An Empirical Study of (Multi-) Database Models in Open-Source Projects

Pol Benats[1]✉, Maxime Gobert[1], Loup Meurice[1], Csaba Nagy[2], and Anthony Cleve[1]

[1] Namur Digital Institute, University of Namur, Belgium
[2] Software Institute, Università della Svizzera italiana, Switzerland
{firstname.lastname}@unamur.be - csaba.nagy@usi.ch

Abstract. Managing data-intensive systems has long been recognized as an expensive and error-prone process. This is mainly due to the often implicit consistency relationships that hold between applications and their database. As new technologies emerged for specialized purposes (e.g., graph databases, document stores), the joint use of database models has also become popular. There are undeniable benefits of such multi-database models where developers combine various technologies. However, the side effects on design, querying, and maintenance are not well-known yet. In this paper, we study multi-database models in software systems by mining major open-source repositories. We consider four years of history, from 2017 to 2020, of a total number of 40,609 projects with databases. Our results confirm the emergence of hybrid data-intensive systems as we found (multi-) database models (e.g., relational and non-relational) used together in 16% of all database-dependent projects. One percent of the systems added, deleted, or changed a database during the four years. The majority (62%) of these systems had a single database before becoming hybrid, and another significant part (19%) became "mono-database" after initially using multiple databases. We examine the evolution of these systems to understand the rationale of the design choices of the developers. Our study aims to guide future research towards new challenges posed by those emerging data management architectures.

Keywords: Data models · Open-source projects · Empirical study

1 Introduction

Modeling, querying, and evolving database-centered systems are known as time-consuming, risky, and error-prone. The main challenges related to those processes originate from the possibly complex interdependencies between the application programs and their underlying databases. This is especially true in the absence of a fully explicit database schema, which partly moves the responsibility of data integrity constraints from the database management system to the client programs. This situation may become even more complex with the increasing popularity of NoSQL database technologies. Such scalable technologies are attractive due to the flexibility offered by the absence of strict data schema. But the long-term impact of their use on data management still has to be assessed.

A. Ghose et al. (eds).: ER 2021, LNCS 13011, pp. 87–100, 2021.
The final publication is available at Springer via https://doi.org/10.1007/978-3-030-89022-3_8.

Furthermore, as their full name suggests, *Not Only SQL* technologies were not initially intended to *replace* relational database management systems but rather *complement* them. Hence there is a recent emergence of *hybrid* data-intensive systems that rely on both relational and NoSQL technologies. Such heterogeneous systems request that developers master several modeling and query languages, and bring new challenges for research and practice.

This paper presents an empirical study of the use of (multi-)database models in open-source database-dependent projects. We mined four years of development history (2017–2020) of a total number of 33 million projects by leveraging Libraries.io. We identified projects with databases and written in popular programming languages (1.3 million), then applied filters to eliminate "low-quality" repositories and remove project duplicates. We gathered a final dataset of 40,609 projects. We analyzed the dependencies of those projects to assess (1) the popularity of the different database models, (2) the extent that they are combined within the same systems, and (3) how their usage evolved.

Our results confirm the emergence of hybrid data-intensive systems as we found that 16% of all database-dependent projects use (multi-)database models (e.g., relational and non-relational).[3] We also observe that one percent of the systems added, deleted, or changed a database during the four years. The majority (62%) of them became hybrid after using a single database technology. On the other hand, a significant number of systems (19%) became "mono-database" after initially using multiple databases. We examined the evolution of these systems to understand the rationale of the design choices of the developers.

This study has several implications for the research community. It provides empirical evidence for the need for novel solutions supporting the design, querying, and evolution of hybrid database systems. It advances the identification of the most representative data-intensive systems for further empirical studies or evaluation purposes. It aims to guide future research towards new challenges posed by those emerging data management architectures.

The remainder of the paper is structured as follows. Section 2 expresses our research objectives through three research questions and presents the research method we followed in conducting our study. Section 3 presents our main findings. Section 4 proposes a discussion about the contribution of this paper to scientific research. Section 5 lists threats to validity, and Section 6 discusses related work. Concluding remarks are given in Section 7.

2 Study Method

Research questions We seek answers to the following research questions:

RQ1: How prevalent are different data models and their related database technologies? Based on the database dependencies of the collected open-source projects, we assess the usage of each database technology (*e.g.,* PostgreSQL, MongoDB, Redis, Neo4J, Cassandra) and, transitively, each data model (*i.e.,* relational, document, key-value, graph, and wide-column).

[3] The detailed results are publicly available in a replication package [3]

RQ2: Are multi-database models frequently used in software systems? We identify projects corresponding to *hybrid* systems, *i.e.,* systems using at least two data models. We analyze which data models are combined in those systems. We call those combined models *multi-database models*.

RQ3: How does data model usage evolve? We analyze the evolution of the database models used in repositories. We track the additions, deletions, or replacements of database models. We then classify the projects based on the evolution of their data models.

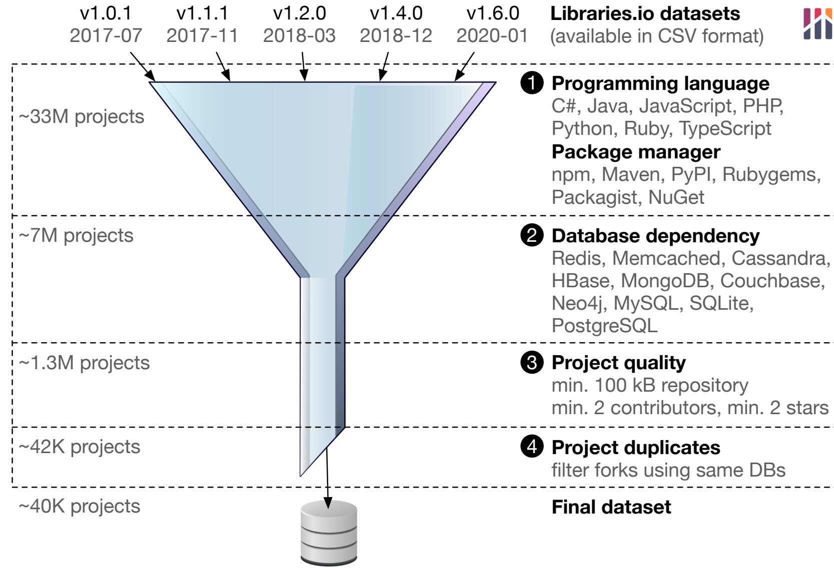


Fig. 1. Overview of project filtering

Preliminaries Our study took as input the dataset of *Libraries.io*,[4] an open-source web service that lists software development projects and their dependencies. The primary aim of this service is to help developers keeping track of libraries, modules, and frameworks they depend upon. However, its extensive and up-to-date dataset is also representative, thus valuable for empirical software engineering studies [10,29,30]. Libraries.io monitors projects of major source code repositories (*i.e., Bitbucket*, *GitHub*, and *GitLab*). It lists 33 million repositories with dependencies to 235 million packages in 37 package managers (*e.g., npm*, *Maven*, *PyPI*). Libraries.io has five major versions of its dataset at the time of writing, covering four years of evolution from 2017 to 2020. We imported each dataset into a database. Then, as shown in Figure 1, we filtered the projects according to the selection criteria described in the following subsections.

Programming language We study projects written in popular programming languages according to *Octoverse*,[5] namely *C#*, *Java*, *JavaScript*, *PHP*, *Python*, *Ruby*, and *TypeScript*. We determine the programming language of a project

[4] Libraries.io – `https://libraries.io/`
[5] Octoverse – `https://octoverse.github.com/`

based on the package manager it relies on. Hence, from the complete list of Octoverse, we excluded *C* and *C++* because their usage of package managers is not as common as in the selected languages. Package managers are important for us also when determining the database dependency in the subsequent filtering step. Libraries.io tracks the most popular managers for the selected programming languages, namely, *npm*, *Maven*, *PyPI*, *Rubygems*, *Packagist*, and *NuGet*. Therefore, besides the programming language, we keep projects having dependencies in these. In the remaining of the paper, we discuss JavaScript and TypeScript projects together due to the similarities of the two languages and, primarily, because of the same package managers and database libraries. The 2017 and 2020 versions of the dataset had 3.7M and 7.2M projects satisfying these criteria.

Database dependency The projects had to rely on database technologies. According to a recent survey [8], the most common NoSQL models are *key-value*, *wide-column*, *document-oriented*, and *graph-based* data models. We focus on the two most popular mono-database technologies for each data model on DB-Engines Ranking,[6] a monthly updated website that ranks database management systems. At the time of our study, the top *document* stores were *MongoDB* and *Couchbase*; *key-value* databases were *Redis* and *Memcached*; *wide-column* databases were *Apache Cassandra* and *HBase*. For *graph* databases, we only considered *Neo4j* as others had very weak rankings.

We added popular open-source *relational* database management systems, *i.e.,* *MySQL*, *SQLite*, and *PostgreSQL*. We excluded proprietary technologies such as *Oracle* and *SQL Server*. The reason is that our dataset represents open-source software systems where the use of proprietary databases is not representative. Consequently, we limit our study to open-source technologies. We established a list of search expressions combining database technologies and access types to identify a list of database access drivers. For direct access drivers (*e.g.,* JDBC connectors), we used *client* and *driver* keywords. For object-relational mapping (ORM) and object NoSQL mapping (ONM) [28] technologies, we used *orm*, *onm*, *odm*, *ogm* and *mapper* keywords. We searched for these expressions in the search engines of the package managers to find libraries. We also queried Google for additional libraries missed by the package managers. Each driver's relevance was manually checked based on its description and provided source code samples.

This collection process led to 707 direct database-access drivers, including 220 object mapping libraries. The complete list of selected drivers is available in our **replication package** [3]. Overall, 18% of the projects (710K in 2017 and 1.3M in 2020) depended on database-access libraries.

Project quality GitHub is known to host many private and inactive projects [16]. To ensure a representative sample, we filter "low-quality" projects from the list of repositories. In particular, we keep a project if it meets the following selection criteria: (1) a repository with a minimum size of 100kB, (2) at least two contributors, and (3) having been starred at least twice.

[6] DB-Engines Ranking – `https://db-engines.com/en/ranking`

Stars and contributors reflect a level of popularity [6] and collaborative activity [5]; thus, we set minimum thresholds for them. However, for us, a popular Python project with a single file of a few code lines is as important as a less popular Java project with thousands of lines of code, given that they use databases. The database dependency filter already ensures the latter one. So the purpose with the additional minimum size of 100kB is merely to expect a minimum content in repositories. A total number of 42,176 repositories (∼3%) remained in the dataset after this filtering step.

Project duplicates We filter duplicated projects (*i.e.,* exact copies of projects or projects with minor differences like bug fixes) by identifying fork projects. We keep only the source project of forks having the same database dependencies if the source is in the dataset. Otherwise, we keep the project with the longest history, *i.e.,* the project in more dataset versions of Libraries.io. More details of this filtering can be found in our replication package [3]. We identified and removed 1,567 duplicated repositories and finally gathered 40,609 projects (26,745 in 2017 and 38,248 in 2020).

Table 1. Projects by programming language and data model (2020)

Data model	Ruby	JavaScript	Python	Java	C#	PHP
Relational	11,049 (44.66%)	4,164 (16.83%)	4,863 (19.66%)	3,707 (14.98%)	956 (3.86%)	2 (0.01%)
Document	953 (9.76%)	6,126 (62.73%)	1,435 (14.70%)	782 (8.01%)	273 (2.80%)	196 (2.01%)
KeyValue	2,548 (28.11%)	2,831 (31.23%)	2,522 (27.82%)	927 (10.23%)	228 (2.52%)	8 (0.09%)
Column	34 (3.26%)	79 (7.57%)	549 (52.64%)	350 (33.56%)	27 (2.59%)	4 (0.38%)
Graph	49 (12.04%)	91 (22.36%)	65 (15.97%)	176 (43.24%)	13 (3.19%)	13 (3.19%)
Total	12,370 (32.28%)	11,747 (30.66%)	7,558 (19.72%)	5,062 (13.21%)	1,362 (3.55%)	221 (0.58%)

Final dataset Table 1 presents an overview of the programming languages and different data models used in the projects of the 2020 dataset. The most common programming languages in the dataset are Ruby (32.28%), JavaScript/-TypeScript (30.66%), Python (19.72%) and Java (13.21%). Interestingly, Ruby, Python, Java and C# are mainly used in systems with relational databases, while document-oriented data stores are frequently used in JavaScript and PHP.

3 Study Results

3.1 RQ1: How prevalent are different data models and their related database technologies?

Table 2 presents the number of projects relying on different data models. Overall, more than half of the projects declare a relational driver dependency. Thus, the relational data model is the most used in the dataset. However, an interesting observation is that the percentage of relational databases is constantly decreasing among the database-dependent projects. It goes down to 54.72% from 57.40%

over the four years. In contrast, the ratio of projects relying on NoSQL data models is increasing. Except for graph data models, an increase in the usage ratio can be observed for document-oriented (21.30% to 21.97%), key-value (18.97% to 19.98%), and wide-column (1.40% to 2.44%) data models.

Table 2. Evolution of database-dependent projects by data model

Dataset	Relational	Document	Key-Value	Wide-Column	Graph
2017-07	17,816 (57.40%)	6,610 (21.30%)	5,886 (18.97%)	436 (1.40%)	288 (0.93%)
2017-11	19,112 (57.27%)	7,173 (21.49%)	6,284 (18.83%)	487 (1.46%)	317 (0.95%)
2018-03	19,622 (57.16%)	7,372 (21.47%)	6,524 (19.00%)	490 (1.43%)	323 (0.94%)
2018-11	21,037 (56.12%)	8,082 (21.56%)	7,349 (19.60%)	677 (1.81%)	344 (0.92%)
2020-01	24,620 (54.72%)	9,884 (21.97%)	8,989 (19.98%)	1,096 (2.44%)	402 (0.89%)

Figure 2 presents the database management systems used in the 2020 snapshot. MongoDB appears as the most used NoSQL database management system competing with relational technologies. It represents the majority of document-oriented database-dependent projects. Both Redis and Memcached are popular technologies for key-value data stores. Redis is almost as popular as SQLite and MySQL among the subject systems. Cassandra is the most used technology for wide-column stores. It is followed by Neo4j, the only graph-based technology on the list, then by HBase and Couchbase.

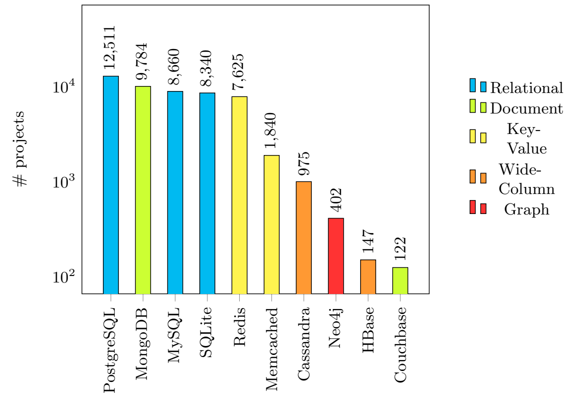
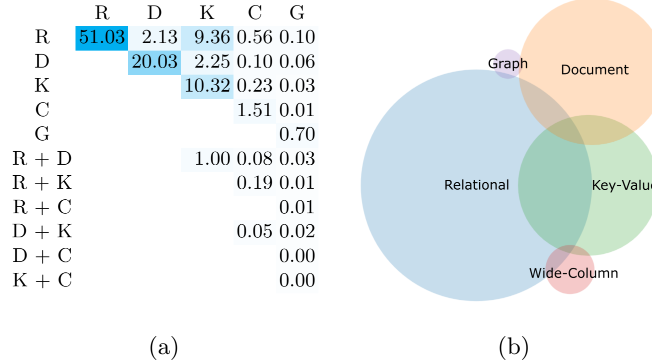


Fig. 2. Usage of database management systems (2020)

Overall, relational models lead the ranking as they are used in over half of the database-dependent projects. Document and key-values stores are the most used NoSQL technologies with approximately 21% and 19% of projects, respectively. Wide-column (1-3%) and graph-based (<1%) database families are much less represented. Interestingly, the proportion of relational database-dependent projects has decreased in the last four years, in contrast to NoSQL datastores.

3.2 RQ2: Are multi-database models frequently used in software systems?

This research question focuses on the heterogeneity of database-dependent projects, *i.e.,* projects having multiple dependencies to different data models.



	R	D	K	C	G
R	51.03	2.13	9.36	0.56	0.10
D		20.03	2.25	0.10	0.06
K			10.32	0.23	0.03
C				1.51	0.01
G					0.70
R + D			1.00	0.08	0.03
R + K				0.19	0.01
R + C					0.01
D + K				0.05	0.02
D + C					0.00
K + C					0.00

(a) (b)

Fig. 3. (a) Percentage of database-dependent projects with one, two and three data models (2020) (b) Distribution of hybrid database-dependent projects

Figure 3a gives the distribution of hybrid database-dependent projects in the 2020 dataset. We only present the percentage of projects relying on one, two or three different data models. The projects with more data models (four or five) represent only 0.19% of all filtered projects. Figure 3b depicts the most common combinations of data models in hybrid projects.

Answering this research question, we make the following observations. More than 16% of our database-dependent projects are hybrid, *i.e.,* define dependencies to access more than one database family. Most systems relying on a relational, document, or graph data model correspond to single-database projects. In contrast, more than 56% of the key-value projects are paired with another data model, typically with a relational or a document database. Another noticeable combination of hybrid projects groups about 45% of the total number of wide-column dependent projects with another technology, such as a relational database or a key-value database. The proportion of hybrid systems increased for almost all models in the last four years, except for key-value models

3.3 RQ3: How does data model usage evolve?

We follow the evolution of the repositories in our dataset and keep track of changes (*i.e.,* replacement, addition, or deletion) in their data models. For example, when developers change from a relational model to a document store, we record it as a *replacement* event. Similarly, when they add a new database technology to the existing one, we record it as an *addition* of a new data model. When we see they do not use a data model anymore, we consider it as a *deletion*.

We identified a total number of 471 repositories with changes in their data models throughout the time period of our dataset. We answer this research question considering only these systems.

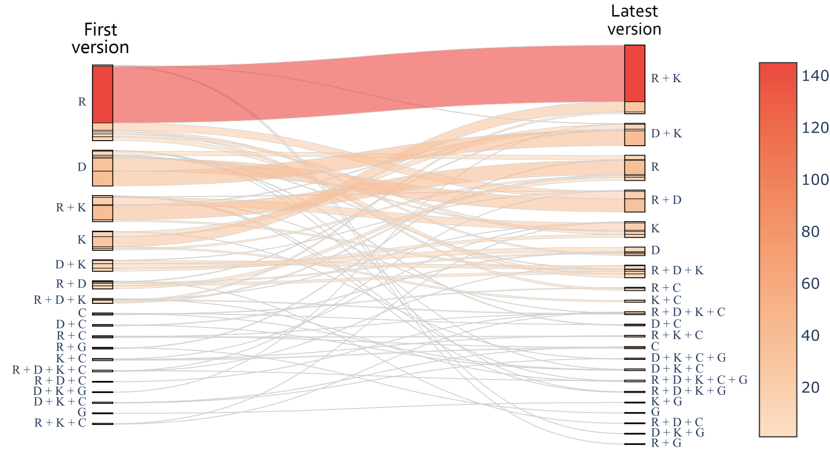


Fig. 4. Changes in projects' database families. Data models: R - relational, D - document, K - key-value, C - column and G - graph

Figure 4 presents how systems changed their data models between their first and latest known versions in our dataset. The first known version can be in 2017 or after it for projects appearing in our dataset later. The latest known version can be 2020 or before if the project became unavailable. For example, if a system had initially used a document store, then changed to a relational model, the system is counted as 'D' on the left side and 'R' on the right side of the diagram. The sizes and colors of the boxes represent the number of systems using the actual data model or the combination of multiple models.

We can make the following interesting observations by looking at the diagram. First, the most common change in the database family is adding a key-value database to an existing relational one. We identified 29.5% of the 471 repositories with such a design decision. This is in line with the findings of RQ2, where we found the combination of key-value and relational data models as the most common in hybrid systems. Second, we found a significant number of projects (7.6%) with the addition of a key-value database to an existing document database. The deletion of a key-value family from a relational and a key-value combination was also common (7.4%), and the addition of a relational database to an existing document database (6.8%), or the addition of a relational database to a key-value database system (5.3%) were also significant.

We also classify the repositories according to their database usage. *Becoming hybrid* indicates that a system starts using more than one database technology, and *staying hybrid* means that it keeps using multiple data models. Similarly, we classify systems as *becoming/staying mono-database* when they start/keep using a single database.

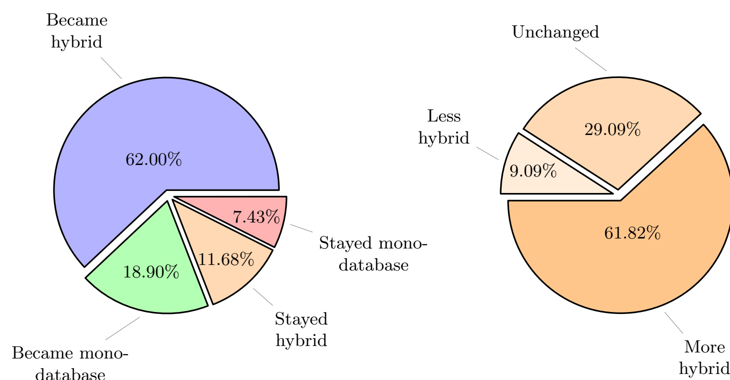


Fig. 5. Changes in data models

Figure 5 presents a summary of the classification of the changes in the database families. In 62% of the repositories which evolved their data model, we observed that the systems became hybrid. They had used a single database model before and added a new model later. On the contrary, 18.9% of the repositories had used a multi-database model before and became mono-database. The right side of Figure 5 elaborates on the systems that stayed hybrid. Here we observe a general trend of adding more data models to the existing ones as 61.82% of these systems become more hybrid.

To better understand developers in applying such changes to the systems, we contacted the developers of five systems where we found interesting changes in the evolution. We got a response from the developers of the ORCID project.[7] We observed the addition of MongoDB to an existing PostgreSQL database in this system in 2018. The developers used the two databases for a while, but then abandoned MongoDB. The dependency is still there, but it is not used anymore.

Manual inspection revealed that both databases used a table column as a dynamic flag updated when a MongoDB message was received in a queue managing that column. This interaction tracked the transfer of data from the PostgreSQL to the MongoDB database. As the developers explained, the team experimented with the benefits of querying the data from a document data store compared to a relational database. For their purpose, the migration had more disadvantages. Hence, they abandoned the document store later.

Overall, the results of this research question indicate the following. Evolving database-dependent projects tend to add more data models to their existing ones, turning mono-database systems into hybrid or already hybrid systems to even "more hybrid." The most common change in the combination of database families is adding a key-value database to an existing relational or document model. Adding a relational database backend to a current key-value/document database-based system is also a frequent combination. Whereas moving to more than two database families is seldom.

[7] ORCID Source – `https://github.com/ORCID/ORCID-Source`

4 Discussion and Implications

Our study confirms the *emergence of hybrid systems*, as 16.41% of subject systems used at least two database models in 2020. This number slightly increased from 2017, when 15.77% of the systems were hybrid. Looking at the evolution of the systems, we found a few (1%) that changed their database model by adding, deleting, or changing a database. In particular, 62% of the database model changes consisted of the addition of a new database. In a few cases (18.9%), these modifications were removals. Goeminne and Mens studied database frameworks in open-source systems [14], and there is an interesting parallelism with our work. They found that "*different database frameworks used in a project tend to co-occur*" and "*all database frameworks remain present in more than 45% of the projects.*" Similar findings were confirmed by Decan et al. [9].

Such observations make us assume that *developers rarely change database models* in the lifecycle of the project. However, *when modifications are needed, developers tend to add new database models* to the system. As previous research has shown, the database schema evolves continuously [7, 24, 26]. Keeping up with the changes is an expensive and error-prone task [27], especially when the database schema is not explicitly provided. Hybrid systems make this evolution process even more complex. The state-of-the-art approaches on data-intensive systems evolution [1, 9, 11, 17, 19, 21, 22, 25] mostly consider software systems relying on a relational database and typically written in Java [9, 17–19, 21] or PHP [1, 22, 25] (see Section 6). Our results show that more complex data architectures and other programming languages are emerging, bringing their own maintainability challenges.

Unfortunately, only a few authors have started investigating the challenges of evolving NoSQL applications [23,24,28] or supporting schema evolution in hybrid systems [12]. In this direction, promising ideas are the use of unified platforms to integrate multiple data sources [15] or provide support in managing multiple schemas [4]. Another approach is to manipulate a unified data schema [2] or query and migrate the data across different databases relying on relational or NoSQL technologies [13]. Developers could greatly benefit from more tools helping them maintain and evolve multi-database systems or recommend useful changes in their data models.

5 Threats to Validity

Construct validity concerns mostly the preparation of the dataset of our study. We rely on the dataset of Libraries.io that monitors Bitbucket, GitHub, and GitLab. While these are the largest open-source repositories, they might not fully represent all open-source projects using databases.

We excluded "low-quality" projects with a quality filter. To balance the quality and the size of the dataset, we chose filtering criteria according to quartiles. Similar, even more strict, values for contributors and stars are used in the literature [5, 20]. Filtering only based on popularity is prone to include projects

with non-software artifacts (*e.g.,* experimental or teaching purposes) [20]. To mitigate this risk, we use a combined filter and the requirement of database dependency. We also filtered project duplicates through the identification of fork projects with the same database dependencies. This filter could consider individual projects as duplicates and miss "non-forked" copies in rare cases. Overall, the filters resulted in less than 5% of all database-dependent projects. The number of projects remains sufficient to obtain representative results; however, the filters can affect our findings. A more in-depth analysis of the projects could still improve the dataset, particularly by classifying the projects and identifying false positive projects that define database dependencies without using them. Our **replication package** [3] provides all the necessary resources (datasets, extraction scripts) to replicate our study by selecting projects based on different quality criteria. Additionally, we performed control analyses on the whole dataset (*e.g.,* by filtering all fork projects) and did not find contradicting results.

The selection of database access libraries was based on keywords and included manual validation. We identified an exhaustive list of 707 drivers. This step can be biased; however, establishing a complete list of drivers is probably an unreachable goal. The actual list of drivers is also available in our replication package to alleviate this threat. It can be reused and extended in future studies.

Internal validity does not affect this study, being an exploratory study, we did not claim any causation. On the other hand, *external validity* concerns the extent to which our results can be generalized. We present observations concerning the use of seven programming languages of subject systems. Although these languages are popular today, the results might not be generalized to other programming languages. Similarly, databases not considered in our study might affect generalizability. In particular, we aim at open-source technologies, and proprietary systems remained outside of our study. Further research might be needed to investigate whether our findings hold on other technologies.

Conclusion validity concerns relationships in our observations. We do not investigate relationships; consequently, we do not perform statistical tests. However, relationships might exist that we did not observe in our study.

6 Related Work

In this section, we discuss the novelty of our work with respect to previous research literature. The most related study by Decan et al. [9] investigates the introduction and the co-existence of *relational* database *access* technologies. The authors analyzed the evolution of 2,457 Java projects on GitHub and focused on JDBC, Hibernate and JPA as database access technologies. They observed a significant technology migration from Hibernate to JPA but did not find evidence of the massive replacement of JDBC.

Several authors identified database accesses in source code to identify smells, inconsistencies or antipatterns. Muse et al. [21] performed an empirical study on GitHub projects and analyzed the prevalence of SQL code smells in Java applications. Dimolikas et al. [11] studied the evolution of tables in a relational

schema over time concerning the structure of the foreign keys to which tables are related. Meurice et al. [19] investigated the co-evolution of source code and database schemas with the ultimate goal to assist developers in preventing inconsistencies. Their study considered three Java systems. Qiu et al. [22] analyzed the co-evolution of SQL database schemas and code in ten open-source PHP applications. Anderson et al. [1] analyzed SQL queries embedded in PHP applications to support their understanding, evolution and security. Shao et al. [25] identified a list of database-access performance antipatterns, mainly in PHP web applications. Integrity violation was addressed by Li et al. [17], who identified constraints from source code and related them to database attributes.

Several previous studies exclusively focus on NoSQL applications. Störl et al. [28] investigated the advantages of using object mapper libraries when accessing NoSQL data stores. They overview Object-NoSQL Mappers (ONMs) and Object-Relational Mappers with NoSQL support. As they say, building applications against the native interfaces of NoSQL data stores create technical lock-in due to the lack of standardized query languages. Therefore, developers often turn to object mapper libraries as an extra level of abstraction. Scherzinger et al. [24] studied how software engineers design and evolve their domain model when building NoSQL applications, by analyzing the denormalized character of ten open-source Java applications relying on object mappers. They observed the growth in complexity of the NoSQL schemas and common evolution operations between the projects. Ringlstetter et al. [23] looked at how NoSQL object-mappers evolution annotations were used. They found that only 5.6% of 900 open-source Java projects using Morphia or Objectify used such annotations to evolve the data model or migrate the data.

The related approaches and studies described above primarily focus on the analysis, maintenance, and evolution of *either* relational *or* NoSQL systems. They mainly consider Java or PHP systems. The number of systems studied varies between 2.5 and 3 thousand projects. In contrast, our study investigates the use of *database models* in 40,609 open-source projects relying on relational *and/or* NoSQL database technologies. Therefore, our work does not restrict to single-database systems but specifically considers *hybrid* systems by covering five different database models (relational, document, key-value, column, and graph) possibly used in combination. Furthermore, we do not focus on a single programming language; we consider seven different languages (C#, Java, JavaScript, PHP, Python, Ruby, and TypeScript) among the most popular languages today.

7 Conclusion

This paper investigates the (joint) use of SQL and NoSQL database models in open-source projects. We started from several million projects, keeping those defining at least one database dependency based on a list of database drivers for the most common programming languages. We then selected only the projects meeting specific quality requirements.

We found that the majority of current database-dependent projects (54.72%) rely on a relational database model, while NoSQL-dependent systems represent 45.28% of the projects. However, the popularity of SQL technologies has recently decreased with respect to NoSQL datastores. As far as programming languages are concerned, we noticed that Ruby and Python systems are often paired with a PostgreSQL database. At the same time, Java and C# projects typically rely on a MySQL database. Data-intensive systems written in JavaScript/TypeScript are essentially paired with document-oriented or key-value databases.

Our results confirm the emergence of hybrid data-intensive systems where (multi-) database models (*e.g.,* relational and NoSQL) are used together (16% of all database-dependent projects). In particular, we found that more than 56% of systems relying on a key-value database also use another database technology, typically relational or document-oriented. Wide-column dependent systems follow the same pattern, with over 47% of them being hybrid. This demonstrates the complimentary usage of SQL and NoSQL database technologies in practice. We also observe that one percent of the database-dependent projects evolved their data model. The majority (62%) were not born hybrid but once relied on a single database model. In contrast, 19% became "mono-database" after initially using multiple database models.

Future work could benefit from analyzing the popularity of business domains in which combinations of database technologies are used together. An investigation of the rationale for adopting or giving up hybrid architectures could also reveal practical implications or lessons that would help the developers. Our findings provide a clear motivation to further understand and address those challenges and constitute an important step towards supporting developers to design and evolve hybrid data-intensive systems.

Acknowledgments. This research is supported by the F.R.S.-FNRS and FWO EOS project 30446992 SECO-ASSIST and the SNF-FNRS project INSTINCT.

References

1. Anderson, D., Hills, M.: Supporting Analysis of SQL Queries in PHP AiR. In: SCAM 2017. pp. 153–158. IEEE (2017)
2. Basciani, F., Rocco, J.D., Ruscio, D.D., Pierantonio, A., Iovino, L.: Typhonml: a modeling environment to develop hybrid polystores. In: MODELS 2020. pp. 2:1–2:5
3. Benats, P.: Repl. pkg., `https://github.com/benatspo/Multi-database_Models`
4. Bernstein, P.A., Melnik, S.: Model management 2.0: Manipulating richer mappings. In: SIGMOD '07. p. 1–12. SIGMOD '07, ACM (2007)
5. Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P.: Don't touch my code! examining the effects of ownership on software quality. In: ESEC/FSE '11. p. 4–14. ACM (2011)
6. Borges, H., Tulio Valente, M.: What's in a GitHub star? understanding repository starring practices in a social coding platform. JSS **146**, 112–129 (2018)
7. Cleve, A., Gobert, M., Meurice, L., Maes, J., Weber, J.: Understanding database schema evolution: A case study. Science of Comp. Progr. **97**, 113–121 (2015)

8. Davoudian, A., Chen, L., Liu, M.: A survey on NoSQL stores. ACM Computing Surveys (2018)
9. Decan, A., Goeminne, M., Mens, T.: On the interaction of relational database access technologies in open source Java projects. In: SATTOSE 2015. pp. 26–35
10. Decan, A., Mens, T., Grosjean, P.: An empirical comparison of dependency network evolution in seven software packaging ecosystems. EMSE. **24**(1), 381–416 (2019)
11. Dimolikas, K., Zarras, A.V., Vassiliadis, P.: A study on the effect of a table's involvement in foreign keys to its schema evolution. In: ER 2020. pp. 456–470
12. Fink, J., Gobert, M., Cleve, A.: Adapting queries to database schema changes in hybrid polystores. In: SCAM 2020. pp. 127–131 (2020)
13. Gobert, M.: Schema evolution in hybrid databases systems. In: VLDB 2020 (2020)
14. Goeminne, M., Mens, T.: Towards a survival analysis of database framework usage in Java projects. In: ICSME 2015. pp. 551–555
15. Jovanovic, P., Nadal, S., Romero, O., Abelló, A., Bilalli, B.: Quarry: A user-centered big data integration platform. Inf. Systems Frontiers **23**, 9–33 (2021)
16. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining GitHub. In: MSR 2014. p. 92–101. ACM
17. Li, B., Poshyvanyk, D., Grechanik, M.: Automatically detecting integrity violations in database-centric applications. In: ICPC 2017. pp. 251–262. IEEE
18. Linares-Vásquez, M., Li, B., Vendome, C., Poshyvanyk, D.: Documenting database usages and schema constraints in database-centric applications. In: ISSTA 2016. pp. 270–281 (2016)
19. Meurice, L., Nagy, C., Cleve, A.: Detecting and preventing program inconsistencies under database schema evolution. In: QRS 2016. pp. 262–273. IEEE
20. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating GitHub for engineered software projects. Empirical Softw. Engg. **22**(6), 3219–3253 (Dec 2017)
21. Muse, B.A., Rahman, M.M., Nagy, C., Cleve, A., Khomh, F., Antoniol, G.: On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. In: MSR 2020. pp. 327–338
22. Qiu, D., Li, B., Su, Z.: An empirical analysis of the co-evolution of schema and code in database applications. In: ESEC/FSE'13. pp. 125–135 (2013)
23. Ringlstetter, A., Scherzinger, S., Bissyandé, T.F.: Data model evolution using object-nosql mappers: Folklore or state-of-the-art? In: 2nd International Workshop on BIG Data Software Engineering. pp. 33–36 (2016)
24. Scherzinger, S., Sidortschuck, S.: An empirical study on the design and evolution of NoSQL database schemas. In: ER 2020. pp. 441–455. Springer (2020)
25. Shao, S., Qiu, Z., Yu, X., Yang, W., Jin, G., Xie, T., Wu, X.: Database-access performance antipatterns in database-backed web applications. In: ICSME 2020. pp. 58–69. IEEE
26. Sjøberg, D.: Quantifying schema evolution. Information and Software Technology **35**(1), 35–44 (1993)
27. Stonebraker, M., Deng, D., Brodie, M.L.: Database decay and how to avoid it. In: Proc. Big Data (2016)
28. Störl, U., Hauf, T., Klettke, M., Scherzinger, S.: Schemaless NoSQL data stores-Object-NoSQL Mappers to the rescue? BTW 2015
29. Sun, Z., Liu, Y., Cheng, Z., Yang, C., Che, P.: Req2Lib: A semantic neural model for software library recommendation. In: SANER 2020. pp. 542–546
30. Yamamoto, K., Kondo, M., Nishiura, K., Mizuno, O.: Which metrics should researchers use to collect repositories: An empirical study. In: QRS 2020. pp. 458–466