Darwin: A Data Platform for NoSQL Schema Evolution Management and Data Migration

Uta Störl[1], Meike Klettke[2]

[1]University of Hagen, Germany
[2]University of Rostock, Germany

Abstract

During the development of NoSQL-backed software, the database schema evolves alongside the application code. Especially in agile development, new application releases are frequently deployed. This leads to heterogeneous data in the database and thus to new challenges for application development. To handle such heterogeneous data, we have developed various algorithms, implemented and evaluated them in a data platform for schema evolution management and data migration called *Darwin*. We provide an overview of *Darwin*, the concepts, algorithms, their implementations and the possible usage of *Darwin* in this paper.

Keywords

NoSQL databases, schema evolution, data migration

1. Introduction

Schema evolution management is one of the most challenging problems in data management today [1]. The popularity of NoSQL databases makes this issue even more complex. Schema-flexible NoSQL databases are especially popular backends in agile development. New software releases can be deployed without migration-related application downtime. An empirical study of NoSQL database schema development shows that more schema-relevant changes are included with the use of NoSQL databases in comparison to relational databases [2]. In addition, schemas become more complex over time and take longer to stabilize.

Managing schema evolution involves two main tasks: discovering (extracting) structural changes to data and dealing with these changes from an application development perspective (data migration). In the past, we have published several papers on specific research results and theoretical achievements of schema evolution management [3, 4] and data migration [5, 6]. We also presented demo papers of implementations of some sub-aspects [7, 8]. This paper is intended to provide an overall view of the *Darwin* system and shows the interaction of the different algorithms.

Darwin supports the whole schema evolution management and data migration lifecycle. The system is implemented for different types of NoSQL database systems. This paper also presents specific architectural and implementation aspects of this data platform. Furthermore, we have now published *Darwin* publicly[1] in a fully operational docker container so that the system can also be used by interested researchers.

The rest of the paper is organized as follows: In Section 2 we discuss the related work. Section 3 gives an overview of the *Darwin* data platform. In Section 4 we discuss the main functionalities of *Darwin* and their interplay. Afterwards extensions of *Darwin* are presented in Section 5. We conclude with a summary and an outlook on further work.

2. Related Work

The implementation of *Darwin* bases on several research results and theoretical achievements, partly developed by our own group.

Schema Extraction There are several suggestions for schema extraction for NoSQL databases [9, 10, 11]. In [3], we developed an approach to schema extraction which generates a graph structure representing all structural variants of a given dataset. In the next step, this internal graph structure is summarized in a JSON schema description. Meanwhile, this basic functionality of static schema extraction for NoSQL databases is available in some commercial tools like Hackolade[2], Studio 3T[3] or research prototypes like Josch [12].

[1]https://github.com/dbishagen/darwin
[2]https://hackolade.com/
[3]https://studio3t.com/

Published in the Workshop Proceedings of the EDBT/ICDT 2022 Joint Conference (March 29-April 1, 2022), Edinburgh, UK
uta.stoerl@fernuni-hagen.de (U. Störl);
meike.klettke@uni-rostock.de (M. Klettke)
0000-0003-2771-142X (U. Störl); 0000-0003-0551-8389 (M. Klettke)

Version History Extraction The lack of all NoSQL schema extraction approaches (cf. [13] for a survey) still is that they do not consider and cannot detect schema changes over time. This observation leads us to the development of a schema version history extraction approach. This algorithm can be applied if a partial order of the datasets is available (e.g. a timestamp or a creation-date). It in turn extracts an internal graph structure and adds timestamp information. Each structural change that the algorithm detects triggers the generation of a new schema version. In addition, the change operations are extracted from the differences of two consecutive structural versions and are represented as evolution operations [4]. Thus this algorithm is able to uncover the complete evolution history and the genesis of available databases. We have presented a demonstration of this function, which is essential for schema evolution management, in [7].

Query Rewriting To read NoSQL datasets in different versions, query rewriting is necessary. Query rewriting is a core database technology which has been introduced to optimize query execution by using materialized views [14]. Query rewriting can also be used to handle irregular structures. A query rewriting approach which considers the data heterogeneity and generates different subqueries for different varieties is developed in [15]. In [16], we suggest how query rewriting can use schema evolution operations to unify different consecutive structural versions in queries.

Data Migration While there are some approaches in the area of managing schema evolution and data migration for relational systems [17, 18], there is very little in the field of NoSQL databases. We proposed the concepts of eager and lazy migration in NoSQL databases in [19]. KVolve, an extension for the Redis NoSQL database that supports lazy migration, was introduced in [20]. The IDE integrated tool ControVol supports eager and lazy migration based on static type checks of object mapper class declarations as recorded by the code repository [21].

We presented initial ideas of hybrid data migration strategies (incremental and predictive migration) in [5] and described them in detail in [6]. We intensively studied the impact of different data migration strategies on migration cost and latency and presented and discussed the results in [22].

3. System Architecture

In this section, we will introduce the system architecture of *Darwin* and the interaction of the individual modules. The entire *Darwin* system is implemented in Java. Figure 1 shows the system architecture of *Darwin*. In the agile application development use case, *Darwin* is a middleware between a Java application and a database storing variational data:

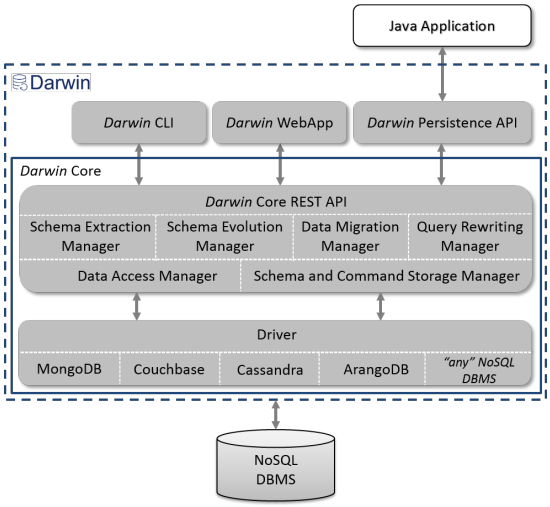

Figure 1: Darwin System Architecture

- At the top of the application stack is the Java application. It stores its data in a NoSQL database, interacting with the system-independent *Darwin Persistence API (DPA)*.
- Via the *Darwin WebApp* or the *Darwin CLI*, application developers may trigger schema evolution management and data migration tasks directly. We will explain the these tasks in detail in Section 4.
- All user interfaces use the *Darwin Core REST API*. This architecture allows the flexible use of *Darwin*, as we will see when we present the extensions in Section 5.
- The *Darwin Core REST API* interfaces with the *core* modules necessary for the schema evolution management and data migration lifecycle, which we will present in detail in Section 4. These modules are implemented independent of a concrete database system.
- A *Data Access Manager* and a *Schema and Command Storage Manager* were implemented as a uniform interface for the interaction of the *core* modules with the respective database systems. The *Schema and Command Storage Manager* stores the schema versions and the schema evolution operations. This information can either be stored in the same database as the data or in a separate database.
- The *Drivers* are responsible for the connection to the specific database system. Since the languages

of all NoSQL DBMS differ, the mapping to the respective system is done in these modules.

Currently *Darwin* supports the most popular document stores MongoDB and Couchbase, the wide column store Cassandra and the multi-model database system ArangoDB. The architecture is designed for easy extensibility. Adding a new DBMS requires only the implementation of the appropriate driver.

4. Main Functionalities

Darwin supports the whole schema evolution management and data migration lifecycle. In the following we will explain this lifecycle and the corresponding functionalities of *Darwin* and their interaction.

4.1. Schema and Version History Extraction

Schema extraction and version history extraction belong to the data preprocessing steps which are implemented in the *Darwin* tool. Both steps are necessary for the analysis of available NoSQL datasets (which have been created outside of *Darwin*) and for understanding the implicit structures and their changes over time. The algorithms are detecting the variabilities in the NoSQL data, structural outliers and the different versions over time.

Figure 2 shows an example of a version history extraction performed in *Darwin*. The screenshot shows two versions side by side in JSON schema notation. The schema evolution operations are stated above. Changes w.r.t. the previous schema version are highlighted [7].

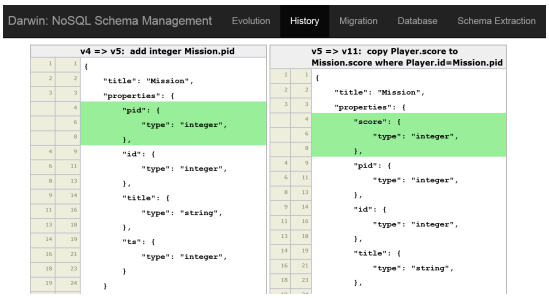

Figure 2: Example of a Schema Version History Extraction

The schema and version history extraction algorithm reads all datasets of the NoSQL database. The implemented algorithm can run incrementally which means in case of new datasets that only new data are analyzed and the results merged.

To the best of our knowledge, *Darwin* is the only schema management tool supporting the extraction of the schema version history.

Notes on Performance and Scalability Extracting and analyzing the entire data instance is a one-time effort. After the initial schema and version history extraction, newly added entities can be analyzed incrementally and on-the-fly. Since *Darwin* does not load the entire data instance into the main memory, but only incremental batches [4], *Darwin* can safely handle large volumes of data.

4.2. Schema Evolution Management

Schema evolution is an ongoing process during the development of an application. Schema evolution operations (SMOs) can be

- manually entered in *Darwin* using the *Darwin CLI* or *Darwin WebApp*, or
- automatically observed by incremental version history extraction.

Which datasets being migrated after a schema evolution operation depends on the chosen data migration strategy. *Darwin* has implemented eager, lazy, and various hybrid data migration strategies. We will present these strategies in Section 4.4.

4.3. Query Rewriting

In NoSQL databases, storage of different schema versions can be done in the same database. Schema evolution in combination with lazy or hybrid data migration has to face the situation that datasets stored in the same NoSQL database have different structural versions. In this scenario, the evolution operations that can transform datasets from the structural version n to the successor version $n + 1$ are stored in the *Schema and Command Storage Manager* (see Figure 1).

If such versioned datasets are to be accessed and queried by an application, *query rewriting* is necessary to distribute the query to the different structural versions. *Forward query rewriting* is applied if a query which assumes the structural version n is translated into the structural version $n + i$. In this case the *list of evolution operations* (for translation from version n to $n+1$, $n+2$, $\ldots$, $n + i$) is applied onto the query. *Backward query rewriting* is used to access preceding structural versions. To achieve this *reverse evolution operations* are used for the translation of the query. In both cases, query rewriting generates different subqueries (one for each schema version), executes the subqueries and unions the results.

Query rewriting enables a transparent access to datasets in different schema versions and is the prerequisite for a lazy and hybrid data migration.

4.4. Data Migration

Migration Strategies In *Darwin*, structural changes can be defined by schema evolution operations (SMOs). In the system both single-type operations (add, rename, delete) and multi-type operations (move and copy) [19] are supported. The evolution operations define the changes of the schema. For each evolution operation, a corresponding data migration operation is generated that executes the same structural changes in the datasets. *Darwin* implements several different data migration strategies:

Eager Migration An *eager data migration* migrates all datasets immediately after the introduction of a new schema version. Eager data migration has the advantage that all datasets always reflect the latest structural version. This reduces the latency when datasets are accessed. A disadvantage is that migration costs are high since in all cases all datasets are updated even those not in use. Migration costs are especially concerning when the database is hosted in the cloud.

Lazy Migration In order to avoid unnecessary migration processes and thus reduce migration costs, *Darwin* provides another migration strategy – the so-called *lazy migration*. The basic idea is that after the introduction of a new schema version, no dataset will be updated. The new schema version and the corresponding schema evolution operation are stored in the *Schema and Command Storage Manager* (see Figure 1). All datasets are kept in their original version. As a result, lazy migration can lead to NoSQL databases containing datasets in different structural versions.

In case that datasets are accessed by a query, the query is rewritten onto the different versions (see Section 4.3). The resulting datasets are migrated at runtime and stored in the database in the new version. In the case of a single-type operation, runtime migration is relatively simple and efficient. In the case of a multi-type operation, the migration is much more complex. It is possible that during a copy or move operation the corresponding objects are not yet in the latest version and must be migrated as well. This in turn can require that further objects need to be migrated (cascading migration). Currently, we limit the depth of cascading migrations to two levels in *Darwin*. An analysis of the best fitting strategies for different cases could be the subject of further research.

A lazy approach has the advantage that only those datasets currently in use are being migrated. Cold data are not accessed and subsequently not migrated. This strategy automatically minimizes data migration costs. The disadvantage of the lazy approach though is that data migration takes place during runtime and can negatively impact data access latency.

Hybrid Data Migration Strategies Beside the two basic data migration strategies that either migrate all datasets immediately after the introduction of a new schema version (eager) or no dataset (lazy) at all, *Darwin* also offers several hybrid migration strategies that provide an intelligent control of the data migration. The hybrid migration strategies optimize the two different targets: low total migration costs and low latency at runtime when a dataset is accessed.

Incremental Migration A simple hybrid strategy is the incremental migration. The data is only migrated completely at certain points in time (for example, after a certain number of schema evolution operations have been executed). Between two incremental migrations, the data is migrated lazily. This approach has lower migration costs than eager migration. All datasets are, however, updated even those not in use.

Predictive Migration A more sophisticated approach is the predictive migration. The so-called hot data, i.e. the data that is frequently accessed, should be kept up-to-date. The prediction of the hot data is implemented in *Darwin* by keeping track of past data accesses while ordering the accessed entities accordingly by means of exponential smoothing. We use a prediction set whose size is configurable. Data in this prediction set is migrated proactively after each schema change. Data not contained in the prediction set is migrated when it is accessed lazily. This reduces both runtime overhead and migration costs.

The size of the prediction set is configurable within *Darwin*. In [6] we presented initial approaches to adjust the size of this prediction set self-adaptive depending on given bounds on migration cost and latency.

Selection of the appropriate Data Migration Strategy We have extensively studied the impact of different data migration strategies on migration cost and latency. Probabilistic Monte Carlo method of repeated sampling were used for the analysis. Figure 3 shows an example. The impact of these different migration strategies on migration costs (assuming a cloud hosted database) and the data access latency is obvious. We presented and discussed the results in detail in [22]. Nevertheless, selecting the appropriate data migration strategy is a significant challenge. We have developed the data migration advisor *MigCast* for this purpose, which we present in Section 5.1.

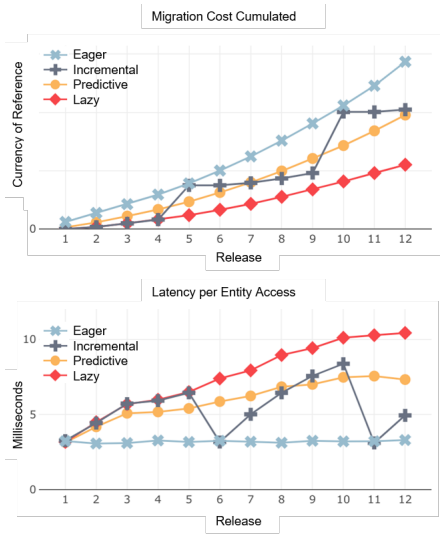

Figure 3: Impact of different Migration Strategies (cf. [22])

4.5. Migration Optimization

There are different opportunities to optimize the execution of the migration operations. We would like to briefly outline three aspects:

Composition of Schema Evolution Operations When a legacy dataset needs to be migrated from several versions back, the pending schema changes may either be applied stepwise or by composite migration. As a simple example, an add and a rename operation can be combined into one add operation on the same data object. In [5] we introduced the composition rules for schema evolution operations (both single-type and multi-type). Measurement results and aspects of the implementation were presented in [23].

Caching An obvious optimization is the extensive use of caching. *Darwin* contains a *Schema Cache* and a *Command Cache* to avoid repeated reading of this information during data migration. Furthermore, a *Composer Cache* was introduced for the described composition of migration operations, the effects of which were mentioned above and discussed in detail in [23].

Location Simple single-type migration operations like add and delete can be executed native directly in the NoSQL DBMS. For more complex multi-type operations like copy and move this is not always possible. This depends on the offered functionality of the NoSQL DBMS.

For example, many NoSQL DBMS do not support joins. In this case, copy and move operations have to be orchestrated within *Darwin* instead using database-provided joins within the *Drivers* (cf. Figure 1). At the beginning of the implementation of *Darwin* in 2014, MongoDB was one of those NoSQL database systems which did not support joins. In MongoDB, this functionality is available since version 3.2.

Performing migration operations within *Darwin* offers the opportunity to support NoSQL DBMS that do not natively support all migration operations. To evaluate this aspect, we have used the *EvoBench* benchmark which we have developed. Figure 5 shows the results of the evaluation which will be explained in Section 5.2.

5. Darwin Ecosystem

In addition to the core functionality of *Darwin* presented in Section 4, two other important aspects were investigated and corresponding tools were developed. In Section 5.1 we introduce the data migration advisor *MigCast*. Then, in Section 5.2 the schema evolution benchmark *EvoBench* is presented.

5.1. MigCast

As explained in Section 4.4, selecting the appropriate migration strategy is a huge challenge. We have developed the data migration advisor *MigCast* for this purpose.

MigCast is implemented on top of *Darwin*. As input parameters *MigCast* takes into account the characteristics of the data instance and the data access pattern, e.g., a Pareto distribution of future reads and writes, the data model changes (schema evolution), and particulars about the cloud pricing model. With these inputs, *MigCast* predicts the migration costs and the data access latency. This estimation is based on three core modules: a *Workload Simulator*, a *Cost Calculator*, and a *Latency Profiler* (see Figure 4).

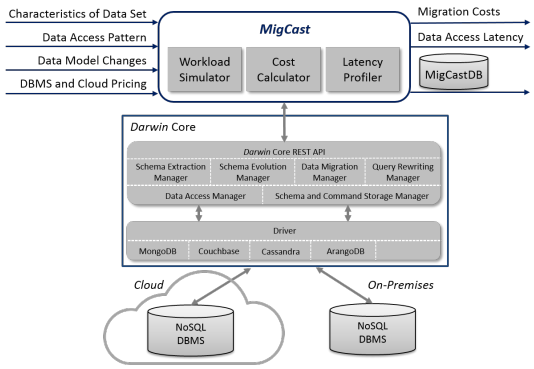

Figure 4: MigCast Architecture

In Figure 3 in Section 4.4 we have shown an example of such an estimation performed by *MigCast*. To support the *reproducibility* of the experiments, the configuration of the performed measurements and the results are stored in a separate database (*MigCastDB* in Figure 4). *MigCast* is publicly available as part of the *Darwin* distribution[4].

5.2. EvoBench

Darwin belongs to the first systems tailored for an ongoing evolution of database backends. For testing and evaluating *Darwin* and other approaches for NoSQL schema evolution and data migration, we defined and implemented the benchmark *EvoBench*. *EvoBench* is the first available benchmark to validate the abilities of a system to evolve NoSQL databases and to determine and compare the performance of the dedicated evolution operations [24, 25]. *EvoBench* bases on a Customer-Product-Order-Invoice dataset originally introduced in [26] and defines 20 schema evolutions on this application, ranging from simple extensions of the data model up to more complex refactoring.

The *EvoBench* tool is implemented in Python. *EvoBench* treats the respective schema evolution platform as a *black box* and uses the provided API for the schema evolution operations. In the case of *Darwin*, we use the *Darwin Core Rest API* (see Figure 1). In addition to the data model and schema evolution operations predefined in the benchmark, the *EvoBench* tool also supports the use of your own data models and schema evolution operations for experiments.

As an example, we return to the impacts discussed in Section 4.5 when performing migration operations natively in the database or in *Darwin*. Figure 5 shows the execution time as well as the migrations costs (in terms of executed operations) executing these operations on 123,200 data objects [25] using MongoDB.

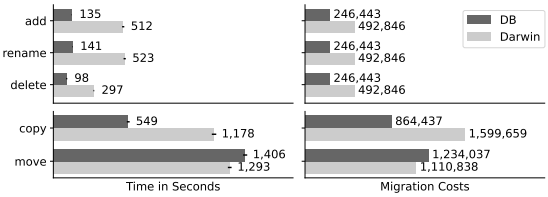

Figure 5: Impact of different Locations of Migration Operation Execution (cf. [25])

As explained earlier, *EvoBench* is designed to be independent from *Darwin* and can also be used to evaluate other schema evolution management platforms. We have deployed *EvoBench* and associated measurements in fully operational docker containers[5].

[4] https://github.com/dbishagen/darwin
[5] https://doi.org/10.5281/zenodo.4993636

6. Conclusion and Outlook

In this article, we have introduced the main algorithms provided by *Darwin* – the tool for a continuous evolution of NoSQL backends. *Darwin* can be applied to databases that are starting from scratch as well as to already existing NoSQL databases even those containing different versions of legacy data in the same database. In all cases, the schema evolution and data migration of *Darwin* component keeps dataset structures up-to-date.

The tool can be applied to different NoSQL databases (e.g. MongoDB, Couchbase, Cassandra, and ArangoDB). A side effect is that *Darwin* can also be used for the migration of data between *different* database systems, e.g. from MongoDB into ArangoDB and thus enables interoperability between different NoSQL backends.

In the data migration component of *Darwin* different optimization aims (migration costs, latency) can be pursued. In Section 4.4, we have introduced different data migration strategies and have shown their impact on the different cost metrics. One task of future work is to develop a self-adaptive data migration which recommends a data migration strategy and optimizes parameter setting in the dedicated algorithm [6].

Another direction of future development in *Darwin* is the development of a polystore data migration method including schema optimization [27].

With *Darwin* we offer a complete solution that is required for every long-running NoSQL database to keep the structures permanently up-to-date and to ensure that NoSQL data is operational over long periods of time.

Acknowledgments

This work has been funded by Deutsche Forschungsgemeinschaft (German Research Foundation) – 385808805. We would like to thank all project members whose work contributed to the success of the project, especially André Conrad, Andrea Hillenbrand, Mark Lukas Möller and Stefanie Scherzinger. Special thanks go to all students of Darmstadt University of Applied Sciences who have contributed to the implementation of *Darwin*.

References

[1] M. Stonebraker, My Top Ten Fears about the DBMS Field, in: Proc. ICDE, IEEE, 2018, pp. 24–28. doi:10.1109/ICDE.2018.00012.

[2] S. Scherzinger, S. Sidortschuck, An Empirical Study on the Design and Evolution of NoSQL Database Schemas, in: Proc. ER, Springer, 2020, pp. 441–455. doi:10.1007/978-3-030-62522-1_33.

[3] M. Klettke, U. Störl, S. Scherzinger, Schema Extraction and Structural Outlier Detection for

JSON-based NoSQL Data Stores, in: Proc. BTW, GI, 2015, pp. 425–444. URL: https://dl.gi.de/20.500.12116/2420.

[4] M. Klettke, H. Awolin, U. Störl, D. Müller, S. Scherzinger, Uncovering the Evolution History of Data Lakes, in: Proc. IEEE Big Data, IEEE, 2017, pp. 2462–2471. doi:10.1109/BigData.2017.8258204.

[5] M. Klettke, U. Störl, M. Shenavai, S. Scherzinger, NoSQL schema evolution and big data migration at scale, in: Proc. IEEE Big Data, IEEE, 2016, pp. 2764–2774. doi:10.1109/BigData.2016.7840924.

[6] A. Hillenbrand, U. Störl, S. Nabiyev, M. Klettke, Self-adapting data migration in the context of schema evolution in NoSQL databases, Distributed and Parallel Databases abs/2104.14828 (2021) 1–21. doi:10.1007/s10619-021-07334-1.

[7] U. Störl, D. Müller, A. Tekleab, S. Tolale, J. Stenzel, M. Klettke, S. Scherzinger, Curating Variational Data in Application Development, in: Proc. ICDE, IEEE, 2018, pp. 1605–1608. doi:10.1109/ICDE.2018.00187.

[8] A. Hillenbrand, M. Levchenko, U. Störl, S. Scherzinger, M. Klettke, MigCast: Putting a Price Tag on Data Model Evolution in NoSQL Data Stores, in: Proc. SIGMOD, ACM, 2019, pp. 1925–1928. doi:10.1145/3299869.3320223.

[9] D. S. Ruiz, S. F. Morales, J. G. Molina, Inferring Versioned Schemas from NoSQL Databases and Its Applications, in: Proc. ER, Springer, 2015, pp. 467–480. doi:10.1007/978-3-319-25264-3_35.

[10] L. Meurice, A. Cleve, Supporting schema evolution in schema-less NoSQL data stores, in: Proc. IEEE SANER, IEEE, 2017, pp. 457–461. doi:10.1109/SANER.2017.7884653.

[11] M. A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, Parametric schema inference for massive JSON datasets, VLDB J. 28 (2019) 497–521. doi:10.1007/s00778-018-0532-7.

[12] M. Fruth, K. Dauberschmidt, S. Scherzinger, Josch: Managing Schemas for NoSQL Document Stores, in: Proc. ICDE, IEEE, 2021, pp. 2693–2696. doi:10.1109/ICDE51399.2021.00306.

[13] P. Contos, M. Svoboda, JSON Schema Inference Approaches, in: Proc. ER Workshops, Springer, 2020, pp. 173–183. doi:10.1007/978-3-030-65847-2_16.

[14] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava, Answering Queries Using Views, in: Proc. PODS, ACM Press, 1995, pp. 95–104. doi:10.1145/212433.220198.

[15] Y. Papakonstantinou, Polystore Query Rewriting: The Challenges of Variety, in: EDBT/ICDT Workshops, CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1558/paper46.pdf.

[16] M. L. Möller, M. Klettke, A. Hillenbrand, U. Störl, Query Rewriting for Continuously Evolving NoSQL Databases, in: Proc. ER, Springer, 2019, pp. 213–221. doi:10.1007/978-3-030-33223-5_18.

[17] C. Curino, H. J. Moon, C. Zaniolo, Graceful database schema evolution: the PRISM workbench, Proc. VLDB Endow. 1 (2008). doi:10.14778/1453856.1453939.

[18] S. Bhattacherjee, G. Liao, M. Hicks, D. J. Abadi, BullFrog: Online Schema Evolution via Lazy Evaluation, in: Proc. SIGMOD, ACM, 2021, pp. 194–206. doi:10.1145/3448016.3452842.

[19] S. Scherzinger, M. Klettke, U. Störl, Managing Schema Evolution in NoSQL Data Stores, in: Proc. DBPL@VLDB, 2013. URL: http://arxiv.org/abs/1308.0514.

[20] K. Saur, T. Dumitras, M. W. Hicks, Evolving NoSQL Databases without Downtime, in: 2016 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2016, pp. 166–176. doi:10.1109/ICSME.2016.47.

[21] S. Scherzinger, T. Cerqueus, E. C. de Almeida, ControVol: A framework for controlled schema evolution in NoSQL application development, in: Proc. ICDE, IEEE, 2015, pp. 1464–1467. doi:10.1109/ICDE.2015.7113402.

[22] A. Hillenbrand, S. Scherzinger, U. Störl, Remaining in Control of the Impact of Schema Evolution in NoSQL Databases, in: Proc. ER, Springer, 2021, pp. 149–159. doi:10.1007/978-3-030-89022-3_13.

[23] U. Störl, A. Tekleab, M. Klettke, S. Scherzinger, In for a Surprise When Migrating NoSQL Data, in: Proc. ICDE, IEEE, 2018, p. 1662. doi:10.1109/ICDE.2018.00202.

[24] M. L. Möller, M. Klettke, U. Störl, EvoBench — A Framework for Benchmarking Schema Evolution in NoSQL, in: Proc. IEEE Big Data, IEEE, 2020, pp. 1974–1984. doi:10.1109/BigData50022.2020.9378278.

[25] A. Conrad, M. L. Möller, T. Kreiter, J.-C. Mair, M. Klettke, U. Störl, EvoBench: Benchmarking Schema Evolution in NoSQL, in: Proc. TPCTC@VLDB, Springer, 2021, pp. 33–49. doi:10.1007/978-3-030-94437-7_3.

[26] C. Zhang, J. Lu, P. Xu, Y. Chen, UniBench: A Benchmark for Multi-model Database Management Systems, in: Proc. TPCTC@VLDB, Springer, 2018, pp. 7–23. doi:10.1007/978-3-030-11404-6_2.

[27] A. Conrad, S. Gärtner, U. Störl, Towards Automated Schema Optimization, in: Proc. ER Demos and Posters, CEUR-WS.org, 2021, pp. 37–42. URL: http://ceur-ws.org/Vol-2958/paper7.pdf.