NoSQL Schema Evolution and Data Migration: State-of-the-Art and Opportunities

Uta Störl
Darmstadt Univ. of Applied Sciences
Germany
uta.stoerl@h-da.de

Meike Klettke
University of Rostock
Germany
meike.klettke@uni-rostock.de

Stefanie Scherzinger
OTH Regensburg
Germany
stefanie.scherzinger@oth-r.de

ABSTRACT

NoSQL database systems are very popular in agile software development. Naturally, agile deployment goes hand-in-hand with database schema evolution. The main aim of this tutorial is to present to the audience the current state-of-the-art in continuous NoSQL schema evolution and data migration: (1) We present case studies on schema evolution in NoSQL databases; (2) we survey existing approaches to schema management and schema inference, as implemented in popular NoSQL database products, and also as proposed in academic research; (3) we present approaches for extracting schema versions; (4) we analyze different methods for efficient NoSQL data migration; and (5) we give an outlook on further research opportunities.

Duration: 1.5 hours

1 INTRODUCTION

Recent position papers demand more schema flexibility, such as the ability to handle variational data [3, 42]. Many agile software developers have long since turned towards NoSQL database systems such as *MongoDB*[1], *Couchbase*[2], or *ArangoDB*[3] which are schema-flexible, or even altogether schema-free. They allow to store datasets in different structural versions to co-exist.

Yet even when the database management system does not maintain an explicit schema, there is commonly an implicit schema, as the application code makes assumptions about the structure of the stored data. For instance, in Figure 1, the Java code in lines 1 through 9 implies a schema: An entity for person `Jo Bloggs` is created, and then persisted in the `people` collection.

```
1  List<Integer> books = Arrays.asList(27464, 747854);
2  DBObject person = new BasicDBObject("_id", "jo")
3                    .append("name", "Jo Bloggs")
4                    .append("address",
5                      new BasicDBObject("street", "123 Fake St")
6                        .append("city", "Faketon")
7                        .append("state", "MA")
8                        .append("zip", 12345))
9                    .append("books", books);
10 DBCollection collection = database.getCollection("people");
11 collection.insert(person);
```

Figure 1: Storing a person entity in MongoDB, using Java.[4]

[1]https://docs.mongodb.com/
[2]https://www.couchbase.com/products/server
[3]https://www.arangodb.com/
[4]From "Getting Started with MongoDB and Java", https://www.mongodb.com/blog/post/getting-started-with-mongodb-and-java-part-i, published August 2014.

With each new release of the software, the application code evolves. Eventually, so will the implicit schema declarations. Then, data stored in the production system will have to be migrated accordingly. Yet writing custom migration scripts — which seems to be the common practice today — is error-prone and expensive. Thus, there is a dire need for well-principled tool support for long-term maintenance of NoSQL database schemas.

With the schema declared within the application layer, the burden of schema maintenance is shifted into the domain of the application developers. Accordingly, we observe various grassroots efforts from the developer community to tackle schema evolution. Unfortunately, these solutions do not build upon the existing state-of-the-art in research. Overall, we see it as an opportunity for the database research community to contribute well-founded and practical solutions to real-world problems.

In this tutorial, we give an overview of schema management in agile development with NoSQL database systems. The authors proposing this tutorial have been publishing in this domain for over 6 years. We present the current state-of-the-art in research, as well as in practice. We cover inferring schema-on-read with outlier detection, deriving schema versions, the corresponding schema evolution operations matching between them, as well as the resulting data migration operations. Different migration strategies and their impact, such as the overall migration costs and the latency upon accessing an entity, are also discussed.

A strong point of this tutorial is that we motivate the problem domain by presenting an empirical study on NoSQL schema evolution in real-world applications. Moreover, we demo existing tools for NoSQL schema management, e.g., for schema design and schema extraction. Incorporating small, live demos, we put together a diverse and diverting tutorial.

2 OUTLINE

This 1.5-hour tutorial is split into five parts:

(1) **Case Studies (~15 min.)**. We present an empirical study on the schema imposed on NoSQL databases by applications, as well as the dynamics of NoSQL schema evolution.
(2) **NoSQL Schema Management (~20 min.)**. In this part we discuss different architectures and existing solutions for NoSQL schema management. Here, we present research approaches as well as first products.
(3) **NoSQL Evolution Management (~25 min.)**. We present solutions for NoSQL evolution management. Beside a language for declaring NoSQL schema evolution operations, we focus on approaches for extracting schema versions.
(4) **Data Migration (~20 min.)**. Based on the previous parts, we present different data migration strategies and discuss their quantitative assessment.
(5) **Future Opportunities (~10 min.)**. Finally, we outline open research problems as potential directions for further research, as well as current development in this area.

3 GOALS AND OBJECTIVES

3.1 Case Studies

For our introduction, we present an empirical study on real-world database applications, each backed by a schema-flexible NoSQL data store. We investigate whether developers denormalize their schema, as is the recommended practice in data modeling for NoSQL data stores, and also a research subject [4, 25]. Further, we analyze the entire project history, and with it, the evolution of the NoSQL schema. By looking at real-world evidence, we pinpoint characteristic problems (such as the increased frequency of schema changes). We discuss how existing solutions do not fully transfer (e.g., since they rely on the schema being declared explicitly, rather than being implicit in the code). Thus, existing solutions cannot address the needs of application developers. Finally, we state the desiderata for tackling NoSQL schema evolution, in preparation to the subsequent tutorial.

3.2 NoSQL Schema Management

NoSQL database systems differ with regard to schema support: There are *schema-free* systems without any native schema support (e.g. *Couchbase*, *CouchDB*[5], *Neo4j*[6]). Other systems offer optional schema support (e.g. *MongoDB*, *OrientDB*[7]; some of these systems support different schema modes: *schema-full* or *schema-flexible*). There are also schema-full systems, with a mandatory schema (e.g. *Cassandra*[8]). In addition, there are proposals for vendor-independent middleware that manages the NoSQL schema (e.g. the *Darwin* schema management component [45]).

3.2.1 Capturing the NoSQL Schema. In our tutorial, we present three strategies how the NoSQL schema can be captured from a schema-free or schema-flexible data store: (a) In the tradition of textbook schema design, the NoSQL schema can be derived top-down from some conceptual model. (b) The schema may be extracted posthumously, given a data instance. (c) The schema may be also derived by static analysis of the application code. We now briefly discuss these options.

a) Forward Engineering/Schema-First. In forward engineering, or schema-first approaches, the schema is explicitly defined – for example with modeling tools such as *Hackolade*[9] or *erwin*[10]: Users of these tools draw extended entity relationship models or graphical visualizations of JSON Schema, which they can compile for a given NoSQL database system. The principles behind NoSQL schema design are being explored in academic research: In [1], a model-driven approach for designing NoSQL databases has been developed. The authors of [4] propose an abstract data model for NoSQL databases, which exploits the commonalities of various NoSQL systems. Another project [8] extends the schema-first approach and generates object-oriented class hierarchies. The class declarations actually represent entity collections, using Object-NoSQL mapper libraries such as Mongoose and Morphia.

Yet in agile development, the schema is often not fixed up front. Therefore, we next discuss schema reverse engineering.

b) Reverse Engineering from Data/Schema-on-Read. For processing data without explicit schema information, *reverse engineering* can be necessary. In the following we will refer to this approach as

[5]https://couchdb.apache.org/
[6]https://neo4j.com/
[7]https://orientdb.org/
[8]http://cassandra.apache.org/
[9]https://hackolade.com/
[10]https://erwin.com/products/erwin-dm-nosql/

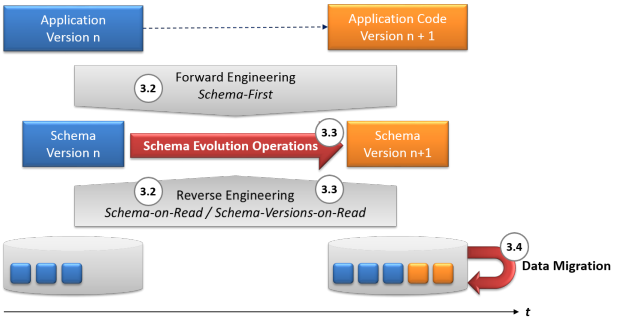

Figure 2: The Big Picture: Moving from schema version n (blue, shown to the left) to version $n+1$ (orange, shown to the right), the persisted entities (blue) may not be immediately migrated. Rather, the data store now holds entities in both schema versions (blue and orange).

schema-on-read (this term is currently used differently in various Big Data/NoSQL application areas).

The general idea has been introduced in [22], based on an earlier approach for XML schema extraction in [26]: The implicit structural information from all entities is combined into a graph, from which the schema and statistics can be derived. The schema inference approach delivers a *schema overview*. The algorithm and its optimizations will be introduced.

Similar approaches [2, 13, 20, 33, 36] also infer a schema or generate conceptual models. The authors of [43] consider the challenge of designing schemas for existing JSON datasets as an interactive problem and present a roll-up/drill-down style interface for exploring collections of JSON records. In [5], schemas are inferred from datasets by typing the input according to a type system. This approach is designed for MapReduce-based, and thus highly scalable, execution. In [48], a descriptive schema is inferred, and documents are indexed by their structural properties, so that they may even be queried accordingly.

Certain (NoSQL) database design tools, such as *Hackolade* and *erwin*, also implement schema reverse engineering, and certain NoSQL database systems (for example MongoDB) come with similar features built-in.

Schema inference from existing data is also one of the subtasks in data lake analytics [29]. In data lakes, the "load-first, schema-later" paradigm requires a combination of schema inference with the inference of integrity constraints [12, 14, 30, 31], and further descriptions of the data content, such as the semantic data type [19]. Data cleaning methods are also based on reverse engineering of structure and frequencies of occurrence [32, 38].

c) Reverse Engineering from Code. Since the application code implicitly declares a schema, this schema may be extracted. This task is straightforward when applications use Object-NoSQL mapper libraries, since class declarations then map to collections of persisted entities. This approach is followed in [34, 39]. For application code without mapper libraries, we need to resort to more involved code analysis. For instance, in programmatically extracting a schema for collection `people` from the code in Figure 1, we might employ data flow analysis, as done in [49], to detect which statements characterize a collection.

3.2.2 The Big Picture. Figure 2 gives an overview of NoSQL schema (evolution) management and data migration, which will

accompany us through the entire tutorial. The left part of the figure visualizes the approaches of forward engineering and reverse engineering discussed so far (top-down vs. bottom-up).

If we want to consider not only a static view of the schema, but also its evolution along with the application code (illustrated by application versions n and $n+1$ in the upper part, and two different versions of the database instance at two different times, in the lower part of Figure 2), new challenges arise. In our tutorial, we next discuss challenges and solutions for handling data in co-existing schema versions, especially schema evolution operations (represented by the right arrow in the center of the figure), as well as approaches for extracting not only a schema, but to also identifying different schema versions.

In the fourth part of the tutorial we then discuss various data migration strategies (symbolized by the self-referencing arrow on the bottom right of Figure 2).

3.3 NoSQL Schema Evolution Management

Handling different versions of data in a single database is becoming more and more important – in relational databases [3, 16, 42] as well as for different types of NoSQL databases [6, 40]. In this tutorial, we survey different approaches.

NoSQL Evolution Language. Most NoSQL database management systems do not provide a language capturing schema evolution operations. There are several proposals for such a language. For instance, the authors of [41] define a language originally for transformation between different NoSQL databases that can also be used for data migration within the same store [15]. In this tutorial, we present the NoSQL schema evolution language introduced in [40] for different types of NoSQL database systems, implemented in [45] and extended in [18] for multi-model data. The language contains operations that affect only one entity-type, or synonymously in MongoDB terminology, one collection (*single-type* operations are *add*, *delete*, and *rename*). For complex refactoring, operations that affect the entities of more than one entity type (*multi-type* operations *copy*, *move*, *split*, and *merge*) are available. Via these schema evolution operations (or schema modification operations by the terminology of [9]), the schema changes between consecutive versions of the application code can be declared. This mapping is sketched by the red arrow in the center of Figure 2.

Heterogeneity Classes. We capture the degree of structural heterogeneity in a data instance via the notion of heterogeneity classes, introduced in [27]. For example, persisted entities may be very homogeneous in their structure, or very heterogeneous. We outline the implications, present illustrative examples, and define the class-specific semantics of evolution operations.

Schema-Versions-on-Read. Whereas the *schema overview* presented in the second part of the tutorial delivers information on structures, data types, nesting of information, as well as information on required and optional parts, it does not capture the structural changes over time. For this reason, we present a further method, developed for inferring schema versions, as well as the changes between consecutive versions [21, 44]. Approaches for extracting *schema-versions-on-read* are based on a partial order of the data, e.g. based on timestamps. Besides schema inference, we can use the partial order to find out when and how structural information has changed and derive schema versions accordingly. Applying additional information on integrity constraints, we also can suggest evolution operations that have caused the schema changes [21]. To our knowledge, this functionality is not yet offered by any commercial products, but primarily implemented in research prototypes, e.g., [7, 35] for representing UML class model versions, and [21, 45] for generating JSON Schema versions as well as the matching evolution operations.

3.4 Data Migration

After identifying the schema evolution operations, the data can be migrated. There are various strategies, with different impacts on latency and migration effort. Traditionally, data migration is carried out *eagerly*, i.e., all data is migrated immediately when the schema changes. Since this can be expensive, especially in a cloud environment [11, 17], data can be migrated *lazily* [23, 37], so data is not migrated until it is actually accessed. This approach is preferable in databases where the share of "hot" data is relatively small compared to the total amount of data. The downside is that lazy migration adds latency to database reads and writes, since the data might still have to be migrated. Figure 3 visualizes the conflicting goals of monetary data migration costs (e.g., charged by a cloud provider) and latency overhead upon read or write access, for different migration strategies.

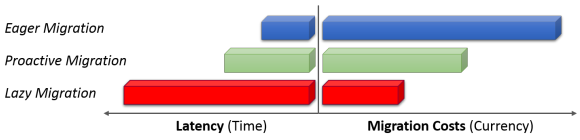

Figure 3: Tradeoffs in data migration (from [17]).

A compromise between the two competing goals is to migrate hot data *proactively* (one of the options in Figure 3). In [23] and [17], different proactive strategies are presented. Depending on the characteristics of the data, the workload, and the schema evolution operations, predictively migrating data promises good latency at moderate costs. In the tutorial, we give a detailed overview over such strategies and their tradeoffs. Further we discuss optimizations, such as the version jumps suggested in [23, 46]. In [17], we propose a first tool providing decision support for the challenge of choosing between migration strategies.

In addition to the impact on costs and latency, it should be noted that for all strategies except eager, the database system or the external schema management component must support *query rewriting*, since data may exist in previous (older) versions and therefore with a different structure than expected by the query [16, 28]. Depending on the heterogeneity class (c.f. Section 3.3), there are additional challenges to query rewriting [27].

In some cases, a *no migration* approach may be necessary, when auditing requires that datasets are preserved in their original version. Then, when an entity in a legacy version is accessed, we may migrate it lazily, but we preserve the original entity.

3.5 Future Opportunities

We finally discuss selected research opportunities related to NoSQL schema evolution and data migration.

Cost Models. For choosing an appropriate data migration strategy, appropriate cost models are needed. These cost models must take into account the characteristics of NoSQL database systems such as distribution, replication, the consistency concepts. A related challenge is the design of a suitable benchmark.

Multi-Model Data. Multi-model database systems like OrientDB, ArangoDB, and Cosmos DB[11] support more than one data model [24]. Similarly, polystores [10, 47] pose new challenges in our context. As we outline in [18], evolution in multi-model databases triggers further research questions, such as inter-model operations, the handling of global vs. local evolution operations, inference of multi-model schemas, and synchronizing migration over different models/systems.

4 INTENDED AUDIENCE AND MATERIAL

Our goal is to give EDBT attendees an overview of the challenges and the current state-of-the-art in both research and practice on NoSQL schema evolution and data migration. We will not assume any background in NoSQL database systems, making our tutorial appropriate for researchers, practitioners, and graduate students.

The material is available at https://tinyurl.com/evolving-nosql.

5 BIOGRAPHIES

Uta Störl is a professor at Darmstadt University of Applied Sciences. Her research focuses on database technologies for Big Data and Data Science. Before, she worked for Dresdner Bank. (uta.stoerl@h-da.de)

Meike Klettke is a professor for Data Science at the University of Rostock. She works on database evolution and reverse engineering of databases. (meike.klettke@uni-rostock.de)

Stefanie Scherzinger is a professor at OTH Regensburg. Her research is influenced by her experience as a software engineer at IBM and Google. (stefanie.scherzinger@oth-r.de)

Acknowledgements: Funded by the Deutsche Forschungsgemeinschaft (DFG) — Projektnummer 385808805.

REFERENCES

[1] F. Abdelhédi, A. A. Brahim, F. Atigui, and G. Zurfluh. MDA-Based Approach for NoSQL Databases Modelling. In *Proc. DaWaK'17*, volume 10440 of *LNCS*, pages 88–102. Springer, 2017.
[2] J. Akoka and I. Comyn-Wattiau. Roundtrip engineering of NoSQL databases. *Enterprise Modelling and Information Systems Architectures*, 13(Special):281–292, 2018.
[3] P. Ataei, A. Termehchy, and E. Walkingshaw. Variational databases. In *Proc. DBPL 2017*, 2017.
[4] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone. Data modeling in the NoSQL world. *Computer Standards & Interfaces*, 67, 2020.
[5] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive JSON datasets. *VLDB J.*, 28(4):497–521, 2019.
[6] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, and H. Voigt. Schema Validation and Evolution for Graph Databases. In *Proc. ER'19*, volume 11788 of *LNCS*, pages 448–456. Springer, 2019.
[7] A. H. Chillón, S. F. Morales, D. Sevilla, and J. G. Molina. Exploring the visualization of schemas for aggregate-oriented nosql databases. In *Proc. ER'17*, pages 72–85, 2017.
[8] A. H. Chillón, D. S. Ruiz, J. G. Molina, and S. F. Morales. A model-driven approach to generate schemas for object-document mappers. *IEEE Access*, 7:59126–59142, 2019.
[9] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema Evolution in Wikipedia - Toward a Web Information System Benchmark. In *Proc. ICEIS'08*, pages 323–332, 2008.
[10] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The BigDAWG Polystore System. *SIGMOD Record*, 44(2):11–16, 2015.
[11] M. Ellison, R. Calinescu, and R. F. Paige. Evaluating cloud database migration options using workload models. *J. Cloud Computing*, 7:6, 2018.
[12] M. H. Farid, A. Roatis, I. F. Ilyas, H. Hoffmann, and X. Chu. CLAMS: Bringing Quality to Data Lakes. In *Proc. SIGMOD '16*, pages 2089–2092. ACM, 2016.
[13] E. Gallinucci, M. Golfarelli, and S. Rizzi. Schema Profiling of Document Stores. In *Proc. SEBD'17*, 2017.
[14] R. Hai, C. Quix, and D. Wang. Relaxed Functional Dependency Discovery in Heterogeneous Data Lakes. In *Proc. ER 2019*, pages 225–239, 2019.
[15] F. Haubold, J. Schildgen, S. Scherzinger, and S. Deßloch. ControVol Flex: Flexible Schema Evolution for NoSQL Application Development. In *Proc. BTW'17*, pages 601–604, 2017.
[16] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner. Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In *Proc. SIGMOD'17*, pages 1101–1116. ACM, 2017.
[17] A. Hillenbrand, M. Levchenko, U. Störl, S. Scherzinger, and M. Klettke. MigCast: Putting a Price Tag on Data Model Evolution in NoSQL Data Stores. In *Proc. SIGMOD'19*, pages 1925–1928. ACM, 2019.
[18] I. Holubová, M. Klettke, and U. Störl. Evolution Management of Multi-model Data - (Position Paper). In *Proc. Poly and DMAH Workshop @ VLDB'19*, volume 11721 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2019.
[19] M. Hulsebos, K. Z. Hu, M. A. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska, Ç. Demiralp, and C. A. Hidalgo. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In *Proc. KDD'19*, pages 1500–1508. ACM, 2019.
[20] J. L. C. Izquierdo and J. Cabot. JSONDiscoverer: Visualizing the schema lurking behind JSON documents. *Knowl.-Based Syst.*, 103:52–55, 2016.
[21] M. Klettke, H. Awolin, U. Störl, D. Müller, and S. Scherzinger. Uncovering the evolution history of data lakes. In *Proc. SCDM'17*, pages 2462–2471. IEEE, 2017.
[22] M. Klettke, U. Störl, and S. Scherzinger. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Proc. BTW'15*, LNI, pages 425–444, 2015.
[23] M. Klettke, U. Störl, M. Shenavai, and S. Scherzinger. NoSQL schema evolution and big data migration at scale. In *Proc. SCDM'16*, pages 2764–2774, 2016.
[24] J. Lu and I. Holubová. Multi-model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.*, 52(3):55:1–55:38, 2019.
[25] M. J. Mior and K. Salem. Renormalization of NoSQL Database Schemas. In *Proc. ER 2018*, pages 479–487, 2018.
[26] C. Moh, E. Lim, and W. K. Ng. DTD-Miner: A Tool for Mining DTD from XML Documents. In *Proc. WECWIS '00*, pages 144–151. IEEE, 2000.
[27] M. L. Möller, M. Klettke, A. Hillenbrand, and U. Störl. Query Rewriting for Continuously Evolving NoSQL Databases. In *Proc. ER'19*, volume 11788 of *LNCS*, pages 213–221. Springer, 2019.
[28] H. J. Moon, C. Curino, and C. Zaniolo. Scalable architecture and query optimization fortransaction-time DBs with evolving schemas. In *Proc. SIGMOD'10*, pages 207–218. ACM, 2010.
[29] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena. Data Lake Management: Challenges and Opportunities. *PVLDB*, 12(12):1986–1989, 2019.
[30] T. Papenbrock, S. Kruse, J. Quiané-Ruiz, and F. Naumann. Divide & Conquer-based Inclusion Dependency Discovery. *PVLDB*, 8(7):774–785, 2015.
[31] T. Papenbrock and F. Naumann. A Hybrid Approach to Functional Dependency Discovery. In *Proc. SIGMOD'16*, pages 821–833. ACM, 2016.
[32] N. Prokoshyna, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava. Combining Quantitative and Logical Data Cleaning. *PVLDB*, 9(4):300–311, 2015.
[33] C. Quix, R. Hai, and I. Vatov. Metadata Extraction and Management in Data Lakes With GEMMS. *CSIMQ*, 9:67–83, 2016.
[34] A. Ringlstetter, S. Scherzinger, and T. F. Bissyandé. Data model evolution using object-NoSQL mappers: Folklore or state-of-the-art? In *Proc. BIGDSE'16*, pages 33–36. ACM, 2016.
[35] D. S. Ruiz, S. F. Morales, and J. G. Molina. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proc. ER'15*, pages 467–480, 2015.
[36] F. J. B. Ruiz, J. G. Molina, and O. D. García. On the application of model-driven engineering in data reengineering. *Inf. Syst.*, 72:136–160, 2017.
[37] K. Saur, T. Dumitras, and M. W. Hicks. Evolving NoSQL Databases without Downtime. In *Proc. ICSME'16*, pages 166–176. IEEE, 2016.
[38] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Bießmann, and A. Grafberger. Automating Large-Scale Data Quality Verification. *PVLDB*, 11(12):1781–1794, 2018.
[39] S. Scherzinger, T. Cerqueus, and E. Cunha de Almeida. ControVol: A framework for controlled schema evolution in NoSQL application development. In *Proc. ICDE 2015*, pages 1464–1467, 2015.
[40] S. Scherzinger, M. Klettke, and U. Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proc. DBPL'13*, 2013.
[41] J. Schildgen, T. Lottermann, and S. Deßloch. Cross-system NoSQL data transformations with NotaQL. In *Proc. BeyondMR@SIGMOD 2016*. ACM, 2016.
[42] W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. C. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, and Y. Yang. Adaptive Schema Databases. In *Proc. CIDR 2017*, 2017.
[43] W. Spoth, T. Xie, O. Kennedy, Y. Yang, B. C. Hammerschmidt, Z. H. Liu, and D. Gawlick. SchemaDrill: Interactive Semi-Structured Schema Design. In *Proc. HILDA@SIGMOD'18*, pages 11:1–11:7. ACM, 2018.
[44] U. Störl, D. Müller, M. Klettke, and S. Scherzinger. Enabling Efficient Agile Software Development of NoSQL-backed Applications. In *Proc. BTW'17*, pages 611–614, 2017.
[45] U. Störl, D. Müller, A. Tekleab, S. Tolale, J. Stenzel, M. Klettke, and S. Scherzinger. Curating Variational Data in Application Development. In *Proc. ICDE'18*, pages 1605–1608, 2018.
[46] U. Störl, A. Tekleab, M. Klettke, and S. Scherzinger. In for a Surprise When Migrating NoSQL Data. In *Proc. ICDE'18*, page 1662. IEEE, 2018. (ICDE Lightning Talk).
[47] M. Vogt, A. Stiemer, and H. Schuldt. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *Proc. SCDM'18*, pages 3364–3373. IEEE, 2018.
[48] L. Wang, S. Zhang, J. Shi, L. Jiao, et al. Schema Management for Document Stores. *Proc. VLDB Endow.*, 8(9), May 2015.
[49] S. Wu and I. Neamtiu. Schema Evolution Analysis for Embedded Databases. In *Proc. ICDEW'11*, 2011.

[11] https://docs.microsoft.com/azure/cosmos-db/