48] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001), 270–294.
- [49] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. 2014. Demonstration of the Myria Big Data Management Service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 881–884. <https://doi.org/10.1145/2588555.2594530>
- [50] W. Hasselbring and G. Steinacker. 2017. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (Gothenburg, Sweden)*. IEEE, New York, New York, USA, 243–246. <https://doi.org/10.1109/ICSAW.2017.11>
- [51] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2016. PipeGen: Data Pipe Generator for Hybrid Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (Santa Clara, California, USA) (SoCC '16)*. Association for Computing Machinery, New York, New York, USA, 470–483. <https://doi.org/10.1145/2987550.2987567>
- [52] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2018. PolyBench: The First Benchmark for Polystores. In *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence - 10th TPC Technology Conference, TPCTC 2018 (Rio de Janeiro, Brazil) (Lecture Notes in Computer Science)*, Raghunath Nambiar and Meikel Poess (Eds.), Vol. 11135. Springer, Cham, Swiss, 24–41. https://doi.org/10.1007/978-3-030-11404-6_3
- [53] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. 2015. Utilizing IDs to Accelerate Incremental View Maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (Melbourne, Victoria, Australia) (SIGMOD '15)*. Association for Computing Machinery, New York, New York, USA, 1985–2000. <https://doi.org/10.1145/2723372.2750546>
- [54] Karamjit Kaur and Rinkle Rani. 2015. A smart polyglot solution for big data in healthcare. *IT Professional* 17, 6 (2015), 48–55.
- [55] Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez, Ricardo Jimenez-Peris, Raquel Pau, and José Pereira. 2016. The CloudMdsQL Multistore System. In *Proceedings of the 2016 International Conference on Management of Data (San Francisco, California, USA) (SIGMOD '16)*. Association for Computing Machinery, New York, New York, USA, 2113–2116. <https://doi.org/10.1145/2882903.2899400>
- [56] Boyan Kolev, Raquel Pau, Oleksandra Levchenko, Patrick Valduriez, Ricardo Jiménez-Peris, and José Pereira. 2016. Benchmarking polystores: The CloudMdsQL experience. In *2016 IEEE International Conference on Big Data (Big Data) (Washington, DC, USA)*. IEEE, New York, New York, USA, 2574–2579. <https://doi.org/10.1109/BigData.2016.7840899>
- [57] Pavlos Kranas, Boyan Kolev, Oleksandra Levchenko, Esther Pacitti, Patrick Valduriez, Ricardo Jiménez-Peris, and Marta Patiño-Martinez. 2021. Parallel query processing in a polystore. *Distributed and Parallel Databases* 39, 4 (2021), 939–977.
- [58] Sebastian Kruse, Zoi Kaoudi, Bertty Contreras-Rojas, Sanjay Chawla, Felix Naumann, and Jorge-Arnulfo Quiané-Ruiz. 2020. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *The VLDB Journal* 29, 6 (2020), 1287–1310.
- [59] Sebastian Kruse, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Sanjay Chawla, Felix Naumann, and Bertty Contreras-Rojas. 2019. Optimizing Cross-Platform Data Movement. In *2019 IEEE 35th International Conference on Data Engineering (ICDE) (Macao, Macao, Macao)*. IEEE, New York, New York, USA, 1642–1645. <https://doi.org/10.1109/ICDE.2019.00162>
- [60] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J. Carey. 2014. MISO: Souping up Big Data Query Processing with a Multistore System. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1591–1602. <https://doi.org/10.1145/2588555.2588568>
- [61] Ulfia A Lenfers, Nima Ahmady-Moghaddam, Daniel Glake, Florian Ocker, Daniel Osterholz, Jonathan Ströbele, and Thomas Clemen. 2021. Improving model predictions—integration of real-time sensor data into a running simulation of an agent-based model. *Sustainability* 13, 13 (2021), 7000.
- [62] Alon Levy, Anand Rajaraman, and Joann Ordille. 1996. *Querying heterogeneous information sources using source descriptions*. Technical Report. Stanford InfoLab.
- [63] Harold Lim, Yuzhang Han, and Shivnath Babu. 2013. How to Fit when No One Size Fits.. In *CIDR*, Vol. 4. Citeseer, 35.
- [64] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs—Opportunities and Limitations for NoSQL-. In *CIDR*.
- [65] Jiaheng Lu and Irena Holubová. 2019. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 55.
- [66] Ji Lucas, Yasser Idris, Bertty Contreras-Rojas, Jorge-Arnulfo Quiané-Ruiz, and Sanjay Chawla. 2018. RheemStudio: Cross-Platform Data Analytics Made Easy. In *2018 IEEE 34th International Conference on Data Engineering (ICDE) (Paris, France)*. IEEE, New York, New York, USA, 1573–1576. <https://doi.org/10.1109/ICDE.2018.00179>
- [67] Antonio Maccioni, Edoardo Basili, and Riccardo Torlone. 2016. QUEPA: QUerying and Exploring a Polystore by Augmentation. In *Proceedings of the 2016 International Conference on Management of Data (San Francisco, California, USA) (SIGMOD '16)*. Association for Computing Machinery, New York, New York, USA, 2133–2136. <https://doi.org/10.1145/2882903.2899393>

- [68] Tim Mattson, Vijay Gadepally, Zuohao She, Adam Dziedzic, and Jeff Parkhurst. 2017. Demonstrating the BigDAWG Polystore System for Ocean Metagenomics Analysis. In *8th Biennial Conference on Innovative Data Systems Research (CIDR 2017)* (Chaminade, California, USA). CIDR Conference, Asilomar, California, USA, 9. <http://cidrdb.org/cidr2017/papers/p120-mattson-cidr17.pdf>
- [69] Sourav Mazumder. 2016. Big Data Tools and Platforms. In *Big Data Concepts, Theories, and Applications*, Shui Yu and Song Guo (Eds.). Springer International Publishing, Cham, Swiss, 29–128. https://doi.org/10.1007/978-3-319-27763-9_2
- [70] John Meehan, Stan Zdonik, Shaobo Tian, Yulong Tian, Nesime Tatbul, Adam Dziedzic, and Aaron Elmore. 2016. Integrating real-time and batch processing in a polystore. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. <https://doi.org/10.1109/HPEC.2016.7761585>
- [71] Meteor Software. 2020. *MeteorJS*. Retrieved March 31, 2020 from <https://www.meteor.com>
- [72] Anisoara Nica. 2011. A call for order in search space generation process of query optimization. In *2011 IEEE 27th International Conference on Data Engineering Workshops* (Hannover, Germany). IEEE, New York, New York, USA, 4–9. <https://doi.org/10.1109/ICDEW.2011.5767651>
- [73] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). arXiv:1405.3631 <https://arxiv.org/abs/1405.3631>
- [74] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems, 4th Edition*. Springer. <https://doi.org/10.1007/978-3-030-26253-2>
- [75] Erhard Rahm, Gunter Saake, and Kai-Uwe Sattler. 2015. *Verteiltes und paralleles Datenmanagement*. Springer.
- [76] Luca Rossetto, Ivan Giangreco, Claudiu Tanase, and Heiko Schuldt. 2016. VitriV: A Flexible Retrieval Stack Supporting Multiple Query Modes for Searching in Multimedia Collections. In *Proceedings of the 24th ACM International Conference on Multimedia* (Amsterdam, The Netherlands) (*MM '16*). Association for Computing Machinery, New York, New York, USA, 1183–1186. <https://doi.org/10.1145/2964284.2973797>
- [77] Pramod J. Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, Boston, Massachusetts, USA. <https://books.google.com/books?id=AyY1a6-k3PIC>
- [78] Mohammed Saeed, Mauricio Villarroel, Andrew T Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H Kyaw, Benjamin Moody, and Roger G Mark. 2011. Multiparameter Intelligent Monitoring in Intensive Care II (MIMIC-II): a public-access intensive care unit database. *Critical care medicine* 39, 5 (2011), 952.
- [79] David Salomon. 2003. *Data privacy and security*. Springer, New York, New York, USA. <https://doi.org/10.1007/978-0-387-21707-9>
- [80] Michael Schaarschmidt, Felix Gessert, and Norbert Ritter. 2015. Towards Automated Polyglot Persistence. In *Proceedings of the 2015 German National Database Conference (BTW'15)* (Hamburg, Germany) (*LNI*), Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.), Vol. P-241. GI, Bonn, Germany, 73–82. <https://subs.emis.de/LNI/Proceedings/Proceedings241/article46.html>
- [81] Alkis Simitis, Kevin Wilkinson, Malu Castellanos, and Umeshwar Dayal. 2012. Optimizing Analytic Data Flows for Multiple Execution Engines. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) (*SIGMOD '12*). Association for Computing Machinery, New York, New York, USA, 829–840. <https://doi.org/10.1145/2213836.2213963>
- [82] Dimitrios-Emmanuel Spanos, Periklis Stavrou, and Nikolas Mitrou. 2012. Bringing relational databases into the semantic web: A survey. *Semantic Web* 3, 2 (2012), 169–209.
- [83] Alexander Stiemer, Ilir Fetaj, and Heiko Schuldt. 2016. Analyzing the performance of data replication and data partitioning in the cloud: The BEOWULF approach. In *2016 IEEE International Conference on Big Data (Big Data)* (Chicago, Illinois, USA). IEEE, New York, New York, USA, 2837–2846. <https://doi.org/10.1109/BigData.2016.7840932>
- [84] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. 2007. One Size Fits All? Part 2: Benchmarking Studies. In *Third Biennial Conference on Innovative Data Systems Research (CIDR'07)* (Asilomar, California, USA). CIDR Conference, Asilomar, California, USA, 173–184. <http://cidrdb.org/cidr2007/papers/cidr07p20.pdf>
- [85] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *Proceedings of the 21st International Conference on Data Engineering, ICDE'05* (Tokyo, Japan), Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE, Washington, DC, USA, 2–11. <https://doi.org/10.1109/ICDE.2005.1>
- [86] Uta Störl, Meike Klettke, and Stefanie Scherzinger. 2020. NoSQL Schema Evolution and Data Migration: State-of-the-Art and Opportunities. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT*. OpenProceedings.org, 655–658. <https://doi.org/10.5441/002/edbt.2020.87>
- [87] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. 2017. Enabling query processing across heterogeneous data models: A survey. In *2017 IEEE International Conference on Big Data (Big Data)* (Boston, Massachusetts, USA). IEEE, New York, New York, USA, 3211–3220. <https://doi.org/10.1109/BigData.2017.8258302>
- [88] Andrew S Tanenbaum and Maarten Van Steen. 2007. *Distributed systems: principles and paradigms*. Prentice-Hall.
- [89] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) (*SOCC '13*). Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages. <https://doi.org/10.1145/2523616.2523633>
- [90] Marco Vogt, Nils Hansen, Jan Schönholz, David Lengweiler, Isabel Geissmann, Sebastian Philipp, Alexander Stiemer, and Heiko Schuldt. 2020. Polypheny-DB: towards bridging the gap between polystores and HTAP systems. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer, 25–36.

- [91] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2017. Icarus: Towards a multistore database system. In *2017 IEEE International Conference on Big Data (Big Data)* (Boston, Massachusetts, USA). IEEE, New York, New York, USA, 2490–2499. <https://doi.org/10.1109/BigData.2017.8258207>
- [92] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2018. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *2018 IEEE International Conference on Big Data (Big Data)* (Seattle, Washington, USA). IEEE, New York, New York, USA, 3364–3373. <https://doi.org/10.1109/BigData.2018.8622353>
- [93] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suci, Andrew Whitaker, and Shengliang Xu. 2017. The Myria Big Data Management and Analytics System and Cloud Services. In *8th Biennial Conference on Innovative Data Systems Research (CIDR 2017)* (Chaminade, California, USA). CIDR Conference, Asilomar, California, USA, 11. <http://cidrdb.org/cidr2017/papers/p37-wang-cidr17.pdf>
- [94] Julius Weyl, Daniel Blake, and Thomas Clemen. 2018. Agent-Based Traffic Simulation at City Scale with MARS. In *Proceedings of the Agent-Directed Simulation Symposium* (Baltimore, Maryland) (*ADS'18*). Society for Computer Simulation International, San Diego, California, USA, Article 2, 9 pages. <https://dl.acm.org/doi/abs/10.5555/3212922.3287082>
- [95] Daya C Wimalasuriya and Dejing Dou. 2010. Ontology-based information extraction: An introduction and a survey of current approaches.
- [96] Raymond C.-W. Wong and Ada W.-C. Fu. 2010. *Privacy-preserving Data Publishing: An Overview*. Morgan & Claypool Publishers, Williston, Vermont, USA. <https://books.google.com/books?id=yFzGVp7xYLMC>
- [97] Katherine Yu, Vijay Gadepally, and Michael Stonebraker. 2017. Database engine integration and performance analysis of the BigDAWG polystore system. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (Waltham, Massachusetts, USA). IEEE, New York, New York, USA, 1–7. <https://doi.org/10.1109/HPEC.2017.8091081>
- [98] Dongxiang Zhang, Chee-Yong Chan, and Kian-Lee Tan. 2014. An efficient publish/subscribe index for e-commerce databases. *Proceedings of the VLDB Endowment* 7, 8 (2014), 613–624.
- [99] Minpeng Zhu and Tore Risch. 2011. Querying combined cloud-based and relational databases. In *2011 International Conference on Cloud and Service Computing* (Hong Kong, China). IEEE, New York, New York, USA, 330–335. <https://doi.org/10.1109/CSC.2011.6138543>