PolyMigrate: Dynamic Schema Evolution and Data Migration in a Distributed Polystore

Alexander Stiemer[1(✉)], Marco Vogt[1], Heiko Schuldt[1], and Uta Störl[2]

[1] Databases and Information Systems Research Group, University of Basel, Basel, Switzerland
{alexander.stiemer,marco.vogt,heiko.schuldt}@unibas.ch
[2] Department of Computer Science, Darmstadt University of Applied Sciences, Darmstadt, Germany
uta.stoerl@h-da.de

Abstract. In the last years, polystore databases have been proposed to cope with the challenges stemming from increasingly dynamic and heterogeneous workloads. A polystore database provides a logical schema to the application, but materializes data in different data stores, different data models, and different physical schemas. When the access pattern to data changes, the polystore can decide to migrate data from one store to the other or from one data model to another. This necessitates a schema evolution in one or several data stores and the subsequent migration of data. Similarly, when applications change, the global schema might have to be changed as well, with similar consequences on local data stores in terms of schema evolution and data migration. However, the aspect of schema evolution in a polystore database has so far largely been neglected. In this paper, we present the challenges imposed by schema evolution and data migration in Polypheny-DB, a distributed polystore database. With our work-in-progress approach called *PolyMigrate*, we show how schema evolution and data migration affect the different layers of a distributed polystore and we identify different approaches to effectively and efficiently propagate these changes to the underlying stores.

Keywords: Polystore databases · Schema evolution · Data migration

1 Introduction

For several decades, relational databases had a monopoly in the data management layer of information systems. This has changed in the course of the 2000s with the proliferation of novel types of applications, data, and access patterns such as analytical processing on structured data or social graphs [17]. As a consequence, a large variety of different data stores has been introduced, from column stores over key-value stores to document and graph databases. As long as applications are based on rather homogeneous data sets and workloads, these systems

V. Gadepally et al. (Eds.): Poly 2020/DMAH 2020, LNCS 12633, pp. 42–53, 2021.
https://doi.org/10.1007/978-3-030-71055-2_4

are well suited. However, in cases of highly heterogeneous data and/or large fluctuations in the access patterns of applications, not even these specialized systems would be able to provide optimal support.

For this reason, the last years have seen the advent of polystore databases [18], which combine several different, heterogeneous data stores underneath a joint interface. Depending on the type of data to be managed or the access patterns, they may decide to deploy several different data stores and distribute (possibly also partly replicate) data among these stores.

In [20], we have introduced Polypheny-DB, a novel distributed polystore database. Polypheny-DB considers distribution at two levels: At *global level*, data is fragmented and replicated (the latter to increase availability) in order to allocate it to different sites in a global network. Hence, fragmentation, replication, and allocation aims at bringing such data items together that are frequently accessed jointly. The allocation then has to guarantee that the fragments are placed close to their corresponding applications in order to minimize access latency. At *local level*, each site runs an independent polystore that can decide unilaterally, based on a local cost model, which data stores to provide, how to distribute data across these data stores, and how to process queries.

Assume, as an example for an organization running such a distributed polystore, an international auction house with databases and compute centers distributed around the globe (see [20] for more details). The auction house has to jointly deal with several workloads such as Online Transaction Processing (OLTP) (for the actual auctions), Online Analytical Processing (OLAP) (for analyzes of past auctions), graph queries (for recommendations to their customers), and finally also multimedia similarity search queries (to find items and thus auctions based on the visual appearance of the former).

While polystores usually assume the database schema to be static, this is not always the case in practice. Schema evolution—one of the "top ten fears" about the future of databases, according to Stonebraker [16]—needs to be taken into account also in polystores. In the auction house example, changes in the product recommendation engine or additional/revised legal requirements may lead to changes of the logical database schema ("external" reasons, from the polystore's perspective). Because shutting down the entire business of the auction house is not an option, these schema changes and the subsequent data migrations have to be performed efficiently online, without any downtime. In addition, there are also schema changes caused by internal data reorganisation in the local polystores, for instance based on workload changes ("internal" reasons).

In this paper, we introduce *PolyMigrate*, a work-in-progress extension to Polypheny-DB that considers schema evolution and data migration derived from schema changes in a distributed polystore. In particular, we analyze the different layers on which schema changes can originate and how they need to be propagated downwards through the different layers of the polystore (or why and when they do not need to be propagated). To the best of our knowledge, this is the first attempt to address schema evolution and data migration in a polystore context.

The contribution of this paper is threefold: (i) We identify *why* and *where* in the Polypheny-DB stack schema changes might occur and *what* changes need to be considered. (ii) We discuss several options as to *when* these changes have to be propagated downwards in the Polypheny-DB stack to the local data stores, thereby taking into account that potentially large volumes of data might have to be migrated. (iii) We analyze *how* the propagation of schema changes and the migration of data have to be implemented.

The remainder is structured as follows: In Sect. 2 we review related work. Schema evolution and data migration in Polypheny-DB are discussed in Sect. 3. Section 4 presents examples and Sect. 5 concludes.

2 Background and Related Work

In what follows, we introduce basic notions and concepts from database schema evolution, data migration, and multi- and polystores, and survey related work.

2.1 Schema Evolution and Data Migration

Schema evolution in databases is a long investigated and still current topic. In schema-flexible database systems (e.g., NoSQL database systems), schema changes do not have to be executed immediately, but can be divided into two separate steps: *schema evolution* and *data migration*.

Schema evolution describes the changes to the schema without immediately executing them on the data. In this paper, we will discuss *why* and *where* (Sect. 3.1) schema evolution in a distributed polystore is triggered. We also describe *what* types of schema evolution operations occur in Polypheny-DB. We use the operations introduced in [14] and extended for multi-model data in [7]. There are operations on single schema objects (e.g., `add`, `rename`, and `delete`) and operations on multiple schema objects (e.g., `copy`, `move`, `split`, and `merge`).

Data migration follows the schema evolution and is traditionally carried out eagerly, upgrading all legacy data. Yet, in the context of Cloud-hosted data backends, eager migration can be rather costly. Thus, lazy migration may be more cost-efficient, as legacy data objects are only migrated on-the-fly in case they are actually accessed by the application. The downside is that lazy migration introduces a runtime overhead on reads and writes [12]. A compromise between the two competing goals of minimizing latency and migration costs can be reached by migrating hot data predictively [6]. The possible applications of these techniques in Polypheny-DB are discussed in Sect. 3.2.

Related Work. The co-evolution of schemas and the associated XML documents is addressed in [4] and [9], for example. Also in relational databases the handling of different versions in single database systems becomes more and more important [1,5,15]. The schema flexibility of NoSQL database systems brings new challenges for the schema evolution, which are analyzed in [2,12,14] for different types of NoSQL database systems. However, all these approaches only discuss

schema evolution in the context of single store databases. Polystore databases lead to new challenges for schema evolution and data migration. In [7], the challenges for multi-model data [8] are outlined in a vision paper. In this paper, we will discuss these challenges and solutions in a more concrete and detailed way using Polypheny-DB.

2.2 Polystore Databases

To efficiently deal with heterogeneous workloads produced by today's zoo of applications, a new generation of database systems has been developed. In the following, we use the taxonomy introduced in [18] to discuss multi- and polystore databases.

When data (or parts thereof) needs to be accessed by different applications, some query languages might be better suited than others—depending on the concrete requirements of the respective applications. *Polyglot persistence* [11,13] addresses this problem and aims at choosing the best suited query language for a concrete use case. This goes back to the concept of *polyglot programming*.

A *polyglot database system* uses a set of *homogeneous* data stores and exposes multiple query interfaces and languages [18]. A *multistore database system*, in contrast, manages data in *heterogeneous* data stores, but offers a common query interface to the outside, as well as only one query language. *Polystore databases* combine the advantages of both polyglot and multistore systems.

Related Work. An example for a multistore system is *Icarus* [19]. Compared to other multistore systems, Icarus always stores all data on all underlying stores and executes incoming queries on the store with the best characteristics for this type of query. Since all data is stored on all stores and Icarus only supports data stores with an SQL interface, there is no need for complex schema and data migrations. *BigDAWG* [3] is a polystore system which organizes heterogeneous data stores into "islands". Each island has an associated query language and data model, (e.g., a relational island is based on the relational data model and exposes an SQL interface). If a query accesses data distributed over different islands, inter-island queries are resolved by migrating data between the islands. *Hybrid.poly* [10] is an in-memory polystore, which is queried using a extended SQL interface. It allows the execution of complex analytical queries on non-relational data being combined with relational data. With *Polypheny-DB* [20] we have introduced a polystore system that does not only provide access to data stored in different kinds of data stores and data models independent of a query language, it can also be deployed in a distributed fashion. Polypheny-DB supports different types of underlying data stores including key-value stores, document stores, plain CSV-files, and relational databases.

3 Schema Evolution and Data Migration in Polypheny-DB

The distributed polystore Polypheny-DB distinguishes two layers (see Fig. 1): At the *global level* ($\mathcal{G}$), data is distributed across several sites in a network, i.e., the global layer consists of interconnected instances. For this, the schema is *fragmented* and *replicated*. The resulting schema fragments are then assigned to sites in the system (*allocation*). At the *local level* ($\mathcal{L}$), each Polypheny-DB site then manages data locally in a polystore ($\mathcal{P}$), i.e., in different data models and data stores. The main objective of the local level is to improve the performance for heterogeneous workloads. There is no communication between the individual data stores which are all considered as black boxes.

Further, applications and clients are not supposed to access the individual data stores directly. Consequently, we follow a top-down approach for data access, which also holds for schema evolution and data migration. Moreover, Polypheny-DB is agnostic to optimizations at the physical level inside the data stores.

To summarize, the global level spans over all instances of the polystore, while the local level only spans over the heterogeneous data stores of a specific polystore instance. Therefore, we distinguish three different types of schemas depicted in Fig. 1: (i) The *global schema* $\mathcal{G}$ (as seen by the applications), (ii) the *local schemas* $\mathcal{L}$ (the schema of an individual polystore instance as a result of fragmentation, replication, and allocation on global level), and (iii) the *physical schemas* $\mathcal{P}$ (the actual schema of an underlying data store).

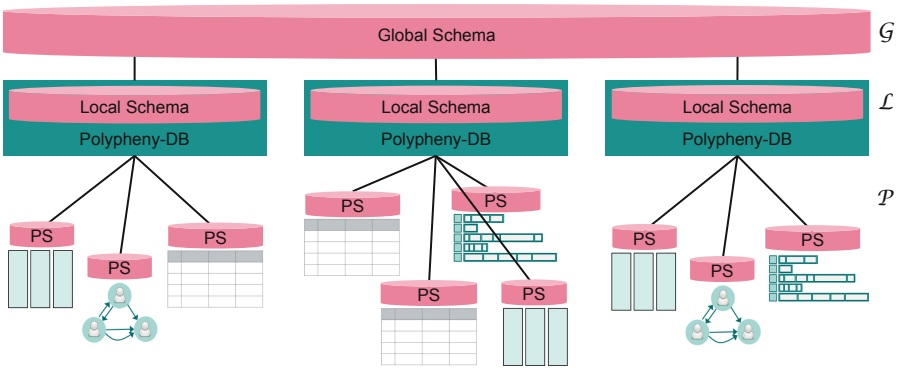

Fig. 1. Different types of schemas in Polypheny-DB. The *global schema* is visible to the application, the *local schema* of an individual polystore instance is the result of the fragmentation and replication process, and the *physical schemas* (PS) are the actual schemas of the data stores. $\mathcal{G}$ denotes the global, $\mathcal{L}$ the local, and $\mathcal{P}$ the physical level.

3.1 Schema Evolution

In the following, we introduce PolyMigrate and analyze the *why*, *where*, *what*, *when*, and *how* regarding schema evolution and data migration in Polypheny-DB.

The "Why": Schema evolution in Polypheny-DB might take place due to one or several of the following three reasons: (i) As a result of an "external" activity like a new version of the application software (e.g., by introducing new features which require additional attributes). (ii) As consequence of an "internal" optimization which leads to a revision of the fragmentation or replication decision at global level. This immediately affects the local schema of at least one of the sites. Internal optimizations might take place due to the attempt to minimize data access latency or request response time. They could also be the result of the relocation of data that is frequently queried together (e.g., by bringing data to the same local site) in order to optimize queries. Other reasons are the minimization of costs (e.g., in the Cloud) or the reduction of the replication degree. And (iii) as a result of the optimization within one polystore instance and the local distribution and allocation of data to one of the underlying data stores.

The "Where": Schema evolution and subsequent migration takes place at each of the three levels in Polypheny-DB. At global level $\mathcal{G}$, schema evolution is triggered by the applications. The fragmentation and replication applied at global level leads to an evolution of one or several of the schemas at local level $\mathcal{L}$. Such inter-model evolution occurs because of global data movements. And finally, the optimizations within a polystore instance (intra-model evolution) lead to the evolution of the individual physical schemas $\mathcal{P}$ of its underlying data stores.

The "What": Schema changes at global level $\mathcal{G}$ are based on the operations introduced in Sect. 2.1 and might thus address only individual schema objects or multiple schema objects jointly. This includes the addition or removal of attributes and alterations of the attribute names or their types. At entity level, the addition of new entities, their deletion, and the split or merge of existing entities need to be considered. Furthermore, changes in the integrity constraints being part of the schema are relevant as well. All these changes at global level need to be properly propagated downwards to the local level.

At local level $\mathcal{L}$, in addition to the changes initialized at global level, schema changes might occur due to a revision of the fragmentation and replication decision and the allocation to sites. At schema level, this means that individual attributes or entire entities will be removed or added from a local site. The allocation is usually determined by the global cost model of Polypheny-DB and is triggered, for instance, by changes in the applications' workloads. Similarly, at the data store level $\mathcal{P}$, aside of schema changes imposed by one of the two layers on top, also internal cost-based optimizations of the local polystore may lead to a re-distribution of entities and a re-allocation of data across its data stores.

3.2 Data Migration

This section discusses aspects related to data migration in PolyMigrate as a result of schema evolution.

The "When": After schema changes have been externally triggered or internally decided, the question is how they are propagated to the next lower level, and how data is handled. For this, five different options can be identified.

First, schema changes and data can be migrated *eagerly*. This means that all the underlying schemas are immediately updated (within the same transaction), and all affected data is migrated to the new schema and/or re-located to the new sites and stores. This guarantees that the entire polystore with all its local sites is up to date at any point in time. However, this comes with significant disadvantages on the performance of the overall system, as all conflicting requests need to be blocked until the migration has successfully completed.

Second, the migration can take place *lazily*. In this case, data is only migrated when accessed. Data which is not requested can still stay in the old schema and at the old site. A request to non-migrated data is temporarily paused and the migration activities for the requested data are triggered. As soon as the migration has succeeded, the request is resumed and the requested data is returned. Therefore, such a lazy migration increases the access latency for the first access to a data item after a schema change, but it does not require a costly eager migration [6,12]. However, in lazy migration, data in the old and the new schema might temporarily co-exist. Even several versions of the revised schema could stay in the system when schema changes appear frequently. It has to be noted that lazy migration is only well suited for OLTP workloads. OLAP queries would trigger a complete migration of the data, resulting in an enormous impact on query latency (and a query would have to be paused for a significant duration).

Third, a compromise between the two competing goals of minimizing latency and migration costs can be reached by a *proactive* strategy. Data is migrated in a background process with the objective that the migration is completed as soon as data is requested. This can be done, for example, by predicting future data accesses based on access statistics and a suitable prediction function [6]. In case data is requested that has not been migrated yet, the lazy approach is applied. Proactive migration is also more suitable for OLTP than for OLAP workloads.

A fourth option is to refrain from physically implementing the schema changes at all. This can be done by using *query rewrites* instead. Any access to a data item (specified in terms of the new schema) will be addressed by a re-written query that exploits the old (and still physically present) schema. While this option has the least effects on the run-time behavior of the system as all changes are not propagated, it is limited to a subset of schema manipulations only (e.g., a `rename` of an attribute, or the `split` or `merge` of an entity). Furthermore, the query rewrite quickly becomes complex if several such changes occur consecutively. In this case, also query re-write would have to be cascaded accordingly.

Fifth, the *incremental* approach combines lazy schema migration and query re-write. The incremental approach propagates schema changes and migrates data when the system's load is below a certain threshold. This ensures that regular operations are not unnecessarily affected by migration activities. For this, data is sequentially migrated, starting from the most recently migrated data item. If the system's load exceeds the threshold, the incremental migration is paused. Access to data that are not yet migrated is subject to a query re-write.

While all these approaches are complementary, they can be seamlessly combined in one system. First, all local polystores are autonomous and can independently implement the schema changes imposed by the global level. One polystore, for example, might decide to perform schema changes eagerly, a second one follows a lazy approach, and a third one relies on query re-writing. Similarly, even within one polystore, several approaches may co-exist (e.g., using a proactive approach for a subset of the data and the incremental approach for the rest).

The "How": Addresses the propagation of schema changes downwards to the local data stores and in particular the handling of data migration.

If schema changes originate at global level $\mathcal{G}$, they are propagated down to the local level $\mathcal{L}$. There, they can be subsequently implemented; in this case, data needs to be migrated accordingly. Alternatively, depending on the nature of the changes, the schema changes are not implemented (and data is not migrated). But this requires the re-write of every query at the local level. Similarly, if the changes are implemented at level $\mathcal{L}$, they will be propagated to the data store level $\mathcal{P}$. There, they can again be enacted, with subsequent data migration, or be avoided using query re-writes.

If schema changes originate at the $\mathcal{L}$ level (i.e., due to a revised fragmentation, replication, or allocation), they will be propagated down to the underlying data stores $\mathcal{P}$. These changes then require data to be migrated once they are implemented in the physical schema, or they are again handled by query re-writes.

Finally, schema changes and data migration activities might start at and only affect one of the data stores at $\mathcal{P}$ level at a site where they will then be enacted.

4 Sample Scenarios and Recommendations

In this section we illustrate the techniques introduced in Sect. 3 with concrete examples from the online auction house scenario mentioned in Sect. 1.

4.1 Global Schema Changes

Archiving: Consider the fictitious international auction house and assume that each completed auction will be archived, for legal and auditing reasons. However, the information that needs to be archived is only a subset of all the information collected on the good which was sold, the buyer and the seller. Therefore, in an attempt to condense the information and to reduce the number of relations per

auction while still meeting all legal auditing requirements, a denormalization of the schema at global level $\mathcal{G}$ is applied for archival purposes:

```
COPY     U.last_name, U.first_name, U.zip_code, U.country,
         S.last_name, S.first_name, S.zip_code, S.country,
         B.amount, B.timestamp
TO       A
  WHERE  A.id = B.auction  AND  U.id = B.user  AND  S.id = A.user
```

where `A` is the relation holding the auction, `B` holds the final bid, `U` is the buyer relation and `S` the seller. Since the archive is only highly infrequently accessed (if at all), schema evaluation is propagated in an incremental approach and the local schema at $\mathcal{L}$ is only updated when data is accessed.

Data Protection: Assume that new data protection laws in one country where the auction house does business require a change in the data distribution plan. Currently, sensitive user data is stored at the cheapest Polypheny-DB instance $\mathcal{L}_{cheap}$ which happens to be located in another country. Since data protection regulations require sensitive data not to leave the country, the database administrators of the auction house implement new constraints. At the global level $\mathcal{G}$, Polypheny-DB uses `split` and `merge` operations to create new entities which separate sensitive from insensitive data. In general, the propagation of the schema changes follow an incremental approach; however, this approach needs to be switched to eager propagation shortly before the laws come into force.

4.2 Local Schema Changes

Data Protection (continued): Consider again the data protection scenario from Sect. 4.1. After new entities have been created at level $\mathcal{G}$ to separate sensitive from insensitive data (only for the country affected by new legal constraints; for all other countries, the schema is unaltered), the insensitive part can still stay at the $\mathcal{L}_{cheap}$ instance while the sensitive part needs to be migrated to a local instance $\mathcal{L}_{local}$. In analogy to level $\mathcal{G}$ propagation, migration takes place incrementally, with a transition to eager shortly before the laws come into force.

```
MOVE     U.address, U.birthdate, U.credit_rating
TO       SD                                          -- Sensitive data
  WHERE  SD.uid = U.id  AND  U.country = 'CH'
```

Replication: In an attempt to minimize access latency, auctions are stored on local servers, close to the location of the seller and the (majority) of the buyers. According to a new legal policy of the Swiss branch of the auction house, auctions exceeding a certain threshold need to be stored redundantly on an instance in Switzerland. Therefore, a materialized view will be created and deployed at $\mathcal{L}_{CH}$. This materialized view replicates auction data subject to the new regulations and leads to an update of the entire data distribution scheme. In order to implement the policy, all auction-related data contained in this materialized view has to be migrated eagerly to $\mathcal{L}_{CH}$, due to legal requirements.

4.3 Physical Level Schema Changes

Specialized Data Store: The local German Polypheny-DB instance $\mathcal{L}_{DE}$ measures an increase in visual similarity searches. So far, the similarity features were stored in the relation `IMG` in a row store $\mathcal{P}_{RS}$. To speedup the similarity searches, the Polypheny-DB instance decides to deploy a new store $\mathcal{P}_{sim}$ optimized for queries addressing visual similarity. The migration procedure, first, separates the similarity features from the $\mathcal{P}_{RS}$ entity using the `split` operation. After the creation of the schema in $\mathcal{P}_{sim}$, the schema dealing with the similarity features is moved to $\mathcal{P}_{sim}$. Finally, the data is moved on a proactive approach. The `feature` relation `F` is created by using the `split` operation (note that `F` will be moved to $\mathcal{P}_{sim}$ while `IMG` remains on $\mathcal{P}_{RS}$; further, the logical schema on $\mathcal{L}_{DE}$ is unchanged):

```
SPLIT  IMG
INTO   IMG.filename, IMG.type, IMG.size, IMG.auction -- Image data
AND    F.filename, F.features                        -- Feature data
```

Self-optimization: The local polystore instance at site $\mathcal{L}$ detects a change in the local workload. As a consequence of the changed interest of users, attributes are jointly accessed for which the physical schemas are not (yet) prepared since these attributes are assigned to different stores (different physical schemas). Hence, in order to optimize the physical schemas at $\mathcal{L}$, attributes are `moved` from one store $\mathcal{P}_1$ to another store $\mathcal{P}_2$. This schema change, together with the corresponding data migration, is done in a lazy approach. In case the entire workload further increases significantly, then even a new physical store $\mathcal{P}_{new}$ can be deployed at $\mathcal{L}$ and data be copied incrementally.

5 Conclusions and Outlook

Polystores allow to address heterogeneous and dynamic application workloads by jointly considering several different data stores. Data can then be provided in different models and systems, and when changes to the workload are detected, data can be migrated across stores. In this paper, we have introduced PolyMigrate, a work-in-progress approach that considers various options to apply and propagate schema changes and data migration in the distributed polystore Polypheny-DB. We plan to thoroughly evaluate and compare these alternatives based on the auction house benchmark [20] tailored to polystore databases.

Acknowledgment. This work has been partly funded by the Swiss National Science Foundation (project *Polypheny-DB: Cost- and Workload-aware Adaptive Data Management*, no. 200021_172763) and the German Research Foundation (project *NoSQL Schema Evolution and Big Data Migration at Scale*, no. 385808805).

References

1. Ataei, P., Termehchy, A., Walkingshaw, E.: Variational databases. In: Proceedings of the 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, 1 September (2017). https://doi.org/10.1145/3122831.3122839
2. Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H.: Schema validation and evolution for graph databases. In: Laender, A.H.F., Pernici, B., Lim, E.-P., de Oliveira, J.P.M. (eds.) ER 2019. LNCS, vol. 11788, pp. 448–456. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33223-5_37
3. Duggan, J., et al.: The BigDAWG polystore system. ACM SIGMOD Rec. **44**(2), 11–16 (2015). https://doi.org/10.1145/2814710.2814713
4. Guerrini, G., Mesiti, M., Sorrenti, M.A.: XML schema evolution: incremental validation and efficient document adaptation. In: Barbosa, D., Bonifati, A., Bellahsène, Z., Hunt, E., Unland, R. (eds.) XSym 2007. LNCS, vol. 4704, pp. 92–106. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75288-2_8
5. Herrmann, K., Voigt, H., Behrend, A., Rausch, J., Lehner, W.: Living in parallel realities: co-existing schema versions with a bidirectional database evolution language. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, 14–19 May 2017, pp. 1101–1116. ACM (2017). https://doi.org/10.1145/3035918.3064046
6. Hillenbrand, A., Levchenko, M., Störl, U., Scherzinger, S., Klettke, M.: MigCast: putting a price tag on data model evolution in NoSQL data stores. In: Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019), pp. 1925–1928. ACM, Amsterdam (2019). https://doi.org/10.1145/3299869.3320223
7. Holubová, I., Klettke, M., Störl, U.: Evolution management of multi-model data. In: Gadepally, V., et al. (eds.) DMAH/Poly -2019. LNCS, vol. 11721, pp. 139–153. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33752-0_10
8. Lu, J., Holubová, I.: Multi-model databases: a new journey to handle the variety of data. ACM Comput. Surv. **52**(3), 55:1–55:38 (2019). https://doi.org/10.1145/3323214
9. Necaský, M., Klímek, J., Malý, J., Mlýnková, I.: Evolution and change management of XML-based systems. J. Syst. Softw. **85**(3), 683–707 (2012). https://doi.org/10.1016/j.jss.2011.09.038
10. Podkorytov, M., Soderman, D., Gubanov, M.: Hybrid.poly: an interactive large-scale in-memory analytical polystore. In: 2017 IEEE International Conference on Data Mining Workshops (ICDMW), pp. 43–50. IEEE (2017). https://doi.org/10.1109/ICDMW.2017.13, http://ieeexplore.ieee.org/document/8215643/
11. Sadalage, P., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, Boston (2012). 0321826620
12. Saur, K., Dumitras, T., Hicks, M.W.: Evolving NoSQL databases without downtime. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME 2016), pp. 166–176. IEEE Computer Society, Raleigh (2016). https://doi.org/10.1109/ICSME.2016.47
13. Schaarschmidt, M., Gessert, F., Ritter, N.: Towards automated polyglot persistence. In: Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Hamburg, Germany, 4–6 March 2015. LNI, vol. P-241, pp. 73–82. GI (2015). http://subs.emis.de/LNI/Proceedings/Proceedings241/article46.html

14. Scherzinger, S., Klettke, M., Störl, U.: Managing schema evolution in NoSQL data stores. In: Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), Riva del Garda, Trento, Italy. (2013). http://arxiv.org/abs/1308.0514
15. Spoth, W., et al.: Adaptive schema databases. In: 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, 8–11 January 2017 (2017)
16. Stonebraker, M.: My top ten fears about the DBMS field. In: Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE 2018), pp. 24–28. IEEE Computer Society, Paris (2018). https://doi.org/10.1109/ICDE.2018.00012
17. Stonebraker, M., Çetintemel, U.: "One size fits all": an idea whose time has come and gone. In: Brodie, M.L. (ed.) Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker, pp. 441–462. ACM/Morgan & Claypool (2019). https://doi.org/10.1145/3226595.3226636
18. Tan, R., Chirkova, R., Gadepally, V., Mattson, T.: Enabling query processing across heterogeneous data models: a survey. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 3211–3220. IEEE, Boston (2017). https://doi.org/10.1109/BigData.2017.8258302
19. Vogt, M., Stiemer, A., Schuldt, H.: Icarus: towards a multistore database system. In: Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), pp. 2490–2499. IEEE, Boston (2017). https://doi.org/10.1109/BigData.2017.8258207
20. Vogt, M., Stiemer, A., Schuldt, H.: Polypheny-DB: towards a distributed and self-adaptive polystore. In: Proceedings of the IEEE International Conference on Big Data (Big Data 2018), pp. 3364–3373. IEEE, Seattle (2018). https://doi.org/10.1109/BigData.2018.8622353