The Ripple Effects of Database Evolution in the Application Software Stack

Wolfgang Mauerer
Technical University of Applied Sciences Regensburg
Siemens AG, Corporate Research
Regensburg/Munich, Germany
wolfgang.mauerer@othr.de

Atalay Karatay
Technical University of Applied Sciences Regensburg
Regensburg, Germany
karatay.atalay@gmx.de

Dimiri Brainingер
Technical University of Applied Sciences Regensburg
Regensburg, Germany
dimitry.brainingер@st.othr.de

Stefanie Scherzinger
University of Passau
Passau, Germany
stefanie.scherzinger@uni-passau.de

ABSTRACT

In the architecture of data-intensive applications, the database is a suspected *dependency magnet*: modifications of the database schema are believed to substantially impact the remaining application code, albeit often based on qualitative evidence. In related contexts, changes with impact on large portions of the code base are known as *ripple effects*. In this paper, we aim at understanding the impact of hidden, database-induced dependencies and their ripple effects. As we show, such dependencies largely elude established analysis approaches, in particular, co-change analysis. Instead, we propose a semantically enriched generalisation of co-change dependencies that allows us to capture long-range dependencies spread across multiple commits. We also investigate a related set of usually hidden dependencies that stems from support for several database-vendor-specific schemas, which can be handled and understood with our approach. The induced couplings add further complexity to the well-explored couplings between a (single-vendor) database schema and the application code.

We present techniques built on a-priori expert knowledge that allow for an automatic identification of database related code changes with high precision and recall. We demonstrate that this type of change is very common in real-world applications, yet lacks attention in software engineering and architecture. We apply and evaluate the techniques on eight real-world data-intensive applications ranging from content management systems to domain name servers. Our insights can serve as a basis for better tool support for professional software development and maintenance, and to improve software architectures for cross-functional concerns at the intersection of database and software engineering.

ESEC/FSE'21, August 23–27, Athens, Greece

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Wolfgang Mauerer, Atalay Karatay, Dimiri Brainingер, and Stefanie Scherzinger. 2021. The Ripple Effects of Database Evolution in the Application Software Stack. In *Proceedings of Foundations of Software Engineering (ESEC/FSE'21).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Processing data is a core endeavour of computer science and information technology. With software as driving force of computer systems, and databases at the heart of data storage, one would assume that software engineering and database research were two closely intertwined fields that produce coordinated research insights. Databases are, at least within their own research community [64], believed to be "dependency magnets" in software architectures. Handling dependencies between artefacts and components, on the other hand, is a long and well researched topic in software engineering [44]. However, previous work has neglected to sufficiently appreciate the connection and its implications, as we argue in this paper.

Relational database management systems (RDBMS) form an extremely successful and widely deployed class of databases. They use *schemas*, typically described in the data definition (sub)language (DDL) of SQL [52], to define the logical and physical layout of data storage, and interact via various interfaces with applications that process and use data stored in the RDMBS. Software changes are caused by changing requirements and evolving features [43], and the same holds for the database component (*e.g.*, data formats [39]), and notably, its schema [3, 4, 18, 22, 39, 50, 60, 61, 63, 64, 67]. Collateral evolution of the database schema and the application code is a known challenge, and has been studied in the past [15, 39, 50]. Analysis of couplings and dependencies are key to understanding structure and evolution of software. Unfortunately, as we demonstrate, common mechanisms for detecting couplings and implicit dependencies do not apply well to reveal relationships between database schemas and application code.

Nonetheless, changes to databases are known to cause substantial maintenance effort [16, 39]: When the underlying database schema changes, queries and their result structures change [18], which in turn can require comprehensive code changes. Unfortunately,

abstraction layers like object-relational mappers (ORM) are no panacea, as the database is known to nevertheless inject dependencies throughout the code base [33]. Code that depends on a database schema has to co-evolve with it, one way or another.

In this work, we construct methods to automatically identify couplings between the database, in particular the database schema, and dependent code. Our methods alleviate the need for analyses to assume that coupled changes happen in a single commit. Instead, they use a novel application of a-priori information to obtain semantic information across commits. We use these techniques to show that the influence of databases on software architectures has not yet been fully understood, possibly because it concerns uncharted territory between software and database engineering. Moreover, we show that database-induced dependencies are more pronounced than previously recognised, especially in two aspects:

(a) Long-range DB/code dependencies. Changes that relate a modification to a database schema, along with the necessary code adaption, are assumed to be performed in a single commit in traditional co-change analysis. We empirically find that this concentration is not upheld by many projects. Rather, we find that co-changes may be temporally distributed over series of commits, leading to what we call *long-range couplings*.

(b) Dependencies between schema variants. Often, large-scale data-centric applications such as MediaWiki, Roundcube, or Joomla!, support *multiple* database backends that must be handled and maintained in code, and of which only one can be chosen for deployment. Often, the different database backends are used to implement the same, or at least very similar, functionalities.

The development team must maintain *vendor-specific schemas* that remain synchronised with each other: Any changes to the conceptual schema underlying the application must be transferred to all vendor schemas. Moreover, developers need to maintain variations of the database population and migration scripts (*e.g.*, for upgrading the production database from one version of the schema to another), and must cope with variant-specific code within a database access layer (DBAL) or other parts of the application.

Hypotheses. The overall question we address is as follows:

> *What is the influence of hidden, database-induced dependencies and their ripple effects on software development and architecture?*

Before we refine this question into specific research hypotheses that address different aspects of the general problem, let us outline our overall results and contributions:

(1) We show that databases introduce dependencies between engineering artefacts that are not reliably captured by standard techniques. In particular, we identify multivariate vendor/vendor dependencies by including domain-specific a-priori knowledge, and introduce an automated analysis that lifts restrictions of previous approaches by including semantic meaning of commits into co-change techniques. The method is validated with a ground truth obtained by multiple human classifiers.

(2) We show that an extension of our mechanism to long-range couplings substantially increases the number of successfully detected co-changes, and that standard coupling analysis mechanisms do not faithfully capture important aspects of such database-induced dependencies.

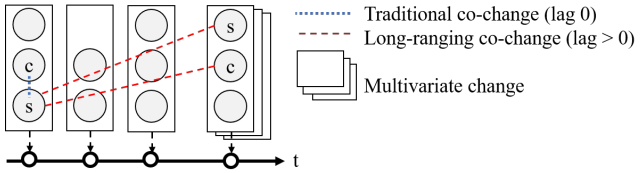

Figure 1: DB-induced couplings along the commit history (t): sql-files (s) and DB-code changes (c), within in the same commit, are captured by traditional co-change analysis. Long-ranging code-changes, spanning several commits, require new approaches. Multivariate DB-applications, with several vendor-specific schemas, add additional complexity that must be accounted for by analyses.

(3) We likewise extend the vendor/vendor mechanism to long-range couplings, and find that this likewise addresses a neglected, but practically relevant scenario that identifies further database-related dependencies. This is especially important for the future design of tools to handle schema evolution. The database research community has mainly focused on proposing tools to support the operations team in managing different schema versions (*e.g.*, [17, 19, 31, 47, 65]). Yet for the family of applications considered here, there is only one specific database product running in production. Thus, these tools do not address the problem of multivariate vendor/vendor evolution, as faced by the software development team.

Our results were obtained by two redundant and independently built analysis pipelines. We thus provide our own replication study. The complete source codes for both analysis pipelines, including input data and generated artefacts, is available as a reproduction package on the supplementary website (clickable hyperlink in PDF). The website also contains detail results omitted here.

Structure. Before we address our hypotheses, we review related work in the context of our efforts in Section 2. We describe and formalise our approach in Section 3, where we also take care to ascertain methodological soundness using a statistical analysis.

2 RELATED WORK

Our work lies at the intersection of software engineering and database research, and mandates reviewing related work from both communities. Since we find that database issues have not reached the full consideration they deserve in software engineering, we particularly try to provide a thorough review of this aspect.

2.1 Dependency and Coupling Mechanisms

Countless mechanisms have been proposed for detecting dependencies between software artefacts; we refer to Ref. [44] and references therein for a comprehensive introduction. Our work concentrates on *semantic* co-changes to software artefacts. Co-change (or evolutionary dependency) analysis [36, 66] is built on the premise that artefacts are coupled when they (frequently) change together in a single commit. Database schema changes are sometimes filtered out by tools as non-programming language artefacts. But even if they are captured by a co-change implementation, their

small occurrence frequency, caused by their disruptiveness [50] and inherent complexity, increases their impact to a few rare, but the more invasive changes [64]–developers try to actively avoid or delay [60, 62] schema changes at all costs. This rarity makes it easy to underappreciate their significance compared to regular, frequent code co-changes.

The restriction of dependency inference to single commits is lifted by association rule mining [68] or semantic couplings [34, 49, 55]. In addition to the rare occurrence of schema changes as such, schemas are also known to be sparsely documented [40], which renders such data-intensive analyses impractical.

Static dependencies [10] such as obtained from function calls do not capture database schemas by design, since these do not constitute programming language code implementing concepts like function calls int the first pace.[1]

We find that database schema changes do transcend the immediate temporal vicinity of the `sql`-file commit. So far, there are few established methods for analysing couplings across several commits (*incomplete commits*) [44], although observations about *ripple effects* [8, 69] go back to at least 1978.

Another important aspect relates to identifying the semantics of a change. Semantic couplings break ground in this direction, albeit based on machine learning techniques, and without *prescribed* semantic notions. Classifying commits by content [32, 56] is related to our approach, albeit different classification categories are used (*e.g.*, bug fix, enhancement, documentation update), usually learned from a textual representation of the change, instead of using classifiers constructed from a-priori expert knowledge.

Popular tools in software engineering research, including Scikit's understand [2] or Lizard [41] (used by GrimoireLab [23]), do not support SQL, and cannot detect schema related couplings irrespective of lags. Research that considers file-level artefacts does not need to parse the content of SQL schema files (and can therefore easily support `sql`-files). However, Refs. [13, 35] show that dependencies obtained on simultaneous changes to localised regions/functions, or even lines [59] are more reliable than the mere fact that developers worked on the same file, although few studies have adopted such fine-grained approaches [29] so far. Our approach rests on function-level decomposition wherever possible.

2.2 Databases and Software Engineering

It is a widely used assumption in the database community that databases are a major source of dependencies in software engineering [64]. In fact, the seminal specialist book [4] by Ambler and Sadalage emphasises the challenge of evolving database applications given these dependencies.

Nevertheless, the seminal textbooks on software patterns, Refs. [11, 12, 26], spend surprisingly little attention on databases. Essentially, they concentrate on the use of a database access layer and some selected techniques to decouple database structure and applications. In a more recent textbook, Fowler [25] argues in favour of a small number of recommended patterns for DB applications; the seminal textbook on software architecture by Bass *et al.* [6] only dedicates a few pages on how to integrate databases in software architectures, essentially advocating a database access layer.

Additionally, all references restrict their consideration to *single-vendor* schemas. That is, it is assumed that the application is backed by a specific database, for instance, PostgreSQL. However, many extremely popular applications support multiple alternative database backends in development (we call such applications *multivariate*), but only use one DB-variant in deployment, which is closely related in spirit to configurable software product lines [5]. Rosenmüller *et al.* [54] study tailor-made, problem-specific SQL dialects generated from a family of SQL dialects, which is also related to the problem of handling different vendor-specific SQL dialectes. In general, the problem of multivariate database applications has only recently started to gain attention in the software and database engineering communities. Vassiliadis *et al.* [62] conduct the so far largest-scale schema evolution analysis (over 195 applications). The authors report on the difficulty of multi-variate DB applications during data collection, and ultimately, resort to analysing the schema of one vendor only. First suggestions for tool-based handling of multi-variate database architectures appear in Ref. [57]

Nevertheless, prior work on database aspects of software engineering—with the notable exception of Qiu *et al.* [50]—usually ignores the role of the commit history, and in particular, the time frame that needs to be considered for analysing the implications of a commit. While Qiu *et al.* focus on the schema changes in a single-vendor scenario (even for repositories that actually are multi-variate), we consider the problem in its full generality.

Interestingly, it has also been shown that object-relational mapper frameworks such as Hibernate do not prevent dependencies between the database and the application code [33, 45].

Delplanque *et al.* study the evolution of a database application in production [20]. The team quantifies the substantial efforts caused by schema evolution, and confirms other studies, decades prior [60].

Coupling between databases and application code is often made more visibly with schema evolution, and manifests in a recognised maintenance challenge. The earliest mentions we are aware of date back several decades [60]. Ref. [21] describes a framework for evolving multi-database (which is not entirely identical with our scenario, but refers to multiple simultaneous databases in production) applications based on CORBA, and evaluate their framework on an artificial prototype of such an application. Based on a similar notion of coupling as our work, Gardikiotis *et al.* [27] estimate the impact of schema changes on a web application. Follow-up work [28] proposes a more general impact analysis of schema changes on application code, based on an extended version of the program control flow graph, and relying on program slicing. The approach is evaluated using a fictional application, which does not include real-world commit cultures that we find essential in our study. Maule *et al.* [42] analyse schema changes on 62 revisions of a single commercial system, and identify locations in the source code that might be affected by the change using code flow analysis.

It has been shown that even when persistence frameworks (ORM mappers) are used, there are still pronounced dependencies between the database and the application code, including test cases [33].

In essence, we find that the amount of scientific literature on the intersection of software and database engineering is comparatively meagre, given the long individual history of both fields. We also

[1] Of course, most relational SQL-based databases offer stored procedures and similar mechanisms, but these extensions do not touch the structural properties of schemas.

sense a large gap between state-of-the-art approaches propagated in research, and the tools actually used in practice (*e.g.* observed in a real-live study in [20], and as a general observation in [37]).

2.3 Database Evolution

Schema evolution itself is an intensively studied topic in the database community. Schema changes are commonly called *schema modification operations* (SMOs) [18]. Ambler *et al.* propose a classification of SMOs [4], which has also been applied in research [50]. Ambler lists SMOs that concern the logical schema (*i.e.*, creating or dropping a table, adding/dropping a column), and SMOs that concern the physical schema (i.e., adding or dropping an index). One of the strongest selling points in using database systems is that they distinguish a logical and a physical schema. Thanks to the desirable property of data independence [52], the physical schema may be changed without affecting the logical schema, or the application code. Studies have shown that SMOs concerning the physical schema are comparatively rare [50] for the majority of projects.

Existing schema evolution research mostly focuses on an empirical understanding of the frequency and nature of schema changes (*e.g.* [9, 18, 22, 39, 50, 60, 61, 63, 64, 67]), and usually ignores consequences for the code. Among the few exceptions, Qiu *et al.* [50] assume a more holistic point of view, and Curino *et al.* [18] study the impact of schema changes on queries embedded in the application code. Neamtiu *et al.* [39] study coupling between database schema and migration scripts for migrating production data between schema versions. In particular, they identify vendor-side changes (*e.g.*, the database file format) as disruptive to applications.

Overall, empirical studies confirm that the schema evolves, and that couplings between database and code cause maintenance overhead. Nevertheless, there is little quantitative evidence further characterising these dependencies, as we set out to provide.

3 RESEARCH DESIGN

3.1 Reference Projects

We analyse a diverse set of eight subject projects, from content management systems to a DNS nameserver; Table 1 states key characteristics like total number of general, schema, or database-related commits, lines of code, etc., and shows that we selected projects that vary in a number of important dimensions: (a) size (from 10k to 500k LoC); (b) age (time since first commit; ten to twenty years); (c) programming language (C++, PHP, Java); (d) application domain; (e) support for at least three different database vendors.

While our selection does not (and cannot) guarantee results that fully generalise to every database centric application, it should represent many typical choices. Moreover, most subject projects have already been subjected to published empirical study [50, 67], or even serve as a widely used schema evolution benchmark [18]. Thus, we cover de-facto reference projects in the analysis of evolving database applications, and have also tried to ascertain that the subject projects represent practically successful database applications that enjoy popularity in the open source communities.

3.2 Analysis Process

One of our main tasks is to identify database-related artefacts in repositories. Typical choices for artefacts in repository mining research use either complete files, or resort to a more fine-grained decomposition of the source code into functions. An analysis of co-changes based on function artefacts is known to deliver considerably better results [35]. Therefore, our analysis operates at the function artefact level. We use established repository mining and analysis tools [35] to decompose source files into artefacts.

Built-in replication study. We applied due diligence in our data analysis. Results were obtained through *two* analyses, with different, independently written analysis pipelines. The second analysis was able to reproduce the results of the first (except for some negligible differences in the sub percent range). The supplementary website describes both pipelines, compares the results, and provides full sources for independent reproduction.

Commit Classification by Witnesses. We implement *witness* functions to categorise artefacts as database related or not. Based on a-priori knowledge and a manual analysis of the database APIs and other abstraction mechanisms (or even SQL keywords embedded in command strings, should projects use such techniques), we compiled lists of project-specific keywords (available on the supplementary website) that mark artefacts as database related when their content matches a keyword. We use two types of witnesses:

(1) A *database code witness* identifies general programming language code that performs an operation related to database issues. At the level of function artefacts, this might check for calls to variables named after project-specific conventions, *e.g.* "`$dbr->`" for the database connection in PHP, or API calls such as "`wfGetDB(`", for MediaWiki. A file may contain artefacts that are related to databases, and others that are not.
It is also possible to specify file-level database code witnesses that identify *all* artefacts in a file as database related, either based on content string matching, or pathname matching. This is useful to compactly classify files that implement, for instance, a database access layer. An example for Joomla! is the file reachable by the path `installation/helper/database.php`.
(2) A *vendor witness* checks for `sql`-files that contain DDL (data definition language) statements, as well as DML (data manipulation language) and rights management statements. Recognition for this witness is at the file level, and can often be performed by checking for certain file extensions (`.sql`) or relative base file paths in the repository.
We experiment with two implementations: (a) The first recognises any change to a `sql`-file, and (b) the second recognises a change to the subset of files where the main database schema is declared (this is identified by applying a-priori knowledge). In particular, the files recognised by implementation (a) additionally contain files that are used by the operations team to migrate production data to the current schema (via ALTER TABLE statements). Unless stated otherwise, we rely on implementation (a), but we will also explore whether there are significant differences by this more coarse-grained approach.

Implementation details and exact witness specifications are available in the replication package on the supplementary website.

Table 1: Overview of the investigated projects (PDF provides clickable links for repository names and hashes). #DBs: number of supported vendors; #Cmts: number of commits; #DB-C: number of multivariate sql-file commits.

Project	Description	Time Range	#DBs	#Cmts	#Schema	#DB-C	Hash	Lines Of Code (@Hash)		
								Schema	DB-C	Non-DB
BiblioteQ	Library Mgmt	03/08 ~ 05/20	3	2.817	162	58	97b46e1	2.815	40.239	246.150
Joomla!	CMS	09/05 ~ 05/20	3	32.230	1.590	377	be05a58	25.893	266.912	891.588
MediaWiki	Wiki	01/02 ~ 05/20	6	96.624	1.121	151	02d827e	14.990	263.606	1.834.769
OSCAR EMR	Medical SW	11/02 ~ 05/20	3	22.693	1.678	13	0aabdfd	343.791	523.623	1.610.974
phpBB	Bulletin Board	02/01 ~ 05/20	7	34.425	1.442	343	2bf9202	8.873	182.491	267.404
PowerDNS	Nameserver	11/02 ~ 05/20	8	19.017	111	32	628cefe	1.114	13.283	545.410
Roundcube	Webmail	09/05 ~ 05/20	5	11.730	148	70	46d3cae	3.154	23.148	280.777
TYPO3	CMS	10/03 ~ 05/20	5	29.606	550	13	4c6d801	1.724	308.373	965.438

Intuitively, a witness checks if a commit satisfies a certain property. We are interested in four discernible properties as shown in the table right below. Therefore, we solve a multi-class classification problem that assigns a commit into one of these categories "S" refers to a sql-file, "DBC" to database related code; ✓ signifies presence, and ✗ absence of such a change in a commit.

Class	S	DBC
⓪	✗	✗
①	✗	✓
②	✓	✗
③	✓	✓

Differently from the seminal work of Qiu *et al.* [50], which relies on manual analysis of co-committed code and assumes all dependencies between schema and related code are resolved within single commits, we fully automate our classification and can scale to any number of commits for arbitrary lags, based on a project specific set of witness functions that needs to be provided by human experts. Compared to a general manual analysis of commits, this is a one-time up-front investment whose effort is independent of the number of analysed commits. In a first step, two of the authors have devised and refined automatic classifiers based on a-priori, expert knowledge, and manual inspection of the subject projects.

Formalisation. To make our scenario precise, let us commence with a formal description. We assume basic familiarity with vector norms, specifically, the family of p norms defined by $\|\vec{x}\|_p = \left(\sum_{i\in\mathbb{N}} \|x_i\|^p\right)^{1/p}$, where the special cases 1 and ∞ used in this paper reduce to $\|\vec{x}\|_1 = \sum_i |x_i|$, and to $\|\vec{x}\|_\infty = \max(|x_1|, \ldots, |x_n|)$.

Consider a repository whose commits have been linearised into a sequence of totally ordered commits $\vec{C} = (c_1, c_2, \ldots, c_N)^{\mathrm{T}} \in H^N$ in the usual way (see, *e.g.*, Refs. [7, 35]). Each commit is characterised by a tuple $(h, t_a, t_c, m, \{f_1, \ldots, f_n\}) \in H$, where h is a unique hash value, t_a and t_c denote author and committer timestamps, m is the commit message, and $\{f_i\}$ the set of all changes induced by the commit, grouped by file.[2] We denote the number of (non-merge) commits in the repository under consideration by N. The total ordering guarantees that $t(c_1) \le t(c_2) \le \cdots \le t(c_N)$, where $t : H \to \mathbb{N}$ extracts committer time t_c out of a commit.

We infer properties of a commit using a *witness* function

$$W : H \to \mathbb{F}_2$$

that maps a commit $c \in H$ to a Boolean value, indicating presence or absence of a property. For instance, we may use a witness to indicate whether a commit contains a change to any schema file or not; another witness decides whether a commit contains a change to database-related code.

We group related witnesses in vectors $(w_1, w_2, \ldots, w_k)^{\mathrm{T}}$. Since we are interested in detecting properties of a given time interval, that is, sub-sequences of $\vec{C}$, we define a *lag operator* with signature

$$\hat{L}^p_{i,j} : (H \to \mathbb{F}_2)^k \times H^N \to \mathbb{N}_0,$$

where $i, j \in \mathbb{N}$ and $p \in \{1, \infty\}$. Given witnesses $\vec{w} = (w_1, w_2, \ldots, w_k)^{\mathrm{T}}$, and a sequence of commits in $\vec{C}$, the operator applies as

$$\hat{L}^p_{i,j}\, \vec{w}\, \vec{C} = \left\| \bigvee_{n=i}^{i+j} \vec{w}\, c_n \right\|_p$$

Example. Intuitively, the lag operator computes a collective property for an interval of commits. The application of witnesses to a commit, denoted by $\vec{w}c_n$, results in the vector

$$(w_1(c_n), w_2(c_n), \ldots, w_k(c_n))^{\mathrm{T}} \in \mathbb{F}_2^k.$$

Depending on the choice of p, either sum (o) or maximum (∞) norm are used. Intuitively, for $p = \infty$, the operator checks *existence* of the witnessed property in commit range $[i, i+j]$, whereas for $p = 1$, the operator counts *how often* a property is satisfied.

Fig. 2, shows three commits (labelled 7-9), together with the files changed in each commit. We visually distinguish sql-files and DB-relevant code changes. In the figure, we show the result of applying he vendor witnesses $\vec{w}_v$ to the commits, to register when sql-files for the vendors MySQL, DB2, or PostgreSQL are changed.

We further show the application of the DB-code witnesses $\vec{w}_{dbc}$ to the commit history, to track whether any DB-relevant code was changed in a commit, and how this code was recognised (on a per-file basis, or by detecting DB-specific API keywords).

We then compute the lag operators, as shown in the table to the right. This table reads as follows. We focus on commit 7 as our starting point. This is a schema-commit, as the file mysql.sql changes. Let us consider the vendor witnesses first, and focus on the second column: With a lag of o, we only encounter a univariate schema change. Extending the lag to 1 and 2, we observe a multivariate

[2] When multiple hunks for a single file are present in a commit, we count these as one combined change to a file. The layout of how a commit is presented in git differs from our exposition, because this is not relevant for the analysis.

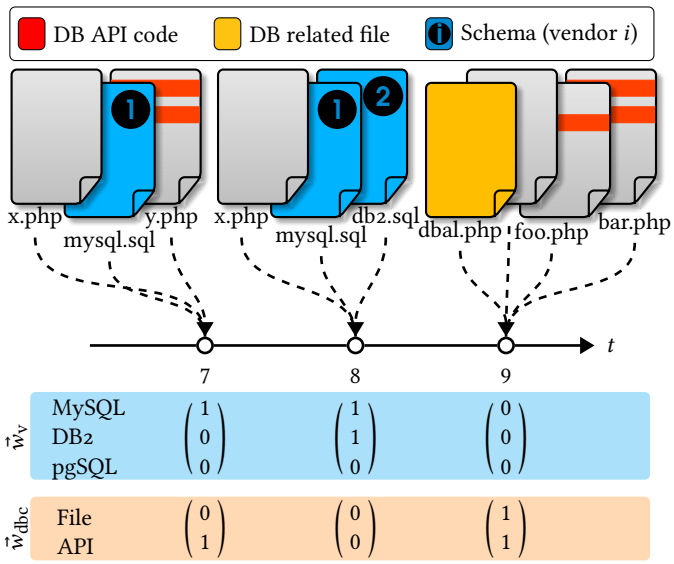

Figure 2: A linearised git commit history, and the application of witness vectors (pgSQL = PostgreSQL).

commit (with the lag operator capturing two changed schemas). For lags 0-2, there is at least one schema change (third column).

We focus on the database code witnesses next. Already with lag 0, we encounter a coupled DB-code change (fifth column).

Thus, the lag operator allows us to track whether database related code is encountered, and how many vendors are witnessed, within a given window.

We focus on the database code witnesses next. Already with lag 0, we encounter a coupled DB-code change (fifth column).

Thus, the lag operator allows us to track whether database related code is encountered, and how many vendors are witnessed, within a given window.

	$\vec{w} = \vec{w}_{\mathrm{v}}$		$\vec{w} = \vec{w}_{\mathrm{dbc}}$	
p	1	∞	1	∞
$\hat{L}^{p}_{7,0}\,\vec{w}\,\vec{C}$	1	1	1	1
$\hat{L}^{p}_{7,1}\,\vec{w}\,\vec{C}$	2	1	1	1
$\hat{L}^{p}_{7,2}\,\vec{w}\,\vec{C}$	2	1	2	1

Quality Analysis. To verify the quality of our classification mechanism–following the recommendations by Reyes *et al.* [53] for software engineering research–, we use a random sample of 20 commits per project, uniformly and randomly sampled from the four classes, as decided by the automatic classifier. This results in 160 commits (which is comparable to the sample of 146 commits of Ref. [50]) subjected to an independent, manual classification by all four authors, *without* up-front consultation of the automatic classifier results. Decisions were based on content of the commit and the complete textual content of the artefacts modified by each commit. A custom UI tool was used to minimise manual glitches; discrepancies were resolved by consensus. Table 2 shows the resulting multi-class confusion matrix [30].

We achieve an overall accuracy of 88.75%, and the 95% confidence interval spans [82,8%, 93.2%], as computed by the methods provided by Refs. [38, 48, 51]. Precision, recall and F1-score by class are provided in Table 2; larger values can hardly be expected for

Table 2: (Left, grey background) Multi-class confusion matrix for database code witnesses, and related statistical characteristics of the classifier. (Right) Class-resolved statistical characteristics. Classes (0)–(3) as defined in the main text.

		Ground Truth (Reference)						
		⓪	①	②	③	Prec.	Recall	F1
Prediction	⓪	23.1%	5.6%	0.0%	0.0%	80%	93%	86%
	①	1.9%	19.3%	0.0%	0.0%	91%	78%	84%
	②	0.0%	0.0%	21.3%	0.0%	100%	85%	92%
	③	0.0%	0.0%	3.8%	25.0%	87%	100%	93%

statistical processing of information involving human participation [46]. Typical values for precision and recall range around 85%, which compares not unfavourably to current state-of-the-art of 72% recall as reported in Ref. [50] for database co-change detection. We explicitly note, however, that method and evaluation differ between *op. cit.* and our work, which prevents an entirely straightforward comparison.

Consequently, we argue that our analysis method and pipeline provide an apt basis to derive sound statistical conclusions from real-world software engineering data.

3.3 Evolution scenarios

Figure 3 illustrates possible temporal evolution scenarios. **Scenario (a)** shows a linear commit history with a univariate `sql`-file change s_1 not accompanied by coupled code changes, neither directly in the same commit, nor in later commits. Most previous empirical studies on schema evolution [18, 22, 50, 60, 61, 64, 67] revolve around capturing nature and frequency of schema changes, assuming this simple scenario.

However, more disruptive schema modifications necessitate altering DB-relevant code. **Scenario (b)** shows a possible approach, where `sql`-file change (s_1) and the coupled code change (c_1) are committed together. **Scenario (c)** generalises to multivariate DB applications. Related changes to `sql`-files for different vendors are combined in a single commit, together with relevant code changes. This strategy is present in the subject projects, although real developer behaviour often leans more towards **scenario (d)**. It shows how a `sql`-file change can be temporally decoupled from related code changes: One code change happens in the same commit (lag 0), another one is delayed until two commits after (lag 2).

Scenario (e) is the most general case. `sql`-files for two vendors and dependent code change in different commits on different branches, which eventually merge with a common ancestor. Linearising commits [35, 70]) creates a well-defined temporal order (dashed commits), and induces a vendor/vendor co-change structure (*e.g.*, commits 0+4, counting from the left) as well as `sql`-file/code couplings (*e.g.*, commits 0+3, or 2+3).

4 HYPOTHESIS 1

We now target our first hypothesis: ***Database induced dependencies between artefacts can be automatically revealed by***

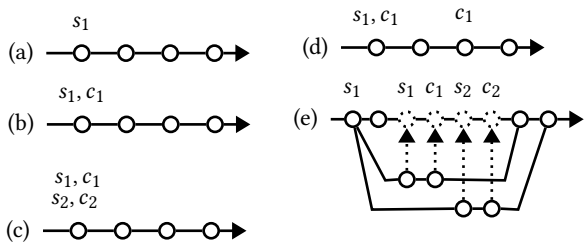

Figure 3: Different commit strategies (circle: commit). (a) Uncoupled univariate sql-file change; (b) univariate sql-file change coupled with a database code change in the same commit; (c) multivariate schema changes (sql-file/DBcode and vendor/vendor coupling in one commit); (d) univariate scenario with lag 0+2 sql-file/DB code couplings; (e) coupled changes with separate branches for two vendors.

semantically-enriched co-change analysis. So far, existing efforts [50] for coupling analysis between sql-files and the application code rely heavily on manual analysis to recognise DB-related code. Moreover, existing approaches ignore multivariate sql-files, even when the applications studied are multivariate.

We next quantify the couplings identified by DB-aware coupling analysis, focusing at one commit at-a-time. This allows us to distinguish the scenarios (a) through (c) from Figure 3.

Witness result distribution. sql-files may happen to be ignored in coupling analysis that is based on co-changes when they are not classified as programming language code. To estimate their possible effects in co-change approaches based on single commits, observe the distribution of commit content shown in Fig. 4: For each project, we classify commits into four groups, depending whether or not the commit affects a sql-file, and whether or not it contains DB-related code. Here, we consider all sql-files, and we will explore the effect of focusing only on those sql-files that contain the main schema declaration in our exploration of Hypothesis 2.

sql-files coupled with database related code are the most infrequent combination for most projects–changes to sql-files are usually not accompanied by database code changes, at least not within the same commit. More importantly, they are dominated by database code changes. In frequency-based evaluations of co-changes, the possible impact of sql-files is therefore substantially limited, and traditional coupling analysis is likely to regard them as "noise".

Database related code changes typically comprise 30%–40% of commits. Analyses without information on change semantics (database vs. non-database) are unable to capture this property: A considerable share of commits involves changes to one particular architectural element of the software stack, namely the database. We are not aware that this observation is reflected in previous work.

We conclude that the share of sql-files is essentially negligible on the single-commit level, yet considerable effort is spent on database-related changes for all subject projects. This is even visible in a simple analysis that utilises a window of size 1 (*i.e.*, lag 0).

Detected couplings. We next examine the dependencies observable for sql-files in more detail. Consider Table 3. For each project, we list the probability of a DB-related code change in the same commit (S/DBC). We further list the probability of a sql-file change for at least one *other* vendor within the same commit (V/V).

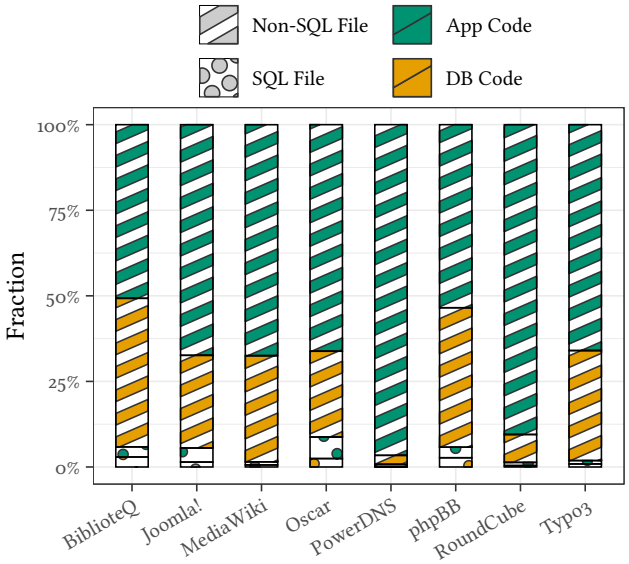

Figure 4: Distribution of change types in individual commits. Changes can affect sql-files or non-sql-files, and modify code related or not related to databases. Four combinations of colour and pattern are possible.

Table 3: Given a sql-file change: Probability of co-change within the same commit (lag 0) for DB-related code (S/DBC), and a sql-file for an alternative vendor (V/V).

Project	S/DBC	V/V	Project	S/DBC	V/V
BiblioteQ	50%	36%	Joomla!	26%	24%
MediaWiki	40%	13%	OSCAR EMR	28%	1%
phpBB	47%	24%	PowerDNS	18%	29%
Roundcube	32%	47%	TYPO3	47%	2%

For the column labeled "S/DBC", we are able to reproduce the core results from Qiu *et al.* [50] (within reason), who showed that in approx. 50% of all commits changing sql-files, we find DB-relevant code. We deem this a relevant contribution: (1) while both studies have an overlap in the analysed projects, there are differences, so the results are robust and generalisable. Further, (2) the results of Qiu *et al.* heavily rely on manual code inspection, allowing only for 100 commits with schema changes to be analysed, whereas we have analysed a total of over 6.8k changes to sql-files.

Regarding the probability of a co-change with the sql-file of an alternative vendor, we can observe that it is consistently lower (per project) than for DB-code. Notably, the probability is close to zero for OSCAR EMR and TYPO3. We provide an explanation as part of our discussion in Section 8.

Conclusion. sql-file changes are comparatively rare. This creates a bias for traditional code coupling analysis, which may regard database-induced dependencies as mere noise. Moreover, code coupling analysis must be made DB-aware, both in recognising sql-

files as relevant sources of dependencies, and in recognising DB-related code. We can confirm earlier results on DB-related code coupled with sql-files in the same commit, based on an automated classification, and a data pool larger by orders of magnitude.

Moreover, we have identified a so far unexplored class of database-induced dependencies that are specific to multivariate applications. These dependencies are less pronounced in comparison, but nevertheless significant. Thus, we can confirm Hypothesis 1.

5 HYPOTHESIS 2

We now challenge traditional coupling analysis based on relating changes within single commits. We hypothesise that ***dependencies between sql-files and database related code predominantly manifest themselves as long-range ripple effects.***

For each commit n containing a sql-file, we compute the lag operator $\hat{L}^{\infty}_{n,l}\vec{w}_{dbc}\vec{C}$ for all lags $l \in [0, 20]$. For each l, and resolved by project, Figure 5 plots the probability that a schema change is accompanied by database relevant code change within lag $[0, l]$ (naturally, all curves are monotonically increasing, as lag $k+1$ subsumes lag k, when counting from the same commit n). Different from H1, we now study lags greater than zero, and thus, a larger window of commits. Scenarios (d) and (e) in Fig. 3 represent typical situations relevant for this hypothesis.

We need to address one possible source of noise, namely the influence of sql-files that are only relevant for the operations, not the development team. Fig. 5 contains two curves for the projects that distinguish between these two types of sql-files: One represent a calculation that only considers all sql-files, while the other is restricted to main sql-files. A green line visualises the pointwise difference. Especially for the lower lags, we observe absolute differences of roughly 10%. The overall shape of the curves is independent of the measurement variant in good approximation, and the differences quickly vanish for higher lags.[3] This indicates that it is sufficient to consider all changes to sql-files without prior filtering (and even more importantly, without leveraging a-priori knowledge which sql-files declare the main database schema).

Secondly, consider how the probability of observing a co-change varies with increasing lag. Joomla!, MediaWiki, and OSCAR EMR start with 25% for lag 0, and rise to around 75% at lag 5. BiblioteQ, phpBB, and Typo3 start off slightly higher ground at 50% probability, but rise to more than 90% at lag 5. Nearly all subject projects reach almost 100% probability at lag 10. This means that established co-change dependency techniques that only consider lag 0 can miss the existence of co-changes by a factor of two to three probability-wise, and lose important structural information.

PowerDNS and RoundCube are apparent outliers. Manual inspection of the project commit histories reveals schema changes that correspond to scenario (a) from Fig. 3 and do not inject dependencies into the application code. Both projects frequently change indexes (this observation is quantitatively confirmed in Ref. [50] for Roundcube). As discussed in Section 2.3, such changes do not require updates in the application code because they concern only the physical, not the logical schema.[4] In contrast, logical changes such as renaming, adding, or deleting columns can be expected to be more disruptive, as we observe for the other subject projects.

Conclusion. Our analysis shows that considering non-zero lags is required for all projects, if we are to reliably capture changes to sql-files along with DB-relevant code changes. Traditional co-change analysis techniques, which are based on single commits, will not faithfully recover all existing couplings between sql-files and database code, due to the long-term ripple effects caused.

Again, we can confirm our hypothesis, which can be seen as quantitative confirmation of database engineering folklore of the database as a dependency magnet [4, 64]. Highlighting the importance of changes spread across commits is possible because our method allows us to provide of a semantic link between temporally remote commits.

6 HYPOTHESIS 3

We now explore the multivariate vendor/vendor couplings identified in Hypothesis 1, and how they behave with increasing lags. Our hypothesis is that ***dependencies caused by multivariate vendor/vendor couplings predominantly manifest themselves as long-range ripple effects.*** This is related to the previous hypothesis, but now addresses a different aspect of software variability.

For each commit n containing a sql-file, we compute the lag operator $\hat{L}^{1}_{n,l}\vec{w}_{v}\vec{C}$ for all lags $l \in [0, 20]$. Unlike for Hypothesis 2, we compute $\hat{L}^{1}$ instead of L^{∞}, because in a first step, we are interested in learning *how many* sql-files associated with different database vendors are encountered within a given window. In a second step, we characterise a lag l window as either multivariate (two or more vendors), or as univariate (only one vendor throughout).[5]

Figure 6 visualises the results, by project: The total amount of witnessed vendor-vendor couplings per lag l is split into univariate and multivariate sql-file changes, and the relative fractions are shown by yellow triangles (multivariate), and grey points (univariate): For instance, 36% of all sql-file changes at lag 0 in BiblioteQ address multiple database vendors, and 64% only concern a single vendor (we list explicit values for lag 0 in Table 3). Standard co-change analyses would therefore conclude that the univariate case dominates the multivariate case. However, the distribution flips when higher lags are taken into account; eventually, the relative fraction of multivariate cases is about 2/3, where the univariate changes only comprise every third schema change.

While the ratios between univariate and multivariate sql-file changes varies among subject projects, all except OSCAR EMR and Typo3 at least double the fraction of multivariate changes by lag 20.

Conclusion. Overall, we observe a distinct effect of long-range multivariate couplings, yet not as influential as for H2. Regardless of the specific fraction, multivariate sql-file changes are well represented in our sample of subject projects. As for H2, in general,

[3] We remark that the observation of larger co-change probabilities for main sql-files is consistent with a DevOps approach: Main sql-file changes occur in the "Dev" phase, whereas incremental changes address modifications to already deployed database instances, and relate to the "Ops" phase, with less need for code changes because the actual development has already taken place in the "Dev" phase.

[4] This so-called data independence between the physical and logical schema is a highly desirable feature of databases [52].

[5] The rationale rests on the consequence that multivariate modifications imply: When the multivariate case must be frequently supported in a project, providing appropriate tool assistance may be required. However, the decision to build or deploy an appropriate tool does not depend on the fact how often a specific vendor is usually addressed within an interval, but the fact that the multivariate case needs to be handled frequently at all. Similar thinking applies to other scenarios that need to distinguish between univariate and multivariate cases.

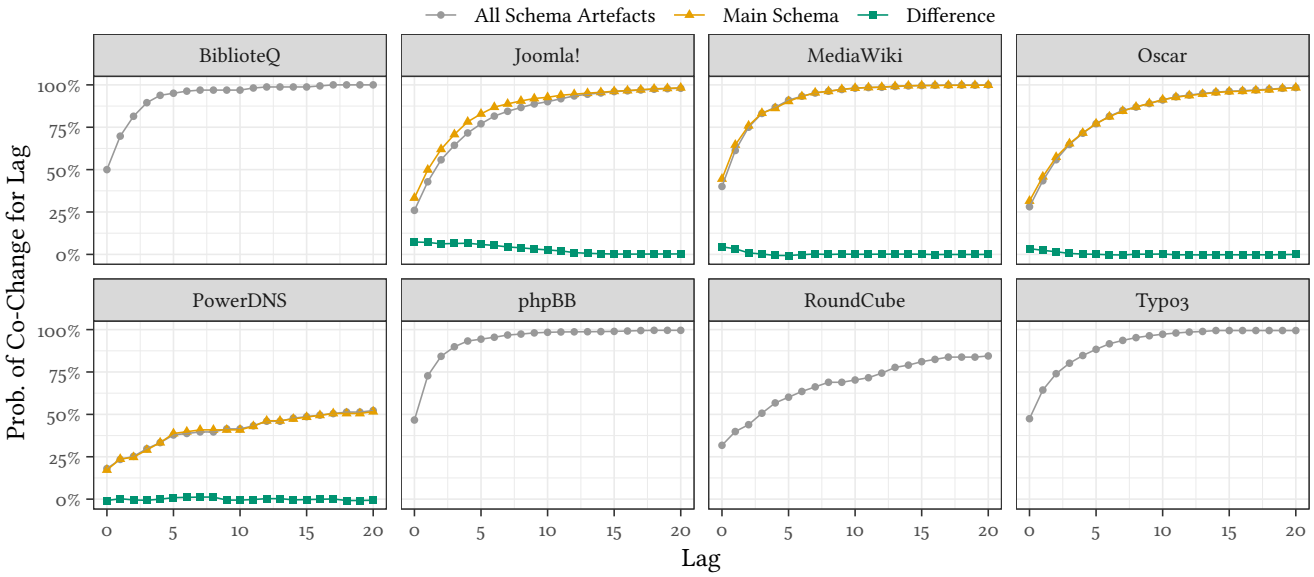

Figure 5: Lag coupling analysis for H1. Each data point summarises the probability of at least one co-change of a `sql`-file and database-related code within the given lag. Gray, round data points consider all `sql`-files. Ochre, triangle data points consider only files where the main schema is declared. The difference between both variants (green, square) never exceeds 10%.

these results show that for most projects analysed, there is a market for designing tools that assist with vendor/vendor couplings.

Apparently, we can group the subject projects: (1) For BiblioteQ and RoundCube, the multivariate case exceeds the univariate case within a few lag increments. (2) OSCAR EMR and TYPO3 are marked outliers, with the univariate line consistently high, and the multivariate line consistently low. (3) In the remaining four projects, it takes the multivariate line a lag of 20 or more to close in on the univariate line. We discuss the implications in Section 8.

7 THREATS TO VALIDITY

This section discusses possible threats to validity, as well as our counter-measures. Some generic threats common to repository mining projects (such as implications of linearising a nonlinear commit history, or relying on the correctness of underlying, widely used tools), are shared with comparable studies (*e.g.*, Refs. [35, 50, 70]), and not discussed here.

Computation of DB-witnesses. The largest threat to validity is our automated computation of DB-witnesses, based on a-priori knowledge of the folder structure, naming conventions, and database-specific APIs. Yet as we show in our results for Hypothesis 1, we were able to confirm that our approach is indeed sound and robust. We thus deem this threat as only marginal.

Computing vendor witnesses. We consider all changes in `sql`-files for coupling analysis. In contrast, Qiu *et al.* [50] perform thorough data cleaning to identify the valid schema changes (and to ignore changes that merely pretty-print, or fix typos in comments, etc.). Moreover, we do not distinguish between DDL changes (*e.g.*, CREATE TABLE statements) from DML changes (*e.g.*, INSERT INTO statements). The latter populate the database with initial data, and do not actually change the schema. Our motivation is that changes to the initial data do affect the database component in the software, and can very well have dependencies into the remaining application code. As our overall theme is to study the database as a dependency magnet, we deem this decision reasonable.

Despite a simpler data preparation phase, for lag-0, we are able to reproduce the main result of Qiu *et al.*, namely that for approximately 50% of schema-changes, the DB-relevant, dependent code is changed within the same commit. Thus, we regard the threat imposed as acceptable.

Lag direction. We consider only lags equal to or greater than zero, but never negative. This is a conscious choice: Empirical studies on the frequencies of schema modification operations [14, 50, 67]), agree that additions outweigh the removal of tables and columns by far. A column can be added to a `sql`-file before adapting dependent code, without breaking the application. When a column is removed, any references must be removed from the dependent code *before*, otherwise the application will break. Additions are captured by positive lags, removals require negative ones, with the `sql`-file change as the anchor. Since additions outweigh deletions, we consider this restriction fungible, given the simplicity gains.

Generalisability. One possible threat is that our results might not generalise to projects other than those studied here. However, we have chosen well-known database applications, most of which are also featured in other studies on schema evolution. They cover a certain range of use cases (from content management systems to a DNS server), and involve different programming languages (*e.g.*, C++, Java, PHP). We are therefore confident that our analysis indeed allows to derive generalisable conclusions.

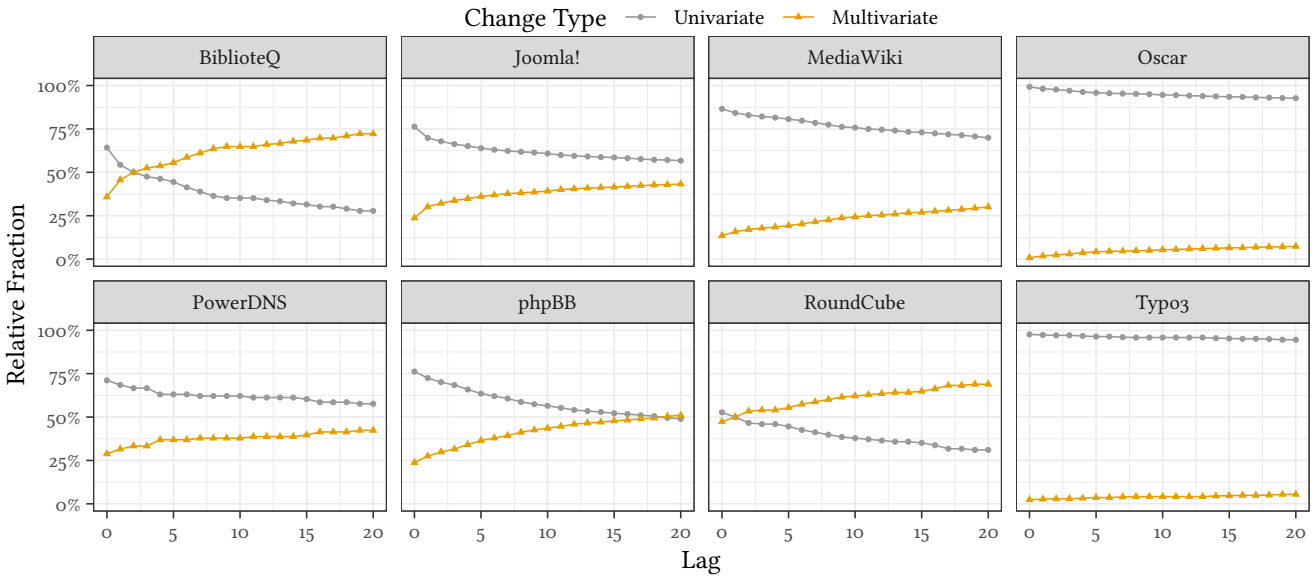

Figure 6: Lag coupling analysis for H2. Each data point summarises the relative fraction of commits with a co-change of a `sql`-file with a `sql`-file for an alternative vendor. Gray, round data points denote univariate commits; ochre triangle data points denote multivariate commits.

8 DISCUSSION

In this paper, we have tried to better understand the influence of hidden, database-induced dependencies and their ripple effects on software development and architecture. Our analysis confirms different types of dependencies induced by databases: Couplings between `sql`-files and database code, and couplings between `sql`-files that represent essentially the same information about physical and logical schemas, but for different vendors. While the former have received initial treatment in the scientific literature, but are still underexplored, the latter are entirely unexplored to the best of our knowledge. We found that both types of dependencies occur frequently in large real-world systems, and cause dependency ripples, whose consequences must be tackled by software developers.

For identifying the relationships, we find it instrumental to augment existing coupling analysis mechanisms with a domain specific semantic understanding. This allows us to include structural properties beyond a pure textual level in the analysis. Typically, a quarter of all commits in the subject projects perform database related changes. This seems to underline the importance of going from a purely factual level ("there exists a connection between A and B") in co-change analyses to an understanding of what a given co-change is actually about.

As we have remarked in Section 2, there is comparatively little, but long-standing research activity on the subject of schema evolution and the coupling between schemas and code in data-intensive applications. This shows the ongoing interest in viable solutions. Yet when the long-standing pattern recommendations on how to decouple databases and applications are contrasted with the strong couplings identified in our work, this seems to indicate that best practices such as introducing a database access layer do not suffice. We find the problem does not yet receive the attention it deserves from both an applied tool-centric point of view and from the perspective of research. Perhaps the involvement of two mostly disjoint communities plays a role. Nonetheless, with the growing influence of data science and the ability to routinely analyse huge datasets, we expect that impact and importance of the problem will increase in the future, and that software architectural solutions that provide an effective decoupling are required. We identify two promising strategies:

(1) Consciously weigh benefits of multivariate database support against the increase in complexity and efforts (obviously, awareness about the effective costs of multivariate database support is a precondition, which we hope to raise with this paper). OSCAR EMR ended support for vendor Oracle in 2011, which manifests in near-to-no multivariate couplings in Figure 6.
(2) Introduce an appropriate abstraction layer in form of a *schema manager*. This might seem closely related to database access layers as recommended for decades, but is a different architectural pattern, where vendor-specific schemas are generated from a common model, rather than synchronised manually. In 2016, TYPO3 introduced the Doctrine schema manager [1]. We believe this explains why the behaviour in Figure 6 resembles that of a univariate application.

We argue that the latter strategy should enter the curriculum of software architecture and engineering textbooks, which currently treat couplings between databases and application code only cursorily. It is interesting to observe that relief came from a "Dev" tool, closely related to ideas of product lines and generative software engineering, while the database community has so far mostly focused on "Ops" tools and approaches, as noted in the introduction.

The current surge of interest in NoSQL databases is fuelled by the desire to eliminate the existence of fixed database schemas,

which would also reduce dependencies between schema and database code. Unfortunately, the approach works mostly for the "Ops" team in DevOps scenarios, but provides only temporary relief for developers because similar problems are known to eventually resurface, *e.g.* for NoSQL stores [58] or polystores [24].

All in all, we find that augmenting the established and proven ideas of co-change dependency analysis with domain-specific semantic understanding of the changes in commits shed light on previously unobserved, but relevant aspects of practical software development. We speculate that extensions to other domain areas may open interesting follow-up research opportunities.

9 CONCLUSION

In this paper, we revisit the important problem of database evolution and the ripple effects caused in the application stack due to co-evolution of database relevant application code.

We show that the problem is more complex and multi-faceted than captured by metrics applied in the past. Since this research problem lies in the no man's land between software engineering and database research, existing solutions from the respective research communities will not be sufficient; as communities, we will need to join forces to solve these problems together.

REFERENCES

[1] [n.d.]. Doctrine Schema Manager. https://www.doctrine-project.org/.

[2] [n.d.]. Scitools Understand toolkit. https://www.scitools.com/.

[3] Scott Ambler. 2003. *Agile Database Techniques: Effective Strategies for the Agile Software Developer.* Wiley Publishing.

[4] Scott W. Ambler and Pramodkumar J. Sadalage. 2006. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley Professional.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[6] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.

[7] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The Promises and Perils of Mining Git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR '09)*. IEEE Computer Society, 1–10. https://doi.org/10.1109/MSR.2009.5069475

[8] Sue Black. 2001. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 13, 4 (2001), 263–279. https://doi.org/10.1002/smr.233

[9] Dimitri Brainninger, Wolfgang Mauerer, and Stefanie Scherzinger. 2020. Replicability and Reproducibility of a Schema Evolution Study in Embedded Databases. In *Advances in Conceptual Modeling - ER Workshops (Lecture Notes in Computer Science, Vol. 12584)*, Georg Grossmann and Sudha Ram (Eds.). Springer, 210–219.

[10] Tyson R. Browning. 2016. Design Structure Matrix Extensions and Innovations: A Survey and New Opportunities. *IEEE Transactions on Engineering Management* 63, 1 (2016), 27–52.

[11] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing.* Wiley.

[12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing.

[13] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work* (Banff, Alberta, Canada) *(CSCW '06)*. Association for Computing Machinery, New York, NY, USA, 353–362. https://doi.org/10.1145/1180875.1180929

[14] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens H. Weber. 2015. Understanding database schema evolution: A case study. *Sci. Comput. Program.* 97 (2015), 113–121. https://doi.org/10.1016/j.scico.2013.11.025

[15] Anthony Cleve and Jean-Luc Hainaut. 2006. Co-transformations in Database Applications Evolution. In *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005.* Springer Berlin Heidelberg, 409–421. https://doi.org/10.1007/11877028_17

[16] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the Database Schema Evolution Process. *The VLDB Journal* 22, 1 (2013), 73–98.

[17] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *VLDB J.* 22, 1 (2013), 73–98.

[18] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. 2008. Schema evolution in Wikipedia: Toward a Web Information System Benchmark. In *Proceedings of the Tenth International Conference on Enterprise Information Systems.* 323–332.

[19] Michael de Jong, Arie van Deursen, and Anthony Cleve. 2017. Zero-Downtime SQL Database Schema Evolution for Continuous Deployment. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Computer Society, 143–152. https://doi.org/10.1109/ICSE-SEIP.2017.5

[20] J. Delplanque, A. Etien, N. Anquetil, and O. Auverlot. 2018. Relational Database Schema Evolution: An Industrial Case Study. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 635–644. https://doi.org/10.1109/ICSME.2018.00073

[21] L. Deruelle, M. Bouneffa, N. Melab, and H. Basson. 2001. A change propagation model and platform for multi-database applications. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* 42–51. https://doi.org/10.1109/ICSM.2001.972710

[22] Konstantinos Dimolikas, Apostolos V. Zarras, and Panos Vassiliadis. 2020. A Study on the Effect of a Table's Involvement in Foreign Keys to its Schema Evolution. In *Conceptual Modeling*, Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr (Eds.). Springer International Publishing, 456–470.

[23] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesus M. Gonzalez-Barahona. 2018. Perceval: Software Project Data at Your Will. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. ACM, 1–4. https://doi.org/10.1145/3183440.3183475

[24] Jérôme Fink, Maxime Gobert, and Anthony Cleve. 2020. Adapting Queries to Database Schema Changes in Hybrid Polystores. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '20)*. IEEE, 127–131. https://doi.org/10.1109/SCAM51674.2020.00019

[25] Martin Fowler. 2012. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch.* Addison-Wesley.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional.

[27] S. K. Gardikiotis and N. Malevris. 2006. DaSIAn: A Tool for Estimating the Impact of Database Schema Modifications on Web Applications. In *IEEE International Conference on Computer Systems and Applications, 2006.* 188–195. https://doi.org/10.1109/AICCSA.2006.205088

[28] Spyridon K. Gardikiotis and Nicos Malevris. 2009. A two-folded impact analysis of schema changes on database applications. *Int. J. Autom. Comput.* 6, 2 (2009), 109–123. https://doi.org/10.1007/s11633-009-0109-4

[29] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. 2019. Git2net: Mining Time-Stamped Co-Editing Networks from Large Git Repositories. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 433–444. https://doi.org/10.1109/MSR.2019.00070

[30] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference and prediction* (2 ed.). Springer.

[31] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. 2017. Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In *ACM SIGMOD*. 1101–1116.

[32] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. 2009. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*. 30–39. https://doi.org/10.1109/ICPC.2009.5090025

[33] Amornrat Jaimoon and Taratip Suwannasart. 2019. Impact Analysis of Database Schema Changes on Hibernate Source Code and Test Cases. In *Proceedings of the 2019 3rd International Conference on Software and E-Business (ICSEB 2019)*. ACM, 120–123. https://doi.org/10.1145/3374549.3374579

[34] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. 2017. Evolutionary trends of developer coordination: a network approach. *Empirical Software Engineering* 22 (2017), 2050–2094. https://doi.org/10.1007/s10664-016-9478-9

[35] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. 2015. From Developer Networks to Verified Communities: A Fine-Grained Approach. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 563–573.

[36] Hassan Khosravi and Recep Colak. 2009. Exploratory Analysis of Co-Change Graphs for Code Refactoring.. In *Canadian Conference on AI* (2009-05-18) *(Lecture Notes in Computer Science, Vol. 5549)*, Yong Gao and Nathalie Japkowicz (Eds.). Springer, 219–223. http://dblp.uni-trier.de/db/conf/ai/ai2009.html#KhosraviC09

[37] Martin Kleppmann. 2016. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly.

[38] Max Kuhn. 2020. *caret: Classification and Regression Training.* https://CRAN.R-project.org/package=caret R package version 6.0-86.

[39] Dien-Yen Lin and Iulian Neamtiu. 2009. Collateral Evolution of Applications and Databases. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops (IWPSE-Evol '09)*. ACM, 31–40. https://doi.org/10.1145/1595808.1595817

[40] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk. 2015. How do Developers Document Database Usages in Source Code?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 36–41. https://doi.org/10.1109/ASE.2015.67

[41] S Martin-Haugh, S Kluth, R Seuster, S Snyder, E Obreshkov, S Roe, P Sherwood, and G A Stewart. 2017. C++ software quality in the ATLAS experiment: tools and experience. *Journal of Physics: Conference Series* 898 (oct 2017), 072011. https://doi.org/10.1088/1742-6596/898/7/072011

[42] A. Maule, W. Emmerich, and D. Rosenblum. 2008. Impact analysis of database schema changes. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 451–460. https://doi.org/10.1145/1368088.1368150

[43] Tom Mens and Serge Demeyer (Eds.). 2008. *Software Evolution.* Springer. https://doi.org/10.1007/978-3-540-76440-3

[44] T. Menzies, L. Minku, and F. Peters. 2015. The Art and Science of Analyzing Software Data; Quantitative Methods. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 959–960. https://doi.org/10.1109/ICSE.2015.306

[45] L. Meurice, C. Nagy, and A. Cleve. 2016. Detecting and Preventing Program Inconsistencies under Database Schema Evolution. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 262–273. https://doi.org/10.1109/QRS.2016.38

[46] Ferenc Moksony. 1999. Small Is Beautiful: The Use and Interpretation of R^2 in Social Research. *Szociologiai Szemle* (01 1999), 130–138.

[47] Hyun Jin Moon, Carlo Curino, MyungWon Ham, and Carlo Zaniolo. 2009. PRIMA: Archiving and querying historical data with evolving schemas. In *ACM SIGMOD*. 1019–1022.

[48] Ludvig Renbo Olsen and Benjamin Hugh Zachariae. 2021. *cvms: Cross-Validation for Model Selection.* https://CRAN.R-project.org/package=cvms R package version 1.2.1.

[49] S. Panichella, G. Bavota, M. D. Penta, G. Canfora, and G. Antoniol. 2014. How Developers' Collaborations Identified from Different Sources Tell Us about Code Changes. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 251–260. https://doi.org/10.1109/ICSME.2014.47

[50] Dong Qiu, Bixin Li, and Zhendong Su. 2013. An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications. In *Proc. ESEC/FSE'13*.

[51] R Core Team. 2021. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. https://www.R-project.org/

[52] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (3 ed.). McGraw-Hill, Inc.

[53] R. P. Reyes Ch., O. Dieste, E. R. Fonseca C., and N. Juristo. 2018. Statistical Errors in Software Engineering Experiments: A Preliminary Literature Review. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1195–1206. https://doi.org/10.1145/3180155.3180161

[54] Marko Rosenmüller, Christian Kästner, Norbert Siegmund, Sagar Sunkle, Sven Apel, Thomas Leich, and Gunter Saake. 2009. SQL á la Carte – Toward Tailor-made Data Management. In *In Datenbanksysteme in Business, Technologie und Web – Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme, volume P-144 of GI-Edition – LNI.*

[55] Neeraj Sangal, Ev Jordan, and Vineet Sinha. 2005. Using dependency models to manage complex software architecture. *ACM Sigplan Notices* (2005).

[56] Stephen R. Schach, Bo Jin, Liguo Yu, Gillian Z. Heller, and A. Jefferson Offutt. 2003. Determining the Distribution of Maintenance Categories: Survey versus Measurement. *Empirical Software Engineering* 8, 4 (2003), 351–365. http://dblp.uni-trier.de/db/journals/ese/ese8.html#SchachJYHO03

[57] Stefanie Scherzinger, Wolfgang Mauerer, and Haridimos Kondylakis. 2021. DeBinelle: Semantic Patches for Coupled Database-Application Evolution. In *Proceedings of the 37th IEEE International Conference on Data Engineering*. Demo paper.

[58] Stefanie Scherzinger and Sebastian Sidortschuck. 2020. An Empirical Study on the Design and Evolution of NoSQL Database Schemas. In *Conceptual Modeling - 39th International Conference (ER) (Lecture Notes in Computer Science, Vol. 12400)*. Springer, 441–455.

[59] Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. 2016. From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open Source Software projects. *Empir. Softw. Eng.* 21, 2 (2016), 642–683.

[60] D. Sjøberg. 1993. Quantifying schema evolution. *Information & Software Technology* 35, 1 (1993), 35–44. https://doi.org/10.1016/0950-5849(93)90027-Z

[61] Ioannis Skoulis, Panos Vassiliadis, and Apostolos V. Zarras. 2015. Growing Up with Stability. *Information Systems* 53, C (2015), 363–385.

[62] Panos Vassiliadis. 2021. Profiles of Schema Evolution in Free Open Source Software Projects. In *Proceedings of the 37th IEEE International Conference on Data Engineering*.

[63] Panos Vassiliadis, Michail-Romanos Kolozoff, Maria Zerva, and Apostolos V. Zarras. 2019. Schema evolution and foreign keys: A study on usage, heartbeat of change and relationship of foreign keys to table activity. *Computing* 101, 10 (2019), 1431–1456.

[64] Panos Vassiliadis and Apostolos V. Zarras. 2017. Schema Evolution Survival Guide for Tables: Avoid Rigid Childhood and You're En Route to a Quiet Life. *J. Data Semant.* 6, 4 (2017), 221–241. https://doi.org/10.1007/s13740-017-0083-x

[65] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. In *Proc. PLDI 2019*. 286–300. https://doi.org/10.1145/3314221.3314588

[66] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. 2009. Design Rule Hierarchies and Parallelism in Software Development Tasks.. In *ASE*. IEEE Computer Society, 197–208. http://dblp.uni-trier.de/db/conf/kbse/ase2009.html#WongCVSS09

[67] Shengfeng Wu and Iulian Neamtiu. 2011. Schema Evolution Analysis for Embedded Databases. In *2011 IEEE 27th International Conference on Data Engineering Workshops*.

[68] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. 2009. Data Mining for Software Engineering. *IEEE Computer* 42, 8 (August 2009), 35–42.

[69] S. S. Yau, J. S. Collofello, and T. MacGregor. 1978. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.* 60–65. https://doi.org/10.1109/CMPSAC.1978.810308

[70] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, 563–572.