DORM: Dynamic Object-Relational Mapping

Alberto Abelló[1,*,†], Enrico Gallinucci[2,†]

[1]U. Politècnica de Catalunya (UPC), Barcelona, Spain
[2]U. of Bologna, Cesena, Italy

Abstract
The relevance of physical database design is out of question since long time ago. However, what is becoming evident these days is the need to reconsider our database schema more and more often, as a reaction to constant query workload changes. Thus, we need to enhance logical independence to support frequent schema evolution reacting to those changes, and shield applications from it. Indeed, we propose DORM as a system to migrate data from the current schema to a new one, and automatically generate the queries corresponding to either one or the other from an abstract representation in terms of application domain concepts. Our experiments show the impact of schema evolution in the performance of the DBMS, and demonstrate how DORM allows to effortlessly move from one schema to another.

Keywords
Database design, Object-Relational Mapping, Local As View

1. Introduction

The relevance of physical database design is well known since the beginning of Relational DBMSs. Beyond purely indexing, this also includes reshaping the data storage structures by using different degrees of normalization and partitioning. Together with NOSQL tools also came the wave of workload-driven design methodologies (e.g., encouraging denormalization to minimize joins), which has motivated several research efforts to identify and recommend optimal schema designs for a given domain and workload (e.g., [1, 2, 3]). Moreover, this kind of tools also adopt schemaless data models, offering fluidity to application functionality evolution. Nevertheless, allowing the evolution of application functionalities does not mean storage structures underneath are being optimized, rather the opposite.

Indeed, in our recent work [4], we outlined the problem of *database refactoring*: with workload-driven design, the changes in workloads (i.e., changes in the frequency of the different queries or access patterns) entail the need to (frequently) revise the existing physical design. Our investigation, based on a real case study, demonstrated that initial design optimization can become detrimental in the light of workload evolution, severely hindering the DBMS performance.

Thus, there is a need for frequent database redesign, which mainly adds an extra cost to (1) the computing resources, due to data movement itself from old to new structures; and (2) the human resources required to adapt the client applications (i.e., generate migration scripts and rewrite existing queries using the new design). Our focus is the latter, which requires applications to be somehow independent of database physical design.

Beyond the usage of ontologies to facilitate the database design [5], a case for logical data independence was recently made in [6], acknowledging the database evolution problem and envisioning the use of mappings over a global Entity-Relationship model to define schemas and rewrite queries. Here, we take it a step further by formalizing the expressiveness of our system in terms of a hypergraph with asserted integrity constraints (IC), evaluating it under a computing perspective on a real case study, and

DOLAP 2026: 28th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data, co-located with EDBT/ICDT 2026, March 24, 2026, Tampere, Finland
*Corresponding author.
†These authors contributed equally.
alberto.abello@upc.edu (A. Abelló); enrico.gallinucci@unibo.it (E. Gallinucci)
https://alberto-abello.staff.upc.edu (A. Abelló); https://www.unibo.it/sitoweb/enrico.gallinucci (E. Gallinucci)
0000-0002-3223-2186 (A. Abelló); 0000-0002-0931-4255 (E. Gallinucci)

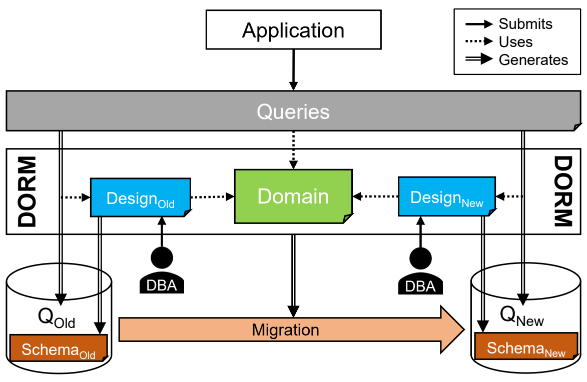


Figure 1: System overview

releasing the tool with comprehensive instructions and examples. The same topic of data independence has been further pushed by [7], which proposed a thought experiment to rebuild the foundations of Relational database technology in terms of mappings - remarking the interest in this line of research.

The system proposed in this paper is called DORM (Dynamic Object-Relational Mapping) and contributes to the simplification and automation of database schema evolution, supporting its management and reducing human effort. Indeed, a limitation identified in [4] was the need to manually define scripts to migrate the data from one schema design to another, as well as to rewrite workload queries in accordance with the new schemas. Our current work fills such a gap through *logical data independence* by simplifying the definition of new schema designs and automating data migration across designs as well as the rewriting of queries from one design to another. For the prediction of workload changes to decide when to migrate, we would rely on work like [8, 9], and for index selection on [10].

Figure 1 shows DORM overview, whose main elements are:

Domain: Concepts to be represented in the database in terms of classes, attributes, and relationships (painted green in all figures and corresponding code snippets).

Design: Abstraction of the structure of the database expressed in terms of *Structs* and *Sets* (as in [11]) (painted blue in all figures).

Schema: Implementation of the *Design* in a given DBMS (i.e., Relational or not), following some paradigm (i.e., flatten relational, object-relational, JSON, XML).

Starting with a *Domain* representing the concepts of interest (a.k.a. the global schema) and a *Design* abstracting how the data are organized (a.k.a. mappings), DORM automatically generates the corresponding database *Schema* (a.k.a. local schema). Thus, the application can express the required queries in an abstract way in terms of the *Domain* concepts, which are automatically translated through the *Design* into the concrete queries to be executed in the DBMS underneath. As soon as workload changes and performance degrades are detected, we can create a new *Design*, which generates both the corresponding database *Schema* and the corresponding statements to migrate the data from the old one - all the while changing neither the queries' abstract declaration nor the application, which remains shielded by DORM.

The contributions provided with this paper are:

1. A generic catalog that manages the hypergraph-based representation of *Domain* and *Design* information (including its corresponding constraints), so that the database *Schema* can be automatically generated from them, either as Relational or not (e.g., JSON).
2. A mechanism to automate the migration of data between the *Schemas* of different catalogs, to accommodate workload changes.
3. A design-agnostic representation of queries based exclusively on the *Domain* concepts, and a mechanism to generate their corresponding DBMS-dependent code.
4. A functional prototype DORM implemented on PostgreSQL, including extensive experiments on its usefulness with real data and query workloads from Sloan Digital Sky Survey (SDSS) organization.

The outline of the paper is as follows. Section 2 discusses the related literature. A motivating example is introduced in Section 3, while Section 4 presents the proposed solution. Experiments are discussed in Section 5 and the conclusions drawn in Section 6.

2. Related Work

Database design Given the workload-driven nature of NoSQL data modeling [12, 13], researchers have dedicated abundant effort to aid users in identifying the logical schema most suitable to a given workload. The definition of *most suitable* can vary, as well as the way to identify it: some papers use heuristics to directly generate one [1, 14, 15, 16, 17, 18] or more [19] target schemas, others evaluate more alternative schemas based on a single criterion [20, 21, 2, 22, 23] or multiple thereof [24, 25, 3].

Schema evolution However, all those works limit their scope to the *initial* design of a logical schema (i.e., none of them considers the challenge of implementing such schema by refactoring an existing one). Nonetheless, database schemas are rarely static; schema evolution is common in the early stages of a project [26] and encouraged by the NoSQL schemaless characteristic (i.e., the concept of attaching schema information directly to each data item instead of at the table/collection level). Researchers have looked for patterns in the evolution of schemas in both Relational [26, 27, 28], embedded [29], and NoSQL databases [30, 31], as well as Wikipedia [32], open source software projects [33], and data-intensive applications [34]. Recent efforts to support and/or automate the management of schema evolution have been directed toward keeping track of different schema versions [35], propagating manually-defined schema modification operations (SMO) to the database [36, 37, 38] and to queries [39], and evaluating multiple strategies to apply schema changes to the data [40, 37, 41, 42].

Polystore systems There are many proposals to manage heterogeneous storage systems. Thus, we find polyglot systems [43], which aim to simplify the formulation and/or optimize the execution of queries over heterogeneous stores. For instance, [44, 45] rely on a high-level dataspace as a global, integrated view over multiple stores and translate the queries over the dataspace into queries to be executed on the local stores. On the other hand, polystores (e.g., [46]) adopt a similar approach, but enable the rewriting of queries from one database system to the others. Though mappings between schema designs are used, the focus in this line of work is more on the integration of data between different sources. DORM distinguishes itself from previous work in that its focus is on the migration from one system (or database schema) to another, thus facilitating the tedious task of database refactoring and paving the way to the automated evaluation of multiple schema design alternatives.

Mapping management From a broader perspective, mappings have been used extensively in the past to address data integration and query rewriting issues [47]. We refer the reader to foundational literature on Global- and/or Local-As-View (GAV, LAV, GLAV) [48] and to recent surveys on their usage for data integration [49, 50] and query answering [51] purposes. DORM essentially adopts a LAV approach, with *Designs* being defined as views over a global (conceptual) *Domain*, and with queries being formulated on the same global Domain and translated onto the local *Schema*; despite the higher complexity of LAV approaches [48], many works have proposed it and shown its feasibility [52, 53, 54].

3. Use Case Description

SkyServer[1] is a publicly available Relational database designed to map the cosmos, made available by SDSS. Access to SDSS data is provided via a web interface, where users can query and download data through either SQL or interfaces designed for both professional astronomers and educational purposes.

Besides the astronomical data themselves, the server also makes public the SkyServer Traffic Log,[2]which captures statistics on SQL queries being executed. This includes columns such as `theTime` (datetime of the query), `webserver` (URL of the server), `winname` (Windows name of the server), `clientIP` (client's IP address), `sql` (the SQL statement executed), and `error` (returned code), among

[1]https://skyserver.sdss.org/dr18
[2]https://skyserver.sdss.org/log/en/traffic/sql.asp

others. It also captures performance metrics like `elapsed` (query execution time), `busy` (CPU time used by the query), and `rows` (number of rows returned by the query), providing a comprehensive snapshot of server activity.

In our previous work [4], we already extensively analyzed the changes in the workload of SkyServer along years, to conclude that the workload significantly changed with different releases. These changes were relevant enough to justify schema changes in the underlying database, whose investment in execution time was returned in a few hours or at most days (depending on the release). The open question, though, was to what extent the human migration effort (in terms of rewriting SQL code as a response to the new schema) was justified or not.

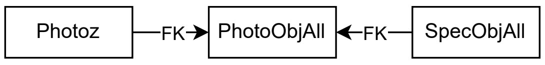


Figure 2: SDSS tables used

Table	Columns	Rows	Size
`PhotoObjAll`	509	99,980	783MB
`SpecObjAll`	195	24,739	39MB
`Photoz`	35	16,925	3.6MB

Table 1: Statistics of source tables

In this work, we decided to use in our experiments data corresponding to Release 18 (available since December 2023). From this, we sampled the three tables shown in Figure 2 with their corresponding Foreign Keys (FKs). There, we find data captured from different astronomical objects observed as `PhotoObjAll`, whose spectroscopic data are also captured and stored per wavelength intervals into `SpecObjAll`. Moreover, their photometric redshift estimations can be found separately in `Photoz`. Statistics of the extraction can be seen in Table 1.

4. Proposed Solution

Our proposal is a system (DORM) to facilitate schema evolution by automating data migration and dynamically adapting queries. The user expresses both database design and queries in an abstract way in terms of *Domain* concepts, and the system automatically generates (1) the corresponding database schema, (2) the migration statements to move data from the existing schema to the new one, and (3) the queries over the new schema. Section 4.1 specifies the catalog information required in the form of a hypergraph; Sections 4.2 and 4.3 respectively explain how the DBMS-specific *Schema* and data manipulation statements are generated from that (with migration statements being queries over the old *Schema* that appropriately reshape the data); Section 4.4 explains how we overcome existing limitations.

4.1. Hypergraph representation

We use a hypergraph formalization (instead of logics), because it seamlessly allows both theoretical definitions as well as direct implementations. Figure 3 shows the content of the hypergraph that governs the behavior of DORM. This is a generalized hypergraph formalized as a triple $\mathbb{H} = \langle \mathbb{N}, \mathbb{E}, \mathbb{G} \rangle$, where hyperedges (called simply *Edges* from here on) can connect not only *Nodes*, but also other *Edges*, and both *Nodes* and *Edges* are uniquely identified by a *name*. Hence, $\mathbb{N} = \{n_1, \cdots, n_n\}$ and $\mathbb{E} = \{E_1, \cdots, E_m\}$ are nodes and edges, so that $\forall i \in [1, m] : E_i \subseteq \mathbb{N} \cup \mathbb{E} \setminus E_i$.[3] $\mathbb{G}$ is just a set of annotations of the *Domain* called *Guards* and will be described later in Section 4.4. Due to lack of space, we do not explicit here the tens of associated Integrity Constraints (IC), but they are summarized in the project documentation.[4]

Logically, $\mathbb{H}$ can be seen as composed of (a) *Domain* information (namely *Attribute*, *Class*, *Association*, and *Generalization*, highlighted by green zigzag lines in the figure), and (b) *Design* information (namely *Struct* and *Set*, with diagonal blue lines in the figure).

Domain corresponds to the information we want to represent in our database. We firstly find a specialization of *Nodes* that, following UML terminology, correspond to *Attributes*. Then, *Classes* are the kind of *DomainEdges* that contain some of those *Attributes*, and are related to each other

[3]We use lower case for single elements, capitals for sets, and double-line capitals for sets of sets or complex structures.
[4]https://github.com/alberto0723/DORM/blob/main/README.md

via *Relationships* (either *Association*, or *Generalization*) that connect some *Classes*. An *Association* has exactly two ends, each with min and max multiplicities; a *Generalization* has one superclass and one or more subclasses, besides a *Discriminant Attribute* (used in the corresponding *constraint* predicates to determine to which of the subclasses an instance belongs to). *Generalizations* can be tagged as *Complete* and/or *Disjoint*.

Design indicates, in an abstract/generic way, how the *Domain* information has to be represented in the database. Following the terminology in [11, 55], we use *Structs* and *Sets*. The former provides structural information about an instance (e.g., a tuple or document), while the latter abstracts a container of instances (e.g., a table or collection) of the kind of some *Struct*. Thus, a *Struct* contains a set of *Nodes* and *Edges* of any kind; we call *Anchor* the subset of a *Struct*'s content (limited to *Classes* or *Associations*) whose identifiers are used to distinguish its instances, hence, providing its semantics. Differently, a *Set* can only contain some *Structs* (exceptionally, we simplify notation by allowing it to also contain one *Class*, as we will see later). We call "rooted" those elements not contained by any other design construct.

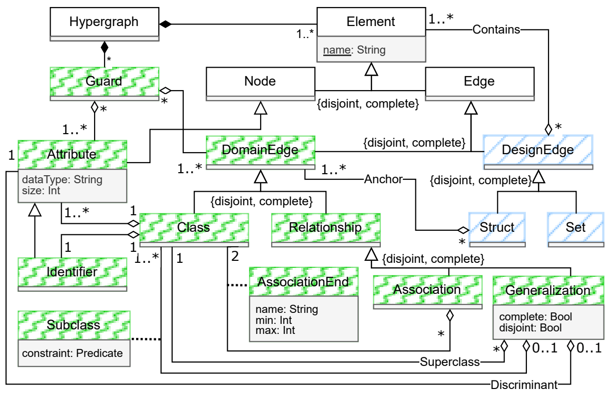


Figure 3: Diagram of the content in the hypergraph

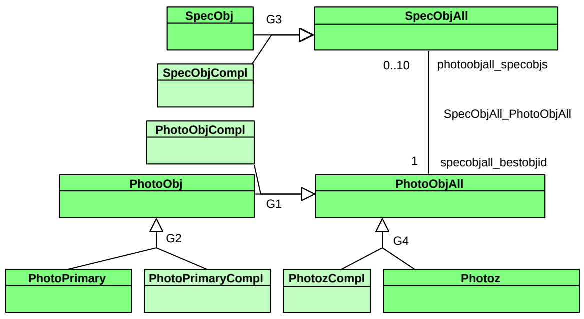


Figure 4: Example of DORM *Domain*;[5] in light green are complementary *Classes* to make *Generalizations* complete

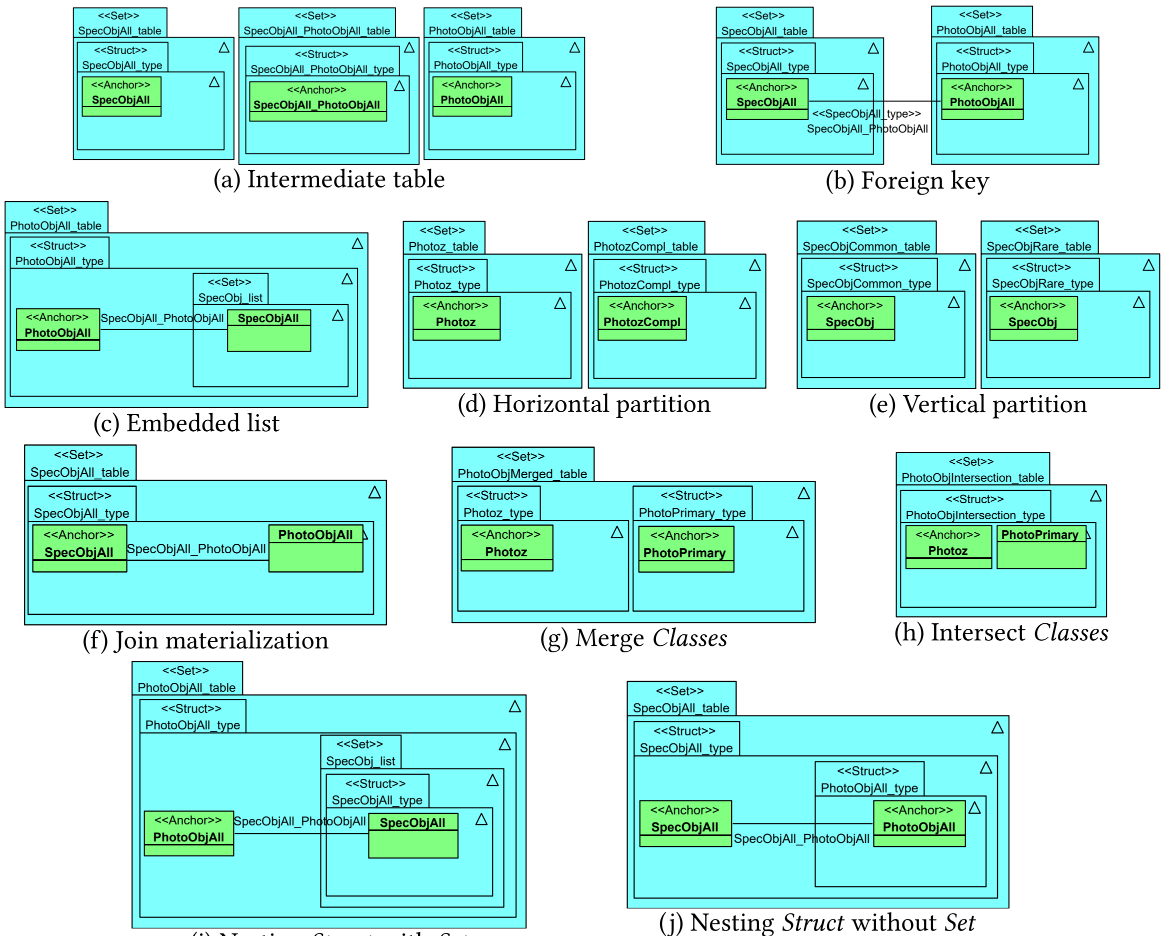
Example 1. *Figure 4 shows our conceptualization of the tables in Figure 2. Firstly, we can see at RHS the Classes* `SpecObjAll`*,* `PhotoObjAll`*, and* `Photoz` *corresponding to original tables. Then, the FK between the first two is shown as an Association* `SpecObjAll_PhotoObjAll` *with AssociationEnds* `photoobjall_specobjs` *and* `specobjall_bestobjid` *(names can be freely chosen, as long as they are unique among all the elements). Differently, the FK between the last two has been represented as Generalization G4, because said FK was also PK of* `Photoz`*, which corresponds to the Relational implementation pattern of a generalization/specialization relationship. Moreover, we have conveniently made explicit three other Generalizations for* `SpecObj`*,* `PhotoObj`*, and* `PhotoPrimary`*. These exist as views in SkyServer and frequently appear in its queries. In all four cases, the corresponding Discriminant has been defined, but like any of the other hundreds of attributes, not graphically represented for the sake of simplicity. Finally, we have added a complementary Class to every Generalization (in light green) to make all of them complete. This is not really necessary, but it facilitates the lossless representation of horizontal partitions.*

Example 2. *In Figure 5, we can see some partial snippets of representative Designs alternatives (these are just examples, since Sets and Structs can be freely combined at will as soon as they respect the ICs):*

a) The Classes and the Association are stored separately. For each of them, we see a Set (representing the corresponding collection/table) containing a Struct with the corresponding concept (indicating the content/type of every instance; attributes are omitted for simplicity). The Anchor annotation indicates which of the potentially many concepts inside give meaning to (i.e., identifies) that content.

b) A collection/table has a pointer to the other. The annotation on top of the Association name indicates where the pointer must go (the name of the corresponding attribute is that of the AssociationEnd).

[5]Generated with Visual Paradigm Community Edition v17.2.



<<Set>> SpecObjAll_table
<<Struct>> SpecObjAll_type
<<Anchor>> SpecObjAll
<<Set>> SpecObjAll_PhotoObjAll_table
<<Struct>> SpecObjAll_PhotoObjAll_type
<<Anchor>> SpecObjAll_PhotoObjAll
<<Set>> PhotoObjAll_table
<<Struct>> PhotoObjAll_type
<<Anchor>> PhotoObjAll
<<SpecObjAll_type>> SpecObjAll_PhotoObjAll
<<Set>> SpecObj_list
SpecObjAll_PhotoObjAll
<<Set>> Photoz_table
<<Struct>> Photoz_type
<<Anchor>> Photoz
<<Set>> PhotozCompl_table
<<Struct>> PhotozCompl_type
<<Anchor>> PhotozCompl
<<Set>> SpecObjCommon_table
<<Struct>> SpecObjCommon_type
<<Anchor>> SpecObj
<<Set>> SpecObjRare_table
<<Struct>> SpecObjRare_type
<<Set>> PhotoObjMerged_table
<<Struct>> PhotoPrimary_type
<<Anchor>> PhotoPrimary
<<Set>> PhotoObjIntersection_table
<<Struct>> PhotoObjIntersection_type
PhotoPrimary

(a) Intermediate table (b) Foreign key (c) Embedded list (d) Horizontal partition (e) Vertical partition (f) Join materialization (g) Merge *Classes* (h) Intersect *Classes* (i) Nesting *Struct* with *Set* (j) Nesting *Struct* without *Set*

Figure 5: Examples of DORM *Design*[5]

c) *Usable only in NF^2, as it generates a list of pointers to* `SpecObjAll` *for every* `PhotoObjAll` *instance.*
d) *A horizontal partition of* `PhotoObjAll` *according to the specialization* `G4`*. Depending on whether they belong to* `Photoz` *or* `PhotozCompl`*, instances go to a different collection/table.*
e) *A vertical partition of* `SpecObj`*. For the sake of simplicity, attributes are not shown, but LHS Struct would only contain the most commonly used, while RHS one would contain those rarely accessed.*
f) *Having the information of* `PhotoObjAll` *replicated into every* `SpecObjectAll`*. Notice that both are in the same Struct, but the latter is its Anchor (i.e., defines its meaning).*
g) *Instances of different Classes like* `PhotoPrimary` *and* `Photoz` *are in the same Set (notice they belong to the same Generalization hierarchy, hence share the identifying Attribute).*
h) *Disconnected Classes* `Photoz` *and* `PrimaryObj` *in the same Struct, which is interpreted as the collection/table having the instances in the intersection of both.*
i) *Nesting a list of instances of* `SpecObjAll` *inside every* `PhotoObjAll` *(this is different from Figure 5c in that now the list will contain complex instances and not just identifiers).*
j) *A single* `PhotoObjAll` *instance (not a set) resides inside the Struct of another Class* `SpecObjAll`*.*

These are only some examples of the many feasible *Designs* (e.g., in Figure 5c, we could include in `SpecObj_list` a complex *Struct* with any number of attributes instead of a simple *Class*; in Figure 5f, we can materialize any subgraph of to-one *Associations* and *Classes* instead of the single one depicted).

4.2. *Schema* Generation

The information stored in our hypergraph allows to generate the database corresponding to the *Design* in any kind of DBMS. For example, in the case of a Relational one, our translation is as follows:

1. Every rooted *Set* generates a table with the attributes of the corresponding *Structs* inside it. The ICs guarantee that all *Structs* in a *Set* have the same meaning and identifier. If attributes of these *Structs* do not coincide, the counterparts in the table would simply accept NULL values.
2. Every table will have a Primary Key (PK), generated according to the *Anchor* of the corresponding *Structs*. Concretely, it will be composed by the identifiers of its loose ends. For example, if the *Anchor* contains only one *Association*, its *AssociationEnds* will be the PK or the table. However, if the *Anchor* contains two connected *Associations*, the PK will be composed by the two disconnected *AssociationEnds* (i.e., the loose ends). This generalizes to more complex subgraphs in the *Anchor*.
3. Every *Identifier* can potentially give rise to a Foreign Key (FK). If it is a loose end in a *Struct* or its *Class* has *Superclasses*, we point to tables corresponding to rooted *Sets* where that *Class* (or any of its *Superclasses*) appears as the *Anchor* of the *Structs* inside.

In case of a schemaless system, translation is much easier, because we do not need to declare the attributes and typically they do not provide FK declaration mechanisms. For example, we can generate a denormalized translation into an RDBMS using JSON as follows:

1. Every rooted *Set* generates a table with two attributes: an autoincrement key, and a JSON value.
2. Each one of those tables will have a unique index similarly to PKs in the Relational case.

In any case, once the *Schema* has been created, we need to migrate the data from the old to the new one. For this to be possible, both *Domains* must coincide, $Dom(\mathbb{H}_{old}) = Dom(\mathbb{H}_{new})$. Given that, just realize that migrating data boils down to simply:

1. Find R as the set of all rooted *Sets* (i.e., collection/table) in the new *Design*.
2. For every rooted *Set* $s_i \in R$, define a migration DORM query M_i with DE_i and A_i as the *Domain* concepts (respectively the *DomainEdges* and *Attributes*) contained in s_i, either directly or transitively.

```
{ "pattern": [$DE_i$], "project": [$A_i$] }
```

3. Translate every M_i into a query Q_i over the corresponding DBMS, considering the *Design* of the old one, $M_i \xmapsto{\mathbb{H}_{old}} Q_i$ (see Algorithm 1 explained in Section 4.3).
4. Execute every Q_i over the *Schema* corresponding to $\mathbb{H}_{old}$.
5. Insert the output data into the *Schema* corresponding to $\mathbb{H}_{new}$.

Hence, migration is just reduced to querying as explained in the next section.

4.3. Data Manipulation

The sharp-eyed reader would have realized that what we have been describing above is a classical data integration mapping mechanism, where the *Domain* corresponds to "global", the *Schema* to "local", and the *Design* maps them. Thus, we have defined the *Schemas* as in Local-As-View (LAV), and what we are looking at now is query translation in such systems [48].

Specifically, we use a recursive algorithm (Algorithm 1) based on the creation of buckets as in [56], see Algorithm 1. Its input is a Select-Project-Join (SPJ) query in the form of a JSON document including: *pattern* determining the involved *Classes* and *Associations*, *project* containing the requested *Attributes*, and *filter* giving the selection predicate. Notice that grouping and aggregating could be easily added by just including them in the input JSON document and translating the attributes involved like the ones used in the filter. The algorithm starts by parsing that specification and extracting the different parts (Line 2). This also checks that all *Attributes* in *project* and *filter*, as well as *Classes* and *Associations* in *pattern* belong to the *Domain* and conform to a connected graph there.

Once we have that validated, we need to check (Line 3) if all *Classes* required in the query are actually in the *Design*. A *Class* may be implicit if its subclasses have been used instead. If this is the case, we find a *Complete Generalization* of one of such *Classes* (Line 4) and get its subclasses (Line 5). Then, we iterate these and make a recursive call (Line 9) to generate a subquery, replacing in the specification the superclass with the corresponding subclass. Once we have all the subqueries resolved, since each may

Algorithm 1 GenerateQuery($spec$)

```
Require: $spec$: SPJ query specification
Ensure: $sts$: list of query statements.
 1: $sts \leftarrow []$
 2: $projAtt, edges, filter, filterAtt \leftarrow parseQuery(spec)$
 3: if $implicitClass(projAtt \cup filterAtt, edges)$ then
 4:     $superclass \leftarrow getSuperclass(projAtt \cup filterAtt, edges)$
 5:     $subclasses \leftarrow getSubclasses(superclass)$
 6:     $subqueries \leftarrow []$
 7:     for $s \in subclasses$ do
 8:         $newSpec \leftarrow spec.replaceEdge(superclass, subclass)$
 9:         $subqueries.append(GenerateQuery(newSpec))$
10:     end for
11:     if disjoint(subclasses) then
12:         $sts \leftarrow combine(subqueries, “UNION\ ALL”)$
13:     else
14:         $sts \leftarrow combine(subqueries, “UNION”)$
15:     end if
16: else                                              ▷ All classes appear in the design
17:     $alts \leftarrow combineBuckets(projAttr \cup filterAttr, edges)$
18:     for $rootedSet \in alts$ do
19:         $alias \leftarrow getAlias(rootedSets)$
20:         $discr \leftarrow getDiscriminants(rootedSets, getClasses(edges))$
21:         $q \leftarrow generateSELECT(projAttrs, alias)$
22:         $q.concat(generateJOINS(rootedSets, edges, alias))$
23:         $q.concat(generateWHERE(filter, discr, alias))$
24:         $sts.append(q)$
25:     end for
26: end if
27: return $sts$
```

have generated more than one statement, we produce all their combinations. Such combinations use “`UNION ALL`” (Line 12) or “`UNION`” (Line 14) depending on the *Generalization* having been declared as “Disjoint” or not in the *Domain*. Notice that it might also happen that none of the *Generalizations* of the *Superclass* had been declared as *Complete*. If that was the case, we take all subclasses (Line 5) from different *Generalizations* and combine all the corresponding subqueries with “UNION”.

Once we know that all *Classes* are explicitly in the *Design* (Line 16), we can identify the collections/tables where they are stored. Thus, for each *DomainEdge*, we generate a “bucket” with all the collections/tables (i.e., rooted *Sets*) where it is transitively included. Then, we generate minimal combinations of the elements of such “buckets”, by taking one container per “bucket” until they cover all required concepts and attributes (Line 17). For each of those combinations (Line 18), we end up generating a query in the result (Line 24) that considers aliases, discriminants (to distinguish subclasses), attribute projections, joins, and filters. Finally, the algorithm returns all those queries (Line 27).

Example 3. *Consider the following query in terms of Design concepts, encoded in our JSON-based syntax:*

```
{ "pattern": ["SpecObjAll"], "project": ["specobjall_specobjid", "specobjall_img"],
  "filter": "specobjall_dec BETWEEN 0.199 AND 0.913 AND specobjall_zwarning=0" }
```

Let us suppose our Design corresponds to a hybrid (i.e., horizontal plus vertical) partition of `SpecObjAll`, corresponding to Figure 5e plus another Set with a Struct inside containing `SpecObjCompl`. Thus, the absence in such Design of the only requested concept `SpecObjAll` would be detected in Line 3. The concrete required Superclass would be identified based on the requested Attributes in Line 4 (i.e., `SpecObjAll`, which includes all the requested Attributes), and the corresponding Subclasses in Line 5 (i.e., corresponding to the two horizontal partitions `SpecObj` and `SpecObjCompl`). Hence, the recursive call in Line 9 will generate a query for each of those Subclasses that will be eventually merged in a single one in Line 12 by means of “UNION ALL” (given that the Generalization was declared as disjoint).

Assume that the first of those recursive calls is done to replace `SpecObjAll` by `SpecObj`. The re-

quested Class is explicit, so we jump to Line 17, which generates a "bucket" of containers for each one of the four Attributes (two in the projection and two in the filter). The "bucket" of `specobjall_img` contains `SpecObjRare_table` (as it is an Attribute rarely requested), the one of `specobjall_dec` and `specobjall_zwarning` contains `SpecObjCommon_table` (as they are much more common), and the one of `specobjall_specobjid` contains both (as this identifier appears in both vertical partitions). Then, the only combination of containers that covers all four Attributes is the one with both `SpecObjCommon_table` and `SpecObjRare_table`. Aliases are generated in Line 19, and Line 20 detects that in this Design there is no need to use the Discriminant, as the two involved containers do not mix instances of different subclasses. The next lines compose the corresponding query with all that information.

The second recursive call over `SpecObjCompl` will be treated similarly, but it is simpler, because the absence of vertical partition in this case does not require any join. Therefore, assuming a Relational implementation in $1NF$, the whole process would produce the following SQL query:

```
SELECT t_1.specobjall_specobjid, t_2.specobjall_img FROM SpecObjCommon_table t_1
    JOIN SpecObjRare_table t_2 ON t_1.specobjall_specobjid = t_2.specobjall_specobjid
WHERE t_1.specobjall_dec BETWEEN 0.199 AND 0.913 AND t_1.specobjall_zwarning=0
UNION ALL
SELECT specobjall_specobjid, specobjall_img FROM SpecObjCompl_table
WHERE specobjall_dec BETWEEN 0.199 AND 0.913 AND specobjall_zwarning=0;
```

Data can not only be queried, but also modified - and in a much simpler way (see Section 4.4). DORM identifies what needs to be modified in the database given a modification specification in terms of the *Domain* concepts by just using the same "combineBuckets" call in Line 17 of Algorithm 1. Then, the corresponding statement aligning the data provided and the current schema is generated.

Example 4. *A modification specification can also be specified over more than one concept at once, as soon as they are all in the same rooted Set in the Design (see Section 4.4) as in Figures 5f and 5j.*

```
{ "pattern": ["PhotoObjAll", "SpecObjAll", "SpecObjAll_PhotoObjAll"],
  "data": ["photoobjall_objid": "1237648720716235050", "specobjall_specobjid": "352457551322834940
      ", ...] }
```

For example, if used together with the Design in Figure 5f, this would generate:

```
INSERT INTO SpecObjAll_table (specobjall_specobjid, photoobjall_objid, ...) VALUES
    ("352457551322834940", "1237648720716235050", ...);
```

4.4. Limitations

Using views to query databases is clearly a powerful tool to hide query complexity from users. Nevertheless, the issues of modifying the database through them are also well known since long ago [57, 58]. Thus, this is not surprisingly the main limitation of our approach, too. The problem is that a modification specification, depending on the *Design*, might result in more than one possible data modification, making it ambiguous. We acknowledge the complexity of this problem and, despite recent research in the field [59], take a very conservative approach. Thus, any modification request requiring the modification of more than one rooted *Set* or not including all *Identifiers* determined by its *Anchor* is rejected by DORM. Indeed, current DBMSs like Oracle and PostgreSQL take a similar approach and allow modifying tables through views only if the view is defined over a single table, and its PKs are provided.

To mitigate the problem and prevent this limitation from interfering with the functionalities of the client application of DORM, we annotate $\mathbb{H}$ with a set of *Guards* $\mathbb{G}$ associated to the *Domain*. A *Guard* is a set of *Attributes* plus a set of *DomainEdges* (i.e., one of our SPJ queries without any *filter*). Then, at *Design* creation time, all *Guards* in the *Domain* are tested to guarantee that the corresponding modifications will be possible later on from the application. If any of them is not, the *Design* is rejected.

1NF	Concept Representation								Workload	
Design	Association	PhotoObjAll	PhotoObj	PhotoPrimary	Photoz	SpecObjAll	SpecObj	Migration	May	June
Baseline	FK	Table			Table	Table		-	412	98
D1	FK	Horizontal P.	Horizontal P.	Vertical P.	Join Mat.	Table		39972	48	72
D2	Table	Horizontal P.	Horizontal P.	Vertical P.	Join Mat.	Table		30568	44	61
D3	FK	Table			Join Mat.	Horizontal P.	Vertical P.	17096	396	92
D4	Table	Table			Join Mat.	Horizontal P.	Vertical P.	14645	405	91
D5	FK++	Horizontal P.	Horizontal P.	Table	Join Mat.	Horizontal P.	Vertical P.	25387	109	60
D6	Table	Horizontal P.	Horizontal P.	Table	Join Mat.	Horizontal P.	Vertical P.	32269	114	87

Figure 6: Characteristics of the considered *Designs* in $1NF$

5. Experiments

To empirically demonstrate our point from a computing perspective, we analyzed in detail the real effect and benefit of schema evolution on an extract of the SkyServer workload. For this, we considered the average workload during the months of May and June 2025. Firstly, in Section 5.1, we explain how we characterized the different workloads identifying the most common query patterns (i.e., ignoring mostly unique queries in the very long tail of frequencies). Then, in Section 5.2, we detail the multiple *Designs* of our sample of SkyServer database that we easily tested with DORM and compare their performance according to both workloads. DORM automatically migrates the data from the baseline schema to each one of the others and rewrites the frequent query patterns previously identified and declared just in terms of the *Domain* concepts. Finally, in Section 5.3, we analyze both the cost of queries in each *Schema* as well as the cost of moving the data from one to another, to see how beneficial each movement is.

DORM is implemented in approximately 3,000 lines of Python (v3.12) code.[6] Tests have been executed on a PostgreSQL (v17) instance, running on a server with an i7-8700 CPU and 64 GB of RAM. For reproducibility, the code is publicly available in GitHub,[6] including a convenient running example.[7]

5.1. Workload extraction

We generated the workload for the experiments as follows:

1. **Fetched the data**: Connecting to SkyServer and retrieving log entries[2] corresponding to the given period, which finished correctly, elapsed some time, and returned some rows (i.e., "`error = 0 AND elapsed > 0 AND rows > 0`").
2. **Parsed the retrieved queries**: Identifying the different parts of the queries (i.e., projected attributes, selection predicate, tables involved, and other characteristics like distinct or top) and discarding those: a) Corresponding to DDL; b) Accessing the database catalog; c) Using user-defined materialized views; and d) Using stored procedures in the FROM clause (those in the WHERE clause were simply replaced by query parameters).
3. **Generated patterns**: Grouping queries that use the same tables in the FROM clause and have the same characteristics (like distinct or top) and whose projected attributes are above a given threshold of Jaccard distance (i.e., they share a minimum percentage of projected attributes, set to 80%). We kept only patterns that appeared more than 2%, and generated an aggregated filter clause as the intersection of all clauses in the group.

Thus, we generated two different workloads with the characteristics in Table 2. The frequencies of query patterns are respectively [33%, 12%, 10%, 7%, 6%, 2%] and [40%, 16%, 9%, 5%, 5%, 4%, 4%, 4%], which clearly reflect their dissimilarity. As an example, the most frequent (33%) pattern in May is:

```sql
SELECT objid, petroRad_r FROM PhotoObj;
```

Which we encode in JSON-based syntax like:

[6] https://github.com/alberto0723/DORM
[7] https://github.com/alberto0723/DORM/blob/main/DOLAP_running_example.md

Table 2
Extraction summary

Month	# Queries			# Patterns
	Discarded	Kept	Total	
May 2025	3,954,198	2,191,734	6,145,932	6
June 2025	5,178,312	5,806,285	10,984,597	8

```
{ "pattern": ["PhotoObj"], "project": ["photoobjall_objid", "photoobjall_petrorad_r"] }
```

On the other hand, the most frequent one (40%) in June is:

```sql
SELECT s.run2d, s.plate, s.mjd, s.fiberID
FROM PhotoObjAll AS p JOIN SpecObjAll AS s ON p.objID = s.bestObjID;
```

Which we encode in JSON-based syntax like (notice that we do not encode join attributes, as we assume they correspond to relationships already expressed in the *Domain*):

```
{ "pattern": ["PhotoObjAll", "SpecObjAll", "SpecObjAll_PhotoObjAll"],
  "project": ["specobjall_fiberid", "specobjall_mjd", "specobjall_plate", "specobjall_run2d"] }
```

5.2. Alternative *Designs*

Given the *Domain* information in Figure 4, we took as baseline the original design in SkyServer as in Figure 2. This is roughly a simple one-to-one mapping between classes and tables (i.e., three tables overall for `PhotoObjAll`, `SpecObjAll`, and `Photoz`), just adding one discriminant attribute in the superclass for every *Generalization* (namely $G1$ to $G4$) encoding the corresponding subclass of each row. The association is also implemented as an FK. Then, we implemented different combinations of the following patterns in Figure 5 applied to those classes:

Intermediate table (Figure 5a): The association between `PhotoObjAll` and `SpecObjAll` with an intermediate table (*Designs* D2, D4, and D6).

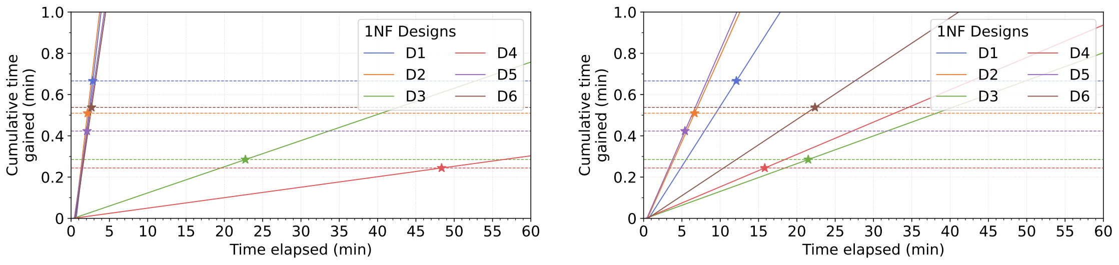
Foreign key (Figure 5b): The association between `PhotoObjAll` and `SpecObjAll` with an FK in the table corresponding to `SpecObjAll` (*Designs* D1, D3, and D5).

Horizontal partition (Figure 5d): Two different horizontal partitions accommodating subclasses of `PhotoObjAll` and `SpecObjAll` in different tables (*Designs* D1, D2, D5, and D6 for `PhotoObjAll`; and D3, D4, D5, and D6 for `SpecObjAll`).

Vertical partition (Figure 5e): Two vertical partitions separating the attributes most used in the queries over `PhotoObjAll` (*Designs* D1, and D2) and `SpecObjAll` (*Designs* D3, D4, D5, and D6).

Figure 6 summarizes the *Designs* implemented in First Normal Form ($1NF$). We did not aim at finding the optimum database schema, but just to showcase the flexibility and possibilities of DORM. Nevertheless, just to guarantee some performance impact, we paid attention to the two sets or query patterns to choose these *Designs*. For example, we avoided redundancy as much as possible, but in the case of `Photoz`, we decided to always materialize its join with `PhotoObjAll`. Also, it is worth noting that in D5, together with the FK in `SpecObjAll`, we stored the discriminants of `PhotoObjAll` (indicated as "FK++" in the table), aiming to improve one of the query patterns in June's workload.

For each one of the patterns, we see the overall time to populate it with the data in the baseline, together with the weighted average of the query execution time of both workloads. These times were directly obtained from PostgreSQL by means of "`EXPLAIN (ANALYZE, SUMMARY)`", and are expressed in milliseconds. The colors indicate the ranking of the different *Designs* for the corresponding workload (red for higher and green for lower values). Thus, we can see that not only the average cost per query changes depending on the month, but also the relative positions of the *Designs*. In both cases, the worst is the baseline, but the best *Design* for May would be D2, while the best one for June would be D5.



(a) May (b) June

Figure 7: Tradeoff between migration costs (dotted lines) and query benefits (oblique lines) per new $1NF$ *Design*

Changing just one parameter of DORM (no redesign being necessary at all), it switches to a Non-First Normal Form (NF^2) implementation with JSON, which enables multiple design possibilities nesting *Sets* and *Structs*. Nevertheless, given the hundreds of attributes in our use case, extracting them from the documents is too expensive, and performance is two orders of magnitude worse than that in 1NF in all the cases. Thus, due to lack of space, we omitted their execution times.

5.3. Benefit quantification

Figures 7a and 7b showcase the cost-benefit analysis of migrating from the baseline to the new *Designs* in $1NF$ (Figure 6), respectively in May (a) and June (b). For each *Design* (indicated by the different colors), the dotted horizontal line is a threshold indicating the migration time. The solid oblique line indicates the cumulative benefit eventually gained (i.e., the improvement in the average query execution time with respect to the baseline) weighted by the frequency of queries in the workload ($0.85\ query/sec$ in May, $2.24\ query/sec$ in June). The star marks the turning point of a *Design* (i.e., after how much time the migration to the new *Design* becomes "profitable", in that the cumulative benefit has compensated the migration time). Execution times for both migration and workload queries correspond to those in Section 5.2. Interestingly, among the tested *Designs*, the "cheapest" ones with regard to migration time (D3 and D4, respectively in green and red) are also the least convenient, while D5 (in purple) dominates as the one becoming "profitable" in the shortest time (for both workloads).

This analysis[8] demonstrates how the tradeoff between migration costs and query benefits varies significantly between different *Designs*, thus supporting our motivation in proposing a system to automate the evaluation and potential adoption of multiple *Design* alternatives.

6. Conclusions

We have presented DORM, a system to automate database schema evolution management. It uses catalog information to define the concepts (i.e., classes, attributes, and relationships) in the application domain. These can be used later to define multiple database designs in an abstract way (in terms of *Sets* and *Structs*), which are automatically translated into the underlying DBMS-specific data structures (e.g., tables if it is Relational, or collections if it is a document store). DORM not only automatically generates the corresponding data definition statements, but also the data management statements to migrate the data from one *Schema* to another. Queries are also defined in an abstract syntax, and automatically translated into the specific statements depending on the data organization indicated by the *Design*, so that users are not affected by changes. Extensive experiments demonstrate the use of our system and show the immediate benefit of schema evolution in performance in a use case with data from SDSS.

As future work, we plan to extend DORM with automatic *Design* generation and a search mechanism to automatically optimize the benefit of schema evolution. This should also consider the possibility of generating redundant structures (e.g., materialized views), which increases the search space.

[8]The analysis is limited to $1NF$ *Designs* as all the NF^2 ones generate a significant performance drop with respect to the 1NF baseline, making the migration clearly inconvenient.

Acknowledgments

This work was supported by the project PID2024-156019OB-I00 under funding schema MICI-U/AEI/10.13039/501100011033 and ERFD/EU. We thank Rozhina Ahamadi and Otman Ezzayat Maid for their work on characterizing the SkyServer workload and translating Visual Paradigm diagrams into DORM syntax.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

Artifacts

The code of the prototype is publicly available at:
https://github.com/alberto0723/DORM
An explanation of the main components is in:
https://github.com/alberto0723/DORM/blob/main/README.md
A demo video (5 min) is in:
https://github.com/alberto0723/DORM/blob/main/DORMdemo.mp4
Finally, step by step instructions to reproduce the experiments are in:
https://github.com/alberto0723/DORM/blob/main/DOLAP_running_example.md

References

[1] A. Chebotko, A. Kashlev, S. Lu, A big data modeling methodology for apache cassandra, in: B. Carminati, L. Khan (Eds.), 2015 IEEE International Congress on Big Data, New York City, NY, USA, June 27 - July 2, 2015, IEEE Computer Society, 2015, pp. 238–245. URL: https://doi.org/10.1109/BigDataCongress.2015.41. doi:10.1109/BigDataCongress.2015.41.

[2] L. Chen, A. Davoudian, M. Liu, A workload-driven method for designing aggregate-oriented NoSQL databases, Data Knowl. Eng. 142 (2022) 102089. URL: https://doi.org/10.1016/j.datak.2022.102089. doi:10.1016/J.DATAK.2022.102089.

[3] M. Hewasinghage, S. Nadal, A. Abelló, E. Zimányi, Automated database design for document stores with multicriteria optimization, Knowl. Inf. Syst. 65 (2023) 3045–3078. URL: https://doi.org/10.1007/s10115-023-01828-3. doi:10.1007/s10115-023-01828-3.

[4] E. Gallinucci, M. Golfarelli, W. Radwan, G. Zarate, A. Abelló, Impact study of nosql refactoring in skyserver database, in: Proc. of the 27th Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP 2025), volume 3931 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2025, pp. 1–11. URL: https://ceur-ws.org/Vol-3931/paper1.pdf.

[5] L. Bellatreche, Y. A. Ameur, C. Chakroun, A design methodology of ontology based database applications, Log. J. IGPL 19 (2011) 648–665. URL: https://doi.org/10.1093/jigpal/jzq017. doi:10.1093/JIGPAL/JZQ017.

[6] A. Deshpande, Beyond relations: A case for elevating to the entity-relationship abstraction, in: Proc. 15th Annual Conference on Innovative Data Systems Research (CIDR 2025), 2025. URL: https://arxiv.org/abs/2505.03536.

[7] J. Dittrich, How to get rid of sql, relational algebra, the relational model, erm, and orms in a single paper – a thought experiment, 2025. URL: https://arxiv.org/abs/2504.12953. arXiv:2504.12953.

[8] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, G. J. Gordon, Query-based workload forecasting for self-driving database management systems, in: G. Das, C. M. Jermaine, P. A. Bernstein (Eds.), Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, ACM, 2018, pp. 631–645. URL: https://doi.org/10.1145/3183713.3196908. doi:10.1145/3183713.3196908.

[9] V. V. Meduri, K. Chowdhury, M. Sarwat, Evaluation of machine learning algorithms in predicting the next SQL query from the future, ACM Trans. Database Syst. 46 (2021) 4:1–4:46. URL: https://doi.org/10.1145/3442338. doi:10.1145/3442338.

[10] S. Chakkappen, S. Kunjibettu, D. Mcgreer, M. Kishi, H. Su, M. Ziauddin, M. Zaït, Z. Li, Y. Zhang, Automatic indexing in oracle, Proc. VLDB Endow. 18 (2025) 4924–4937. URL: https://www.vldb.org/pvldb/vol18/p4924-chakkappen.pdf. doi:10.14778/3750601.3750616.

[11] P. Atzeni, F. Bugiotti, L. Rossi, Uniform access to nosql systems, Inf. Syst. 43 (2014) 117–133. URL: https://doi.org/10.1016/j.is.2013.05.002. doi:10.1016/J.IS.2013.05.002.

[12] P. J. Sadalage, M. Fowler, NoSQL distilled: a brief guide to the emerging world of polyglot persistence, Pearson Education, 2013.

[13] A. Meier, M. Kaufmann, SQL & NoSQL databases, Springer, 2019.

[14] C. de Lima, R. dos Santos Mello, On proposing and evaluating a NoSQL document database logical approach, Int. J. Web Inf. Syst. 12 (2016) 398–417. URL: https://doi.org/10.1108/IJWIS-04-2016-0018. doi:10.1108/IJWIS-04-2016-0018.

[15] A. Jindal, H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, S. Krishnan, Peregrine: Workload optimization for cloud query engines, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019, ACM, 2019, pp. 416–427. URL: https://doi.org/10.1145/3357223.3362726. doi:10.1145/3357223.3362726.

[16] A. de la Vega, D. García-Saiz, C. Blanco, M. E. Zorrilla, P. Sánchez, Mortadelo: Automatic generation of NoSQL stores from platform-independent data models, Future Gener. Comput. Syst. 105 (2020) 455–474. URL: https://doi.org/10.1016/j.future.2019.11.032. doi:10.1016/j.future.2019.11.032.

[17] J. Mali, F. Atigui, A. Azough, N. Travers, Modeldrivenguide: An approach for implementing NoSQL schemas, in: S. Hartmann, J. Küng, G. Kotsis, A. M. Tjoa, I. Khalil (Eds.), Database and Expert Systems Applications - 31st International Conference, DEXA 2020, Bratislava, Slovakia, September 14-17, 2020, Proceedings, Part I, volume 12391 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 141–151. URL: https://doi.org/10.1007/978-3-030-59003-1_9. doi:10.1007/978-3-030-59003-1_9.

[18] N. Roy-Hubara, A. Sturm, P. Shoval, Designing NoSQL databases based on multiple requirement views, Data Knowl. Eng. 145 (2023) 102149. URL: https://doi.org/10.1016/j.datak.2023.102149. doi:10.1016/j.datak.2023.102149.

[19] F. Abdelhédi, A. A. Brahim, F. Atigui, G. Zurfluh, UMLtoNoSQL: Automatic transformation of conceptual schema to NoSQL databases, in: 14th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2017, Hammamet, Tunisia, October 30 - Nov. 3, 2017, IEEE Computer Society, 2017, pp. 272–279. URL: https://doi.org/10.1109/AICCSA.2017.76. doi:10.1109/AICCSA.2017.76.

[20] M. J. Mior, K. Salem, A. Aboulnaga, R. Liu, NoSE: Schema design for NoSQL applications, IEEE Trans. Knowl. Data Eng. 29 (2017) 2275–2289. URL: https://doi.org/10.1109/TKDE.2017.2722412. doi:10.1109/TKDE.2017.2722412.

[21] J. Kossmann, R. Schlosser, A framework for self-managing database systems, in: 35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019, IEEE, 2019, pp. 100–106. URL: https://doi.org/10.1109/ICDEW.2019.00-27. doi:10.1109/ICDEW.2019.00-27.

[22] W. Y. Mok, A conceptual model based design methodology for mongodb databases, in: 7th International Conference on Information and Computer Technologies, ICICT 2024, Honolulu, HI, USA, March 15-17, 2024, IEEE, 2024, pp. 151–159. URL: https://doi.org/10.1109/ICICT62343.2024.00030. doi:10.1109/ICICT62343.2024.00030.

[23] M. Mozaffari, E. Nazemi, A. Eftekhari-Moghadam, CONST: continuous online NoSQL schema tuning, Softw. Pract. Exp. 51 (2021) 1147–1169. URL: https://doi.org/10.1002/spe.2945. doi:10.1002/SPE.2945.

[24] V. Reniers, D. V. Landuyt, A. Rafique, W. Joosen, A workload-driven document database schema recommender (DBSR), in: G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, H. C. Mayr (Eds.), Con-

ceptual Modeling - 39th International Conference, ER 2020, Vienna, Austria, November 3-6, 2020, Proceedings, volume 12400 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 471–484. URL: https://doi.org/10.1007/978-3-030-62522-1_35. doi:10.1007/978-3-030-62522-1_35.

[25] E. M. Kuszera, L. M. Peres, M. D. D. Fabro, Exploring data structure alternatives in the RDB to NoSQL document store conversion process, Inf. Syst. 105 (2022) 101941. URL: https://doi.org/10.1016/j.is.2021.101941. doi:10.1016/j.is.2021.101941.

[26] I. Skoulis, P. Vassiliadis, A. V. Zarras, Growing up with stability: How open-source relational databases evolve, Inf. Syst. 53 (2015) 363–385. URL: https://doi.org/10.1016/j.is.2015.03.009. doi:10.1016/j.is.2015.03.009.

[27] D. Sjøberg, Quantifying schema evolution, Inf. Softw. Technol. 35 (1993) 35–44. URL: https://doi.org/10.1016/0950-5849(93)90027-Z. doi:10.1016/0950-5849(93)90027-Z.

[28] P. Manousis, P. Vassiliadis, A. V. Zarras, G. Papastefanatos, Schema evolution for databases and data warehouses, in: E. Zimányi, A. Abelló (Eds.), Business Intelligence - 5th European Summer School, eBISS 2015, Barcelona, Spain, July 5-10, 2015, Tutorial Lectures, volume 253 of *Lecture Notes in Business Information Processing*, Springer, 2015, pp. 1–31. URL: https://doi.org/10.1007/978-3-319-39243-1_1. doi:10.1007/978-3-319-39243-1_1.

[29] S. Wu, I. Neamtiu, Schema evolution analysis for embedded databases, in: S. Abiteboul, K. Böhm, C. Koch, K. Tan (Eds.), Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, IEEE Computer Society, 2011, pp. 151–156. URL: https://doi.org/10.1109/ICDEW.2011.5767627. doi:10.1109/ICDEW.2011.5767627.

[30] S. Scherzinger, S. Sidortschuck, An empirical study on the design and evolution of NoSQL database schemas, in: G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, H. C. Mayr (Eds.), Conceptual Modeling - 39th International Conference, ER 2020, Vienna, Austria, November 3-6, 2020, Proceedings, volume 12400 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 441–455. URL: https://doi.org/10.1007/978-3-030-62522-1_33. doi:10.1007/978-3-030-62522-1_33.

[31] S. Fedushko, R. Malyi, Y. Syerov, P. Serdyuk, NoSQL document data migration strategy in the context of schema evolution, Data & Knowledge Engineering 154 (2024) 102369. URL: https://www.sciencedirect.com/science/article/pii/S0169023X24000934. doi:https://doi.org/10.1016/j.datak.2024.102369.

[32] C. Curino, H. J. Moon, L. Tanca, C. Zaniolo, Schema evolution in wikipedia - toward a web information system benchmark, in: J. Cordeiro, J. Filipe (Eds.), ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume DISI, Barcelona, Spain, June 12-16, 2008, 2008, pp. 323–332.

[33] P. Vassiliadis, Profiles of schema evolution in free open source software projects, in: 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021, IEEE, 2021, pp. 1–12. URL: https://doi.org/10.1109/ICDE51399.2021.00008. doi:10.1109/ICDE51399.2021.00008.

[34] B. A. Muse, F. Khomh, G. Antoniol, Refactoring practices in the context of data-intensive systems, Empir. Softw. Eng. 28 (2023) 46. URL: https://doi.org/10.1007/s10664-022-10271-x. doi:10.1007/s10664-022-10271-x.

[35] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, W. Lehner, Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language, in: S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, D. Suciu (Eds.), Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, ACM, 2017, pp. 1101–1116. URL: https://doi.org/10.1145/3035918.3064046. doi:10.1145/3035918.3064046.

[36] P. Koupil, J. Bártík, I. Holubová, *MM-evocat:* A tool for modelling and evolution management of multi-model data, in: M. A. Hasan, L. Xiong (Eds.), Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022, ACM, 2022, pp. 4892–4896. URL: https://doi.org/10.1145/3511808.3557180. doi:10.1145/3511808.3557180.

[37] I. Holubová, M. Vavrek, S. Scherzinger, Evolution management in multi-model databases, Data Knowl. Eng. 136 (2021) 101932. URL: https://doi.org/10.1016/j.datak.2021.101932. doi:10.1016/j.

datak.2021.101932.

[38] A. H. Chillón, D. S. Ruiz, J. G. Molina, Towards a taxonomy of schema changes for NoSQL databases: The orion language, in: A. K. Ghose, J. Horkoff, V. E. S. Souza, J. Parsons, J. Evermann (Eds.), Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings, volume 13011 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 176–185. URL: https://doi.org/10.1007/978-3-030-89022-3_15. doi:10.1007/978-3-030-89022-3_15.

[39] L. Caruccio, G. Polese, G. Tortora, Synchronization of queries and views upon schema evolutions: A survey, ACM Trans. Database Syst. 41 (2016) 9:1–9:41. URL: https://doi.org/10.1145/2903726. doi:10.1145/2903726.

[40] A. Hillenbrand, M. Levchenko, U. Störl, S. Scherzinger, M. Klettke, Migcast: Putting a price tag on data model evolution in NoSQL data stores, in: P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, T. Kraska (Eds.), Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, ACM, 2019, pp. 1925–1928. URL: https://doi.org/10.1145/3299869.3320223. doi:10.1145/3299869.3320223.

[41] P. Benats, L. Meurice, M. Gobert, A. Cleve, Query-based schema evolution recommendations for hybrid polystores, in: S. Link, I. Reinhartz-Berger, J. Zdravkovic, D. Bork, S. Srinivasa (Eds.), Proceedings of the ER Forum and PhD Symposium 2022 co-located with 41st International Conference on Conceptual Modeling (ER 2022), Virtual Event, Hyderabad, India, October 17, 2022, volume 3211 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3211/CR_122.pdf.

[42] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, T. Zhang, Self-driving database management systems, in: 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings, www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf.

[43] R. Tan, R. Chirkova, V. Gadepally, T. G. Mattson, Enabling query processing across heterogeneous data models: A survey, in: 2017 IEEE International Conference on Big Data (Big Data), IEEE, 2017, pp. 3211–3220.

[44] C. Forresi, E. Gallinucci, M. Golfarelli, H. B. Hamadou, A dataspace-based framework for OLAP analyses in a high-variety multistore, VLDB J. 30 (2021) 1017–1040. URL: https://doi.org/10.1007/s00778-021-00682-5. doi:10.1007/S00778-021-00682-5.

[45] C. Forresi, M. Francia, E. Gallinucci, M. Golfarelli, Cost-based optimization of multistore query plans, Information systems frontiers 25 (2023) 1925–1951.

[46] L. El Ahdab, I. Megdiche, A. Péninou, O. Teste, Unified models and framework for querying distributed data across polystores, in: International Conference on Research Challenges in Information Science, Springer, 2024, pp. 3–18.

[47] C. Beeri, A. Y. Levy, M.-C. Rousset, Rewriting queries using views in description logics, in: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '97, Association for Computing Machinery, New York, NY, USA, 1997, p. 99–108. URL: https://doi.org/10.1145/263661.263673. doi:10.1145/263661.263673.

[48] M. Lenzerini, Data integration: a theoretical perspective, in: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, Association for Computing Machinery, New York, NY, USA, 2002, p. 233–246. URL: https://doi.org/10.1145/543613.543644. doi:10.1145/543613.543644.

[49] M. Masmoudi, S. Ben Abdallah Ben Lamine, M. H. Karray, B. Archimede, H. Baazaoui Zghal, Semantic data integration and querying: a survey and challenges, ACM Computing Surveys 56 (2024) 1–35.

[50] I. M. Putrama, P. Martinek, Heterogeneous data integration: Challenges and opportunities, Data in Brief 56 (2024) 110853.

[51] A. Y. Halevy, Answering queries using views: A survey, The VLDB Journal 10 (2001) 270–294. URL: https://doi.org/10.1007/s007780100054. doi:10.1007/s007780100054.

[52] F. Goasdoué, M.-C. Rousset, Answering queries using views: A krdb perspective for the semantic web, ACM Trans. Internet Technol. 4 (2004) 255–288. URL: https://doi.org/10.1145/1013202.1013204.

doi:10.1145/1013202.1013204.

[53] J.-F. Baget, M. Leclère, M.-L. Mugnier, S. Rocher, C. Sipieter, Graal: A toolkit for query answering with existential rules, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Springer International Publishing, Cham, 2015, pp. 328–344.

[54] S. Nadal, A. Abelló, O. Romero, S. Vansummeren, P. Vassiliadis, Graph-driven federated data management, IEEE Trans. Knowl. Data Eng. 35 (2023) 509–520. URL: https://doi.org/10.1109/TKDE.2021.3077044. doi:10.1109/TKDE.2021.3077044.

[55] M. Hewasinghage, A. Abelló, J. Varga, E. Zimányi, Managing polyglot systems metadata with hypergraphs, Data Knowl. Eng. 134 (2021) 101896. URL: https://doi.org/10.1016/j.datak.2021.101896. doi:10.1016/j.datak.2021.101896.

[56] A. Y. Levy, A. Rajaraman, J. J. Ordille, Querying heterogeneous information sources using source descriptions, in: Proceedings of the 22th International Conference on Very Large Data Bases, VLDB '96, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996, p. 251–262.

[57] F. Bancilhon, N. Spyratos, Update semantics of relational views, ACM Trans. Database Syst. 6 (1981) 557–575. URL: https://doi.org/10.1145/319628.319634. doi:10.1145/319628.319634.

[58] U. Dayal, P. A. Bernstein, On the correct translation of update operations on relational views, ACM Trans. Database Syst. 7 (1982) 381–416. URL: https://doi.org/10.1145/319732.319740. doi:10.1145/319732.319740.

[59] N. Makhija, W. Gatterbauer, Is integer linear programming all you need for deletion propagation? A unified and practical approach for generalized deletion propagation, Proc. VLDB Endow. 18 (2025) 2667–2680. URL: https://www.vldb.org/pvldb/vol18/p2667-makhija.pdf.