HyDRa: A Framework for Modeling, Manipulating and Evolving Hybrid Polystores

Gobert, Maxime; Meurice, Loup; Cleve, Anthony

Published in:
Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)

Publication date:
2022

Document Version
Version created as part of publication process; publisher's layout; not normally made publicly available

HyDRa: A Framework for Modeling, Manipulating and Evolving Hybrid Polystores

Maxime Gobert, Loup Meurice and Anthony Cleve

PReCISE Research Center, Namur Digital Institute, University of Namur, Belgium
e-mail:{firstname.lastname}@unamur.be

***Abstract*—Data-intensive system evolution is a complex and error-prone process, as most evolution scenarios impact several interdependent artefacts such as the application code, the data structures or data instances. This process is becoming even more challenging with the emergence of heterogeneous database architectures, commonly called hybrid polystores, that rely on a combination of several, possibly overlapping relational and NoSQL databases. This paper presents HyDRa, a framework aiming to facilitate the evolution of polystores thanks to automatically generated data access APIs. For a given polystore, a conceptual API can be derived from the conceptual schema of the polystore and its correspondences with the physical schemas of the underlying databases. Applications built on top of the generated API are then protected from future schema and data reconfiguration changes applied to the polystore. Furthermore, HyDRa automatically enforces cross-database data integrity constraints and does not require developers to master multiple data models and query languages. This paper presents HyDRa and demonstrates its main features based on open-source datasets and realistic use cases.**

I. Introduction

Nowadays, data intensive systems may rely on more than one databases [1], and those databases are not necessarily of the same data model. A recent study [2] reported on the increasing use of heterogeneous database systems, commonly called *hybrid polystores*. Holubova et al [3] explored the challenges that the developers of such systems often face, which include data modeling, data querying and evolution. The HyDRa framework, presented in this paper and depicted in Fig. 1, was developed to address those three challenges.

First, data modeling in a polystore context may prove complex, due to the combination of *relational* and *NoSQL* data models. Relational database design is a well-known and defined process, supported by standard methods. In contrast, NoSQL databases, such as *key-value stores*, *graph databases*, *document databases* and *column stores* all have their own specific data representation patterns, often based on database vendors guidelines or best practices. Physical database design can greatly vary depending on query purposes, performance requirements or technology-specific constraints. Several specific [4] or unifying abstraction [5] design methods and languages exist for NoSQL data modeling, but none of them integrate relational and NoSQL design while allowing developers to specify fine-grained physical data structures.

Second, querying polystores require developers to be familiar with the query language of each database technology used in the polystore. Transversal queries may even involve more than one databases, thus necessitating to write application glue code for data reconciliation. We argue that querying data at the conceptual level gives developers the advantage of being independent from the actual polystore data configuration, and it eliminates the need for writing cross-database queries. Object relational mappers (ORM) or their equivalent for NoSQL technologies exist [6], but (1) very few of them cover multiple data models, and (2) none of them supports the automatic generation of conceptual data manipulation libraries.

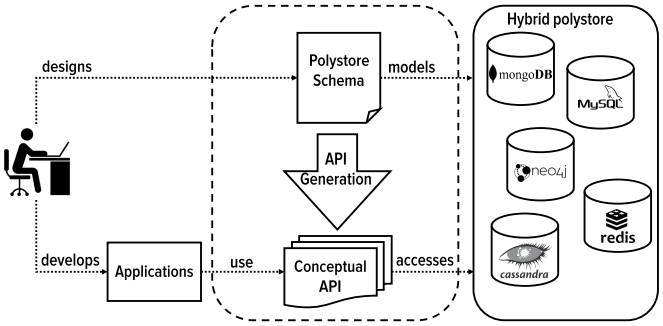


Fig. 1. HyDRa framework

Finally, evolving data-intensive systems relying on a single database is already a difficult task, as several software artefacts (schemas, data and programs) have to be evolved and kept consistent with each other. We argue that this co-evolution process is even more challenging in the context of *hybrid polystores*, due to the multiplicity of data models and the presence of possibly overlapping and interdependent data across multiple databases.

As a contribution to address those three challenges, we present HyDRa (Hybrid Data Representation and Access), depicted in Fig. 1, an integrated framework for conceptual modeling, manipulation and evolution of hybrid polystores. HyDRa's modeling language [7] enables the design of relational and NoSQL databases, by allowing developers to keep full control on physical data representations. HyDRa supports the automatic generation of conceptual data access APIs that (1) facilitate the manipulation of heterogenous databases and (2) reduce the impact of polystore reconfigurations on client applications.

II. HyDRa

This section details the main features of HyDRa, including its modeling language and its support to conceptual API generation and schema evolution.

A. Modeling Language

HyDRa provides a textual modeling language, detailed in [8], to specify (1) the conceptual schema of the polystore, expressed in the *Entity-Relationship* model; (2) the physical schemas of each of its databases (NoSQL or relational), specifying data structures and their fields; and (3) a set of mapping rules to express possibly complex correspondences between the conceptual elements and the physical databases.

For physical schemas, HyDRa currently supports the relational model and the four most popular NoSQL data models, i.e., document, key-value, graph and column-based representations. Mapping rules enable possibly complex design choices, such as *data structure split*, *data instance partitioning*, *data heterogeneity* and *data duplication*.

Table I illustrates the variety of physical structures that can be chosen to implement conceptual constructs. All physical structures can be expressed in HyDRa, via a combination of mapping rules. The databases depicted in *Physical Schema*s column may be either of relational or NoSQL types.

Row **CS (1)** represents the design construct of *entity split* in multiple databases; an entity type can be entirely stored in a single database (a), or its attributes may be split among two databases (b). *Entity data duplication* is also possible, by declaring the same attribute in more than one database.

Row **CS (2)** shows *foreign keys*, *cross-database foreign keys* or *embedding structures* design choices. A one-to-many relationship type can be stored in a single database (a and d), or in different databases (b and c). In a single database, one can use an inner database foreign key reference (a), or an embedded structure (d). Physical schemas (b and c) implement *cross-database* references, which can be *hybrid* when the databases are of different types. In (b), a *mono-valued* foreign key is established between $E1$ and $E2$, i.e., the value in $E1.e2$ references a particular $E2.id$. In (c), a *multi-valued* foreign key is established, in the opposite direction, i.e., each value in $E2.E1$ list references a particular $E1.id$.

Row **CS (3)** shows how a many-to-many relationship type (with attributes) can be physically represented in one (a), two (b) or three databases (c). A physical join structure R stores the relationship type attributes and references particular $E1$ and $E2$ instances (two foreign keys within R).

B. Conceptual API Generation

Using the HyDRa polystore model as input, HyDRa also supports conceptual API code generation. It provides developers with a ready-to-use library, allowing polystore data manipulation operations on the conceptual level. HyDRa automatically generates an API including entity objects, access classes and access methods. Using the API to manipulate polystore data enforces the data heterogeneity, duplication and overlapping constraints declared in the mapping rules.

TABLE I
CONCEPTUAL CONSTRUCTION AND PHYSICAL CORRESPONDENCES

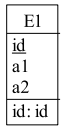
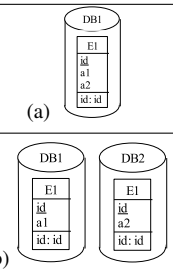
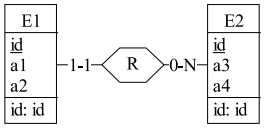
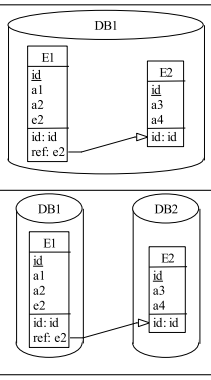
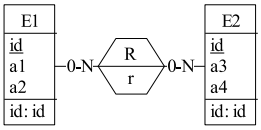
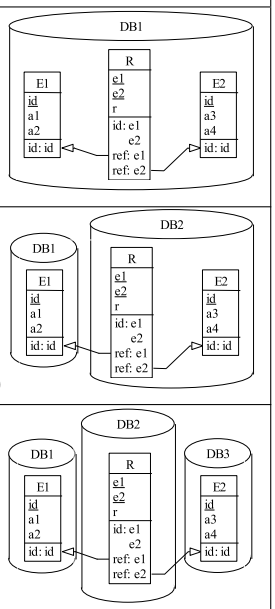
CS#	Conceptual Schema	Physical Schema
(1)		
(2)		
(3)		

Table II shows the generated conceptual methods of the described conceptual schemas (1), (2) and (3) of Table I.

CS (1). CRUD operations are generated for each declared conceptual entity types and relationship types. Selection methods may use a custom *Condition* object, serving as selection condition, that can include a list of *and* or *or* conditions(e.g., $E1.a1 = x \ and \ E1.a2 <> y$). Those conditions specify a value and an operator on any declared conceptual attributes of the concerned entity types. One can also select objects with a specific method taking an attribute value as argument for filtered selection. Methods for inserting, deleting and updating $E1$ instances are also provided. A condition can be used to determine the $E1$ instances to delete/update.

CS (2). The generated API also offers a way to navigate the data by relationship type. Indeed, selection methods can retrieve the list of $E1$ instances related to a given $E2$ instance through R. The reverse access is also possible i.e. reading the $E2$ instance(s) associated with a given $E1$. The return type (list or single instance) directly depends on the role cardinality, i.e., [1-1] or [0-N]. In addition, more general methods are offered to select $E1$ (or $E2$) instances implied in R, according to given *Conditions* objects on $E1$ and $E2$. The selection of R instances (i.e., couples of $< E1, E2 >$ associated via R) is allowed using *Conditions* objects as well. The insertion method of $E1$ is also impacted by the presence of R. Since $E1$ has a [1-1] mandatory role to $E2$, inserting a new $E1$ instance requires to provide, as argument, an existing $E2$ instance linked to the new $E1$ through R.

CS (3). Selection methods similar to CS (2) are generated in the case of a many-to-many relationship type with attributes. The major difference is the possibility to specify a condition on R attributes when reading data. The second difference is the possibility to insert and delete R instances. Inserting a new R instance consists in connecting two existing $E1$ and $E2$ instances via R. Deleting a R instance consists in disconnecting those instances, without deleting them.

The API generation algorithm reads the HyDRa polystore schema and more specifically the mapping rules in order to correctly implement the previously described conceptual methods of the API. The generated code executes native queries to access the different databases involved, joins the results and constructs conceptual object based on raw data.

Table III summarizes the implementation of the conceptual selection methods of Table II, according to the physical representations of Table I. The first row describes the implementation of methods $getE1List(...)$, with $E1$ any declared conceptual entity type. When $E1$ instances are stored in a single database, a single native query is executed and returns a list of $E1$ instances. If a selection condition is specified, the returned instances must respect it. In the case $E1$ instances are stored in several databases, a native query per database is executed. Finally, a procedural full outer join between the query results is necessary to reconciliate the distributed data.

The second and third row describe the implementation of methods $getRList(...)$ in the case of CS (2) and CS (3), respectively. If a single database is used to store R, a single inner join query is executed. This is only possible if the database has a query language supporting the join operator, e.g., SQL. If the R relationship type is distributed across multiple databases, the join operation is performed procedurally. The implementation of methods $getE1ListInR(...)$ and $getE2ListInR(...)$ are not described in the table, but they follow the same logic.

TABLE II
CONCEPTUAL CONSTRUCTION AND GENERATED API FUNCTIONS

CS#	Generated Conceptual Methods	Description

(1)	`getE1List(Condition<E1> cond)`
	`getE1ListById(Object idvalue)`
	`getE1ListByA1(Object a1value)`
	`getE1ListByA2(Object a2value)`
		Read, from every database mapped to $E1$, a list of $E1$ instances, according to the given selection condition or attribute value.

	`insertE1(E1 newObject)`
	`deleteE1(E1 toDelete)`
	`deleteE1(Condition<E1> cond)`
	`updateE1(E1 updatedE)`
	`updateE1(Condition<E1> cond,`
	`Set<E1> set)`
		Insert, delete or update $E1$ instances in mapped databases. A deletion/update condition can be mentioned. For update methods, a Set clause can be used to specify which attributes and values should be updated.

(2)	`getE1ListInR(E2 rE2)` `getE2InR(E1 rE1)`	Read the E2 instance (resp. the list of E1) associated with a given instance of E1 (resp. E2) through R. The return type (list or single instance) depends on the role cardinality.

	`getE1ListInR(Condition<E1> c1,`
	`Condition<E2> c2)`
	`getE2ListInR(Condition<E1> c1,`
	`Condition<E2> c2)`
		Read the list of $E1/E2$ instances implied in R, according to particular selection conditions (that can be combined) on $E1$ and $E2$.

	`getRList(Condition<E1> c1,` `Condition<E2> c2)`	Return a list of R instances (i.e., list of <$E1$, $E2$>, such $E1$ is associated with $E2$ through R) according to particular selection conditions on $E1$ and $E2$.
	`insertE1(E1 e1, E2 e2)`	Insert an $E1$ instance and connect it to an existing $E2$ instance. This link is mandatory due to the [1-1] role cardinality of R.

(3)	`getE1ListInR(Condition<E1> c1,`
	`Condition<E2> c2,`
	`Condition<R> cR)`
	`getE2ListInR(Condition<E1> c1,`
	`Condition<E2> c2,`
	`Condition<R> cR)`
		Read the list of $E1/E2$ instances implied in R, according to particular selection conditions on $E1$, $E2$ and R (condition on R is allowed due to the existence of attributes).

	`getE1ListInR(E2 rE2)` `getE2ListInR(E1 rE1)`	Read the list of E1 (resp. E2) instances associated with a given instance of E2 (resp. E1) through R.

	`getRList(Condition<R> cR,`
	`Condition<E1> c1,`
	`Condition<E2> c2)`
		Returns a list of R, according to particular selection conditions on $E1$, $E2$ and R.

	`insertR(Object rAttr,`
	`E1 e1Instance,`
	`E2 e2Instance)`
	`deleteR(R r)`
		Associate (insert) and dissociate (delete) existing $E1$ and $E2$ instances through R.

C. HyDRa and Evolution

Evolving data intensive systems is a complex and error prone task, since it often requires to co-evolve multiple artefacts, including data structures, data instances, queries and application code, to keep the system consistent. HyDRa, by generating conceptual data manipulation APIs, aims to facilitate this co-evolution problem, as illustrated in Fig. 2. The

TABLE III
PHYSICAL SCHEMAS AND THEIR IMPLEMENTATION OF GET METHODS

CS#	PS#	Description Implementation
1	(a)	Generate a single database query to get $E1$ data.
	(b)	- Generate a db query to get $l1$, the list of $E1$ stored in $DB1$. - Generate a db query to get $l2$, the list of $E1$ stored in $DB2$. - Perform a procedural full outer join between the two lists based on the id matching ($l1 \cup l2$).
2	(a)	Generate a single db inner join query based on the foreign key value (iff $DB1$ permits it, e.g., SQL. Otherwise, see 2(b))
	(b)	- Generate a db query to get $l1$, the list of $E1$ stored in $DB1$. - Generate a db query to get $l2$, the list of $E2$ stored in $DB2$. - Perform a procedural inner join between the two lists based on the foreign key value ($l1 \cap l2$).
	(c)	- Generate a db query to get $l1$, the list of $E1$ stored in $DB1$. - Generate a db query to get $l2$, the list of $E2$ stored in $DB2$. - Perform a procedural inner join between the two lists based on the multi-valued foreign key value.
	(d)	Generate a single db query to get the list of nested $E1$ ($DB1$ permits nested structures, e.g., MongoDB)
3	(a)	Generate a double inner join query based on the two foreign key values (iff $DB1$ permits it, e.g., SQL. Otherwise, see 3(c)).
	(b)	- Generate a single db inner join query between R and $E2$ (iff $DB2$ permits it, e.g., SQL. Otherwise, see 3(c)). - Generate a db query to get the list of $E1$ stored in $DB1$. - Perform a procedural inner join between both results.
	(c)	- Generate a db query to get $l1$, the list of $E1$ stored in $DB1$. - Generate a db query to get $l2$, the list of $E2$ stored in $DB3$. - Generate a db query to get $l3$, the list of R stored in $DB2$. - Perform a procedural double inner join between $l1$, $l2$ and $l3$ based on the two foreign key values.

upper left of the figure depicts a first version *v1* of a polystore system, manipulating *Product*, *Order* and *Customer* entities. For readability reasons, relationship types and attributes have been omitted. The mapping rules are represented with arrows, linking the conceptual elements to their physical databases. *Customer* instances are stored in a relational MySQL database, *Order*s are stored in MongoDB database and *Product* data is stored in both databases. This *v1* version of the polystore schema was given as input to the HyDRa API generator. Based on the generated API, developers developed application code accessing the polystore databases declared in *v1*.

Let us now assume that evolving requirements require the polystore to evolve from version *v1* to *v2*. *Customer* is migrated to the MongoDB database and *Order* is duplicated in a newly deployed key-value Redis database. Without HyDRa, application programs would use database native queries to access the polystore data. Those programs would need to be manually rewritten to (1) change the queries manipulating *Customer* from MySQL to MongoDB, (2) add Redis queries for *Order* data, (3) add glue code to handle the duplication of *Order* data in multiple databases. On top of that, the developers would need to know, or to learn, the Redis query language.

In contrast, by using the HyDRa API conceptual methods to access the polystore in version *v1*, the developers only need to adapt the HyDRa polystore model, regenerate the API, and recompile their application programs. The latter remain unchanged and can immediately manipulate the *v2* data structures of the target polystore.

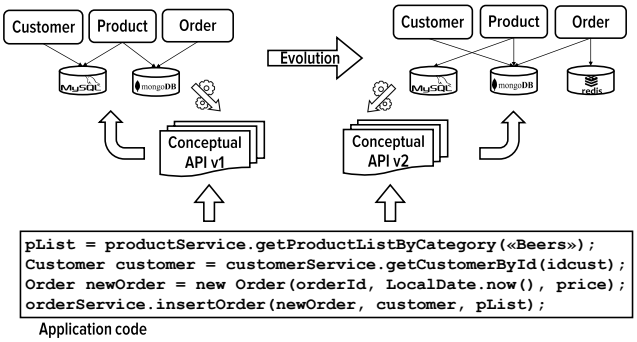


Fig. 2. HyDRa API in evolution context

III. IMPLEMENTATION AND PRELIMINARY EVALUATION

HyDRa is implemented as an Eclipse plugin, publicly available on GitHub [7]. The plugin includes a textual editor for polystore schemas, and a conceptual Java API generator. The modeling language grammar was specified with Xtext [1]. The editor provides auto completion, syntax checking and highlighting. It currently supports the design of relational, document, graph, column and key-value databases.

The user can create a polystore schema file to specify the conceptual and physical schemas as well as mapping rules to possibly pre-existing databases. This file can then be given as input of the API generator, that then produces a set of ready-to-use Java classes with their configuration files. The API generator uses the Acceleo technology [2]. The generated API currently supports selection and insertions operations on relational databases (MySQL and MariaDB), document databases (MongoDB) and key-value databases (Redis). Database access is possible either through native access libraries or via Apache Spark [3] implementations. Spark provides easy-to-use processing facilities for distributed data, with promising levels of scalability.

HyDRa has been applied to several illustrative use cases relying on the IMDB [4] and Unibench[5] [9] datasets. Unibench is a framework for testing multi-model DBMSs, i.e., single database platform handling multi-model data. They generated data simulating an e-commerce system in multiple data formats. For example social network information was stored as graphs, purchase history as documents and customer information in relational tables.

The document, key-value and relational datasets of Unibench have been successfully modeled and queried using HyDRa framework. Some of its queries have been written using the generated conceptual API. As an example, query 2, which states *For a given product, find the persons who had bought it*, can be expressed using the HyDRa API, as shown in Fig. 3.

We then simulated several polystore reconfigurations, by migrating data and modifying data structures. After API regen-

[1] https://www.eclipse.org/Xtext/ [2] https://www.eclipse.org/acceleo/
[3] http://spark.apache.org/ [4] https://www.imdb.com/interfaces/
[5] https://github.com/HY-UDBMS/UniBench

```
Condition condProdId = Condition.simple(
        ProductAttribute.id, Operator.EQUALS, "B0036UNK0I");
Condition condDate = Condition.simple(
        OrderAttribute.orderdate, Operator.GT, LocalDate.of(2021, 9, 21));
orderDataset = orderService.getOrderList(Order.composed_of.orderP, condDate, condProdId);
List<Customer> customerList = new ArrayList<>();
for (Order o : orderDataset.collectAsList()) {
    Customer cust = customerService.getCustomer(Customer.buys.client, o);
    customerList.add(cust);
}
customerList.forEach(System.out::println);
```

Fig. 3. HyDRa API code query example

eration, we observed that conceptual queries were still valid and could be successfully executed on the target structures. This demonstration is available in a short video[6] and on GitHub [7].

IV. Related Work

Database design for NoSQL applications is not as mature as relational database design. State-of-the art approaches are mainly technology or data model specific [10]–[12]. Roy-Hubara et al. [4] did a systematic literature review on NoSQL database design. NoAM [5] proposes a uniform way to design NoSQL databases by abstracting the common features of each data model and by designing an aggregate identification step. Herrero et al. [13] define a 3-step design method, with data access performance as main objective. Conceptual design takes evolution into account by means of flags on entities. Flags are considered when deciding to map an entity to a NoSQL or a relational database. TyphonML [14] also supports conceptual modeling of hybrid polystores, but imposes implicit restrictions on the way conceptual entities, attributes and relationships are physically translated in each different native backend. Fernandez et al. [15] propose *U-Schema*, a unified language, combining relational and NoSQL data models. Orion [16] is a language for NoSQL schema evolutions, defined on top of *U-Schema*. It allows one to generate scripts that modify the unified schema and the underlying databases, but does not provide support for generating or evolving data manipulation programs. The major novelty of the HyDRa modeling language relies on the use of fine-grained mapping rules, enabling to model data overlapping across heterogeneous databases, and to keep full control over the physical data structures.

Cleve et al. [17] propose to generate conceptual data access APIs for systems relying on a single relational database. Guidoni et al. [18] also allow the conceptual manipulation of a relational database, using ontology-based data mappings. Object Relational Mappers (ORM) and Object NoSQL Mappers (ONM) [6] also provide access to data using conceptual APIs, but very few support multi-database technologies and the class implementations have to be manually written.

The novel contribution of HyDRa in terms of conceptual API generation is its ability to support multi-model data access. The generated API is able to handle the variety of data storages, even for querying the same entity type, by means of several native queries and join-based data reconciliation. The physical configuration of the polystore remains transparent for the developers, thanks to the high-level of abstraction of conceptual API methods. This also facilitates schema evolution by reducing manual co-evolution effort.

[6] https://youtu.be/oTTFhHpt9IY [7] https://tinyurl.com/wxna468

V. Conclusion and Future Work

This paper presents HyDRa, an integrated conceptual framework for hybrid polystore modeling, manipulation and evolution. HyDRa includes a modeling language able to *conceptually design hybrid polystores* while preserving the possibility to design data at physical level to exploit the strengths of each native data model. As far as data manipulation is concerned, HyDRa follows a generative, conceptual approach that (1) eliminates the need to master multiple query languages, (2) ensures the data keep complying with cross-database integrity constraints and (3) is resistant to database schemas changes.

As future work, we plan to provide better assistance to polystore schema design. In particular, as NoSQL database design is mainly query-centered, existing query logs could be exploited to recommend adequate mapping rules to particular database technologies. Regarding API generation, we intend to support graph and wide column datastores, and to conduct systematic performance evaluations of HyDRa.

Acknowledgements. This research is supported by the F.R.S.-FNRS and FWO via the EOS project 30446992 SECO-ASSIST.

References

[1] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013.
[2] P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "An empirical study of (multi-) database models in open-source projects," in *ER*, 2021.
[3] I. Holubová, M. Svoboda, and J. Lu, "Unified management of multi-model data," in *ER 2019*. Springer.
[4] N. Roy-Hubara and A. Sturm, "Design methods for the new database era: a systematic literature review," *SoSyM 2020*.
[5] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone, "Database design for nosql systems," in *ER*. Springer, 2014, pp. 223–231.
[6] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger, "Schemaless nosql data stores-object-nosql mappers to the rescue?" *BTW 2015*.
[7] M. Gobert, "HyDRa repository," 2021. [Online]. Available: https://github.com/gobertm/HyDRa
[8] M. Gobert, L. Meurice, and A. Cleve, "Conceptual modeling of hybrid polystores," in *ER 2021*.
[9] C. Zhang, J. Lu, P. Xu, and Y. Chen, "Unibench: A benchmark for multi-model database management systems," in *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 2018.
[10] C. de Lima and R. dos Santos Mello, "A workload-driven logical design approach for nosql document databases," in *iiWAS*, 2015, pp. 1–10.
[11] G. Rossel, A. Manna *et al.*, "A modeling methodology for nosql key-value databases," *Database Syst. J*, vol. 8, no. 2, pp. 12–18, 2017.
[12] J. Pokorný, "Conceptual and database modelling of graph databases," in *IDEAS16*.
[13] V. Herrero, A. Abelló, and O. Romero, "Nosql design for analytical workloads: variability matters," in *ER*. Springer, 2016, pp. 50–64.
[14] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Pierantonio, and L. Iovino, "Typhonml: a modeling environment to develop hybrid polystores," in *MoDELS*, 2020.
[15] C. J. Fernández Candel, D. Sevilla Ruiz, and J. J. García-Molina, "A unified metamodel for nosql and relational databases," *arXiv*, 2021.
[16] A. H. Chillón, D. S. Ruiz, and J. G. Molina, "Towards a taxonomy of schema changes for nosql databases: The orion language," in *ER 2021*.
[17] A. Cleve, A.-F. Brogneaux, and J.-L. Hainaut, "A conceptual approach to database applications evolution," in *ER*. Springer, 2010, pp. 132–145.
[18] G. L. Guidoni, J. P. A. Almeida, and G. Guizzardi, "Forward engineering relational schemas and high-level data access from conceptual models," in *ER*. Springer, 2021, pp. 133–148.