An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications

Dong Qiu Bixin Li
Southeast University, China
{dongqiu, bx.li}@seu.edu.cn

Zhendong Su
University of California, Davis, USA
su@cs.ucdavis.edu

ABSTRACT

Modern database applications are among the most widely used and complex software systems. They constantly evolve, responding to changes to data, database schemas, and code. It is challenging to manage these changes and ensure that everything co-evolves consistently. For example, when a database schema is modified, all the code that interacts with the database must be changed accordingly. Although database evolution and software evolution have been extensively studied in isolation, the co-evolution of schema and code has largely been unexplored.

This paper presents the first comprehensive empirical analysis of the co-evolution of database schemas and code in ten popular large open-source database applications, totaling over 160K revisions. Our major findings include: 1) Database schemas evolve frequently during the application lifecycle, exhibiting a variety of change types with similar distributions across the studied applications; 2) Overall, schema changes induce significant code-level modifications, while certain change types have more impact on code than others; and 3) Co-change analyses can be viable to automate or assist with database application evolution. We have also observed that: 1) 80% of the schema changes happened in 20-30% of the tables, while nearly 40% of the tables did not change; and 2) Referential integrity constraints and stored procedures are rarely used in our studied subjects. We believe that our study reveals new insights into how database applications evolve and useful guidelines for designing assistive tools to aid their evolution.

Categories and Subject Descriptors

H.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*

General Terms

Language, Measurement

Keywords

Co-evolution, database application, empirical analysis

ESEC/FSE '13, August 18–26, 2013, Saint Petersburg, Russia

1. INTRODUCTION

A database application is a software system that collects, manages, and retrieves data, which are typically stored in a database managed by a database management system (DBMS) and organized *w.r.t.* database schemas. For example, most online services are powered by database applications. Wikis, social networking systems (SNS), Web-based content management systems (CMS), mailing systems, enterprise resource planning systems (ERP) are all database applications. As Figure 1 illustrates, a program needs to obey the structure of the data organization defined by a schema when it accesses the data. Namely, a schema is a mediator that manages the interactions between code and data, bridging their gap.

Software systems are subject to continuous evolution due to modified system requirements; database applications are no exception. Cleve *et al.* [5] observe that little work exists on understanding the evolution of database applications considering both data and code. Different from traditional applications, the evolution of database applications is more complex. For example, consider a system that uses a table USER to store both user authentication information and other personal data. Now the system requirements change, and the system needs to store user authentication information and personal data separately. Thus, the original table USER must be split into two new tables, say USER_LOGIN and USER_DETAILS. Data and application code must be synchronized to be consistent with the new schemas. First, the original data organization should be migrated to the new one defined by USER_LOGIN and USER_DETAILS. Second, the original application code that accesses data in USER must be modified to correctly access the newly organized data in USER_LOGIN and USER_DETAILS.

Figure 1 illustrates these two types of co-evolution in database applications: 1) data co-evolve with schemas, and 2) code co-evolves with schemas. The first type of co-evolution involves three main tasks: i) predicting and estimating the effects before the proposed schema changes are performed; ii) rewriting the existing DBMS-level queries to work on the new schemas; and iii) migrating data to the new schemas. The second type involves two main tasks: i) evaluating the cost of reconciling the existing code *w.r.t.* the new schemas before any schema changes; and ii) locating and modifying all impacted code regions after applying the schema changes.

The database community has addressed the first co-evolution problem gracefully to support automatic data migration and DBMS-level query rewriting to operate on the new schemas [6,7]. However, little work has considered the second co-evolution problem. Its difficulties are twofold. First, query updates and data migration for the first problem are done by DB Administrators (DBA), who have the domain knowledge. In contrast, the application developers, who have different level of database knowledge, may not precisely capture the whole evolution process of the database structure. In

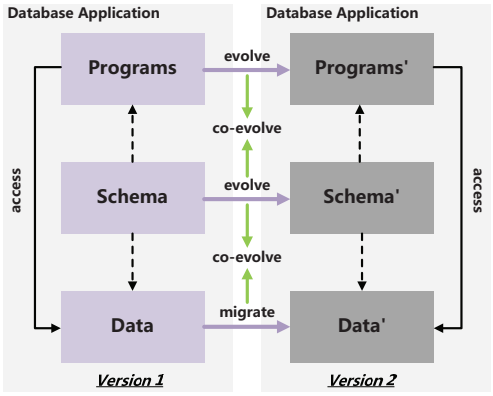

Figure 1: The evolution of database applications.

particular, they may not fully grasp what the DBA intends through a schema change. When the schema is modified, if application developers do not understand why and how the schema changes, they can have difficulties in correctly evolving the application code. Second, schema changes, query updates, and data migration are straightforward as they are done at the same language level, typically using Data Definition Language (DDL) and Data Manipulation Language (DML). In contrast, database schema changes and application code changes are at different levels. This is a much more challenging problem because how schema changes impact code is not as direct.

In this paper, we focus on the second problem and perform a large-scale empirical study on ten popular database applications to gain insights into how schemas and application code co-evolve. In particular, we study the applications' long-time evolution histories from their respective repositories to understand whether database schemas evolve frequently and significantly, how schemas evolve and how they impact application code. Furthermore, we present guidelines for developing automated tool to aid schema and code co-evolution based on our observations from the study. To the best of our knowledge, this is the first work that attempts to empirically understand how application programs co-evolve with the schemas in large, popular database applications.

The remainder of this paper is organized as follows. Section 2 describes our methodology for the study, including its context, research questions and the process for extracting information needed for the study. Section 3 presents our detailed results, followed by Section 4 that further discusses the results. Next, we discuss possible threats to validity (Section 5) and survey related work (Section 6). Finally, Section 7 concludes with a discussion of future work.

2. METHODOLOGY

This section presents basic information on the ten database applications used in our study, the three research questions we consider, and the process we use to perform the study.

2.1 Context Description

We selected 10 open-source database applications from various domains, including gallery management, project management, CMS, Wiki, shopping cart, webmail system and ERP. They are popular both among developers and users. For example, both *Joomla!* and *PrestaShop* won the *Open Source Award*[1] in 2011. Table 1's first 3 columns report each application's basic description and popularity.

For all ten applications, *Subversion* (SVN) [16] was selected as their default version control system because SVN is one of the most popular version control systems in the open-source community. Table 1 also summarizes the applications' evolution histories extracted from their respective SVN repositories. The 4th, 5th and 6th columns list, for each project, its development cycle, the number of stable releases and revision range in the selected lifecycle that we consider. Most projects (9/10) are still active and have frequent updates. Only *Dotproject* is no longer updated. For *e107* and *TYPO3*, we only selected one of their finished development trunks. The 7th column shows each project's total number of revisions. *Mediawiki*'s revisions form only a part of its revision range since its repository also hosts other projects, thus we kept only those revisions for *Mediawiki*. The 8th and 9th columns show two ranges for each project respectively: one for schema size (the least and greatest) and the other for application code size (the smallest and largest). The ranges were computed based on information from the 5th column. We use the number of tables to measure schema size and lines of code (LoC) to measure code size.

[1] `http://www.packtpub.com/open-source-awards-home`

2.2 Research Questions

At a high-level, our study aims to answer how database schemas and program co-evolve in database applications. To this end, we designed three specific research questions for investigation:

*RQ*1: *How frequently and extensively do database schemas evolve?* This RQ helps answer how often and how much schemas change across different versions to understand whether they intensively evolve during an application's development and maintenance process.

*RQ*2: *How do database schema evolve?* This RQ helps analyze all possible schema changes in database applications to understand what schema change types usually occur in practice. Furthermore, we are interested in the distribution of schema changes *w.r.t.* schema change type to see whether some change types appear more frequently than others.

*RQ*3: *How much application code has co-changed with a schema change?* This RQ helps quantify schema changes' real impact on application code. Using change history from the repository, we associate changes in source lines with schema changes to *estimate* their impact on code. We are also interested in whether certain schema change types tend to have more impact on code than others.

In addition, based on the answers to the three above RQs, if indeed schemas evolve frequently and extensively and schema changes can significantly impact code, it should be important to develop tools to aid developers in evolving database applications. From our results of empirical analysis, can we provide some evidences or guidelines on helping people efficiently evolve database applications?

2.3 Analysis Process

We now describe the steps we use to extract the necessary information from project repositories.

Locate schema files The first step extracts the schema files. We observe that most schema files have the `.sql` suffix although some projects (*e.g. e107*) specify schema information using embedded SQL statements in `PHP` source files. To ascertain that we do not omit any schema files, we manually trace the schema files even if their locations or names have been modified.

Extract DB revisions The second step identifies *DB revisions*, which are revisions (commits) that contain modifications to schema files. In SVN, we can easily retrieve the paths of all changed files in any revision — if a schema file is among the changed files of revision *i*, we say *i* is a DB revision.

Extract valid DB revisions Once we have extracted the DB revisions in each project, we need to filter those containing *only*

Table 1: The ten studied database applications and their evolution history.

Project	Description	D/L(M)	Life Cycle	# Releases	Revision Range	# Revisions	# Tables	LoC(K)	Changed LoC(M)
Coppermine	Web Gallery	6.8	09/03 ∼ 01/12	4	4 ∼ 8,307	8,304	8 ∼ 22	27 ∼ 300	1.86
Dotproject	Project Mgmt.	1.4	10/01 ∼ 05/07	7	2 ∼ 4,960	4,959	15 ∼ 63	8 ∼ 150	0.46
e107	Enterprise CMS	1.9	04/04 ∼ 01/11	2	4 ∼ 12,063	12,060	33 ∼ 30	36 ∼ 150	0.96
Joomla!	CMS	30.0	09/05 ∼ 03/12	5	3 ∼ 22,934	22,932	35 ∼ 61	10 ∼ 250	3.57
Mediawiki	Wiki	1.5	01/02 ∼ 01/12	18	48 ∼ 107,887	41,792	3 ∼ 51	3 ∼ 880	5.91
PrestaShop	Online Store	2.2	12/08 ∼ 01/12	10	1 ∼ 13,863	13,863	113 ∼ 157	11 ∼ 230	1.21
RoundCube	Webmail	2.0	10/05 ∼ 03/12	7	13 ∼ 5,755	5,743	5 ∼ 12	20 ∼ 120	0.76
Tikiwiki	Wiki/CMS	1.0	10/02 ∼ 03/12	19	3 ∼ 40,195	40,193	20 ∼ 242	10 ∼ 1,240	10.79
TYPO3	Enterprise CMS	7.2	10/03 ∼ 01/11	14	20 ∼ 10,200	10,181	10 ∼ 18	78 ∼ 440	3.75
webERP	Business Mgmt.	0.4	02/04 ∼ 02/12	10	2 ∼ 4,888	4,887	63 ∼ 122	36 ∼ 210	1.25

unrelated schema changes we are not interested in, *i.e.*, we need to select only the *valid DB revisions* for further analysis. We define a valid DB revision as a DB revision that does *not only* include the following types of schema changes:

(1) *Syntax change*. There are two main kinds of syntax changes: *optional syntactic structure conversion* and *syntax error fix*. Most syntax descriptions in different implementations of DDL have several options, which leads to the first kind of syntax changes. This situation often occurs when multiple developers work on the same code, but favor different syntax. For syntax errors, we have observed that redundant commas and quotation marks are most common syntax errors in DDLs examining the revision histories of the ten projects. Figures 2a and 2b show concrete examples for the two cases.

(2) *Comment change*. A comment change refers to a modification of a schema file's comments.

(3) *Format change*. Format changes have three main types: i) adjusting the position of a table in a schema file (*e.g.*, ordering the tables alphabetically *w.r.t.* table names); ii) adjusting the position of columns in a table (*e.g.*, moving a newly-added column from the very end to a different position, perhaps for better understanding); and iii) traditional formatting (*e.g.*, indentation, adding or deleting blank lines and whitespace). Although a format change has no effect on the content of a schema file, it can be detected by text differencing algorithms from the repository, and the corresponding revision is considered a DB revision.

(4) *Data-sensitive change*. A data-sensitive change refers to a modification of the data stored in the database. Mostly, DML is used for inserting, deleting and updating data in a database. Some projects, such as *Joomla!*, put DML and DDL in the same file, which leads to some DB revisions contain only data-sensitive changes. Since this work focuses on schema/code co-evolution, we do not consider data-sensitive changes.

(5) *DBMS-related change*. DBMS-related changes are mainly caused by version migration of DBMS used in database applications. Figure 2c shows an example where the change was made to satisfy the upgrade of the *MySQL* version.

(6) *System-related change*. System-related changes refer to modifications that are irrelevant for schema changes, but caused by the implementation of database applications. Figure 2d shows an example where newly added comment /*$wgDBprefix*/ was for runtime replacement by system programs to avoid name conflicts if multiple versions of database coexisted.

(7) *Rollback*. Suppose a schema file evolves from revision *i-1* to *i*, and later is recovered to revision *i-1* in revision *j*. If revision *j* is judged as a rollback of revision *i*, we filter both *i* and *j* as unrelated DB revisions. Identifying rollbacks needs both the schema change history and log information in the repository.

Extract atomic changes After having identified the valid DB revisions for each project, we extract all schema changes by *manually*

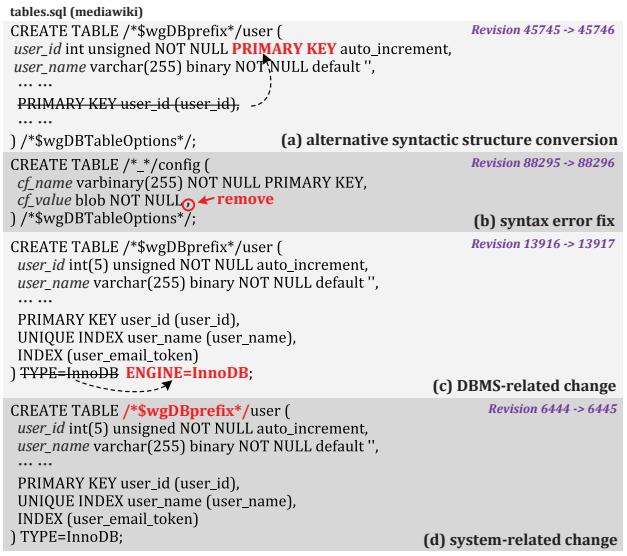

Figure 2: Examples of invalid DB revisions.

comparing schema files of contiguous valid DB versions. That is, we try to understand how schemas evolve semantically by examining the textual differences between two revisions and the relevant log messages. Although many tools, such as *mysqldiff* [15] used in [11], support difference extraction between two schema versions, they have the following disadvantages. First, only syntax-level schema changes can be obtained; semantic-level information that how schema actually evolve is omitted in the automatic analysis process. Consider the example in Figure 3. *mysqldiff*, running on revisions 4924 and 4925 of *Mediawiki*, outputs that column user_right is deleted from table user, and a new table user_rights has been created. The tool does not recognize the relationship between the deleted column user_rights and the added table user_rights. A better interpretation of this evolution is that table user is split into two sub-tables user and user_right. This precise semantic information is quite important since it can guide us better understand the code-level changes. Second, these tools are incapable of extracting the differences if any schema file contains syntax errors or system-related code (also see our earlier discussions on *syntax change* and *system-related change*) since they need to execute the schema scripts in a database engine to create concrete tables and relations. We noticed many syntax and system-related changes in the ten projects, making it impossible to generate precise schema changes for all revisions.

Before extracting schema changes, we need to provide a category of schema change types. Ambler *et al.* [1] summarized all possible schema changes during evolutionary database development. They included six high-level categories: *transformation*, *structure refactoring*, *referential integrity refactoring*, *architecture refactor-*

Table 2: Low-level categories of atomic change types for database schema evolution.

Ref.	Atomic Change	Category	DDL (MySQL Implementation)
A1	Add Table	Transformation	`CREATE TABLE` t_name
A2	Add Column	Transformation	`ALTER TABLE` t_name `ADD` c_name
A3	Add View	Transformation	`CREATE VIEW` v_name `AS` ...
A4	Drop Table	Structure Refactoring	`DROP TABLE` t_name
A5	Rename Table	Structure Refactoring	`ALTER TABLE` o_t_name `RENAME` n_t_name
A6	Drop Column	Structure Refactoring	`ALTER TABLE` t_name `DROP COLUMN` c_name
A7	Rename Column	Structure Refactoring	`ALTER TABLE` t_name `CHANGE COLUMN` o_c_name n_c_name
A8	Change Column Datatype	Structure Refactoring	`ALTER TABLE` t_name `MODIFY COLUMN` c_namec_def
A9	Drop View	Structure Refactoring	`DROP VIEW` v_name
A10	Add Key	Structure Refactoring	`ALTER TABLE` t_name `ADD KEY` k_name
A11	Drop Key	Structure Refactoring	`ALTER TABLE` t_name `DROP KEY` k_name
A12	Add Foreign Key	Referential Integrity Refactoring	`ALTER TABLE` t_name `ADD FOREIGN KEY` fk_name ...
A13	Drop Foreign Key	Referential Integrity Refactoring	`ALTER TABLE` t_name `DROP FOREIGN KEY` fk_name
A14	Add Trigger	Referential Integrity Refactoring	`CREATE TRIGGER` $trig\_name$ ... `ON TABLE` t_name ...
A15	Drop Trigger	Referential Integrity Refactoring	`DROP TRIGGER` $trig\_name$
A16	Add Index	Architectural Refactoring	`ALTER TABLE` t_name `ADD INDEX` idx_name
A17	Drop Index	Architectural Refactoring	`ALTER TABLE` t_name `DROP INDEX` idx_name
A18	Add Column Default Value	Data Quality Refactoring	`ALTER TABLE` t_name `MODIFY COLUMN` c_name `SET DEFAULT` $value$
A19	Drop Column Default Value	Data Quality Refactoring	`ALTER TABLE` t_name `MODIFY COLUMN` c_name `DROP DEFAULT`
A20	Change Column Default Value	Data Quality Refactoring	`ALTER TABLE` t_name `MODIFY COLUMN` c_name `SET DEFAULT` $value$
A21	Make Column Not NULL	Data Quality Refactoring	`ALTER TABLE` t_name `MODIFY COLUMN` c_name `NOT NULL`
A22	Drop Column Not NULL	Data Quality Refactoring	`ALTER TABLE` t_name `MODIFY COLUMN` c_name `NULL`
A23	Add Stored Procedure	Method Refactoring	`CREATE PROCEDURE` pro_name ...
A24	Drop Stored Procedure	Method Refactoring	`DROP PROCEDURE` pro_name

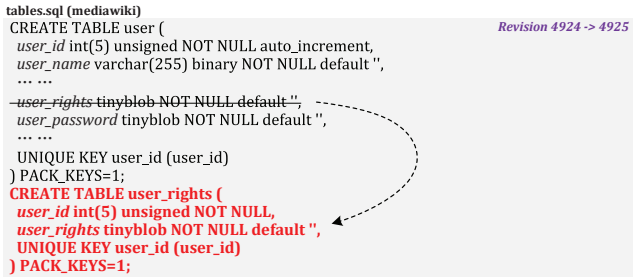

Figure 3: An example to illustrate the weaknesses of automatic schema difference extraction tools.

ing, *data quality refactoring* and *method refactoring*. The first is a non-refactoring transformation that changes the semantics of the schema, while the other five are refactoring transformations. To extract schema changes as accurately as possible, we further divide the six high-level categories into a more fine-grained classification that contains 24 *atomic* schema change types. They are listed in Table 2. Most of our atomic change types are adopted from Ambler *et al.* [1]. To be more complete, we have also introduced additional ones, such as A8 (change column datatype), A10 (add key), A11 (drop key), A14 (add trigger), A15 (drop trigger), A17 (drop index), A20 (change column default value), A23 (add stored procedure) and A24 (drop stored procedure). All possible composite schema change types (such as split a table or move a column) proposed by Ambler *et al.* [1] can be represented as a sequence of atomic changes using the 24 atomic ones. Another reason we selected these 24 atomic change types is that each can be simply translated into DDL, thus can be applied on schema files directly. In the 4th column of Table 2, we show an example how atomic changes are represented by DDL based on the implementation of *MySQL*.

Co-change analysis After having identified all possible atomic changes, we analyze the real impact caused by these atomic schema changes by mining a project's version control history. Different from traditional change impact analysis, which tries to identify the potential consequences of a change or estimate what needs to be modified to accomplish a change [2], we want to calculate the impact that has been triggered by schema changes. Co-change analysis [19] has been effectively applied on traditional software artifacts

Table 3: Results of DB revision and schema change extraction.

Project	# Total DB Rev.	# Valid DB Rev.	% Valid /Total	# Atomic Changes	# Atomic /Valid
Coppermine	116	69	59.5%	118	1.7
Dotproject	163	88	54.0%	279	3.2
e107	76	63	82.3%	114	1.8
Joomla!	532	133	25.0%	888	6.7
Mediawiki	377	221	58.6%	892	4.0
PrestaShop	221	203	91.9%	928	4.6
RoundCube	56	45	80.4%	101	2.3
Tikiwiki	941	493	52.4%	2,208	4.5
TYPO3	73	58	79.5%	249	4.3
webERP	189	91	48.1%	640	7.0
Total	2,744	1,464	53.4%	6,417	4.4

to estimate a change's impact area from co-change histories of similar previous changes. Adopting the same methodology, we use a database application's co-change history to estimate the application code area affected by a schema change. Thus, we approximate the set of schema-driven changes with *source code co-changed with schema files* in the same valid DB revision. SVN can tell which files changed together in one revision, so when a schema file changes, we can easily get all files co-changed with the schema file. For every pair of adjacent revisions *i* and *i+1* where *i+1* is a valid DB revision, we compare them to identify addition/deletion/change of co-changed source code. This analysis is automatic; we designed a simple difference extractor to calculate changed source lines (excluding comments) using static analysis. We have also performed a manual, in-depth examination of 10% randomly selected samples to guarantee the validation of our design choice.

3. DETAILED STUDY RESULTS

We first present results obtained from the first two steps of the analysis. Table 3 reports the summary statistics of the extracted information. The 2nd column lists the number of each application's DB revisions, while the 3rd column lists the number of *valid* DB revisions for each project. The 4th column shows that the ratio of valid over total revisions falls mostly in the 50-90% range. *Joomla!* has a much lower ratio because DMLs are involved in the same schema file with DDLs, making *data-sensitive changes* cover a large fraction of invalid revisions. The 5th column lists the total number of

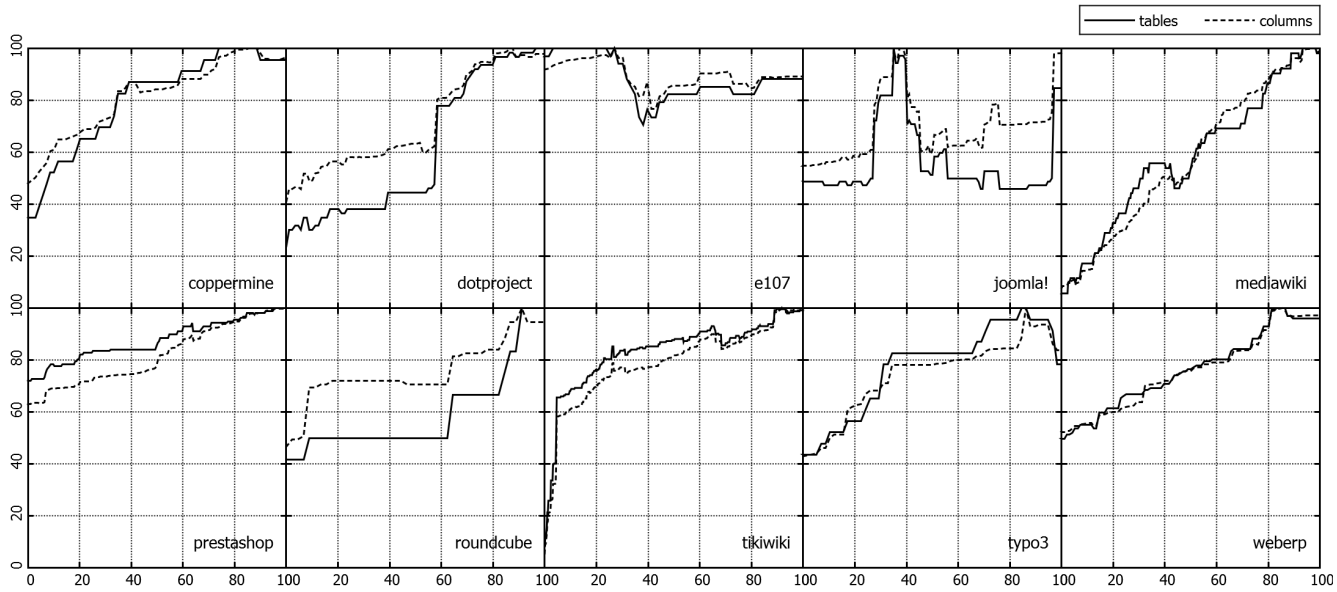

Figure 4: The evolution trend of *tables/columns* in the studied projects. The x-axis of each sub-figure shows the progression of the corresponding project's schema evolution, and the y-axis displays the accumulative percentages of the numbers of tables/columns *w.r.t.* the *maximum* numbers of tables/columns in the schema files.

Table 4: Frequency of schema evolution *w.r.t.* release and year.

Project	# Valid /Release	# Atomic /Release	#Valid /Year	# Atomic /Year
Coppermine	17.3	29.5	8.3	14.2
DotProject	12.6	39.6	13.3	42.2
e107	31.5	57.0	8.6	15.6
Joomla	26.6	177.6	20.5	136.6
Mediawiki	12.3	49.6	22.1	89.2
Prestashop	20.3	92.8	65.5	299.3
Roundcube	6.42	14.4	7.0	15.8
Tikiwiki	25.9	116.2	52.4	234.9
TYPO3	4.1	17.8	7.9	34.1
WebERP	9.1	64	11.4	80.0
Total	15.3	66.8	20.5	90.0

atomic changes; the 6th column lists the average number of atomic changes per valid revision, which falls mostly in the 2-7 range.

Next we use our study results to address the three research questions (Section 2.2).

3.1 RQ1: How frequently and extensively do schemas evolve?

First, we measure how frequently schemas evolve by examining the occurrences of schema changes *w.r.t.* each project's lifecycle. In particular, for each stable release/year, we calculate the average number of valid DB revisions/atomic schema changes. Table 4 reports this information. For each release, there are around 5∼25 valid DB revisions and 15∼180 atomic schema changes. For each year, there are around 10∼65 valid DB revisions and 15∼300 atomic schema changes. Although the numbers in each column differ due to the projects' varying levels of development activities and different definitions of stable releases, they provide solid evidence that schemas evolve frequently.

Second, we measure how extensively schemas change, by examining the trend on schema size changes. To this end, we collect the number of tables/columns in each valid revision to see how much schemas evolve. Figure 4 shows this trend information. The results clearly show that schemas increase in size in most projects over time. Two projects, *e107* and *joomla!*, exhibit frequent fluctuations. To facilitate a more precise evaluation, we use two metrics, *Growth Rate* (GR) and *Change Rate* (CR):

Table 5: Growth and change rates of schema size.

Project	Tables					Columns				
	IE	AE	DE	GR	CR	IE	AE	DE	GR	CR
Coppermine	8	15	1	175%	200%	85	102	16	101%	139%
Dotproject	15	54	6	320%	400%	182	316	64	159%	188%
e107	33	10	13	-9%	70%	249	99	106	-3%	82%
Joomla!	35	86	60	74%	417%	292	651	420	79%	367%
Mediawiki	3	59	11	1600%	2300%	27	377	58	1181%	1611%
Prestashop	113	55	11	39%	58%	547	376	53	59%	78%
Roundcube	5	8	1	140%	180%	35	58	22	103%	229%
Tikiwiki	20	279	57	1110%	1680%	111	2341	438	1714%	2504%
Typo3	10	13	5	80%	180%	122	166	50	95%	177%
WebERP	63	67	8	94%	119%	537	544	81	86%	116%

$$\mathrm{GR} = \frac{\#\ \text{Added Elements(AE)} - \#\ \text{Deleted Elements(DE)}}{\#\ \text{Initial Elements(IE)}} \quad (1)$$

$$\mathrm{CR} = \frac{\#\ \text{Added Elements(AE)} + \#\ \text{Deleted Elements(DE)}}{\#\ \text{Initial Elements(IE)}} \quad (2)$$

The different *elements* in equations (1) and (2) should be replaced respectively with tables and columns when calculating the *GR* and *CR* for tables and columns separately. The *GR* and *CR* of schema size in each project are shown in Table 5.

Results 1) Schemas evolve frequently: on average 65 atomic schema changes occurred per release, and 90 atomic schema changes occurred per year across the ten projects. 2) The size of schemas in most projects grew significantly: The *GR* of tables in 60% of the projects exceeded 100%; the *CR* of tables in 90% projects exceeded 100%. Although the number of tables in some projects increased slowly or even decreased (such as *e107*), they show frequent fluctuations. 3) We have observed very similar trend for columns (as compared to tables). 4) Seven projects' schema sizes reached 60% of their maximum values in about 20% of the selected project lifecycle, which indicates that more database related features were imported in the projects' early development phases.

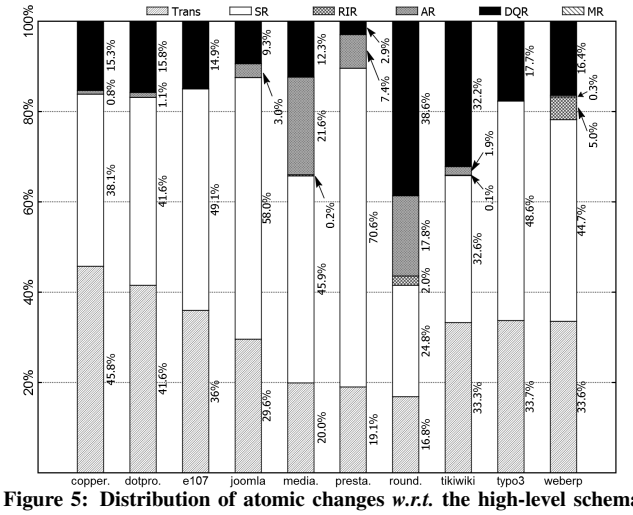

Figure 5: Distribution of atomic changes *w.r.t.* the high-level schema change categories.

In summary, database schemas evolve significantly during the development lifecycle of database applications. Next, we examine the co-evolution of application code caused by schema changes.

3.2 RQ2: How do database schemas evolve?

To answer RQ2, we analyze which schema change category each atomic change belongs to. Figure 5 shows the percentages for the six schema change categories: Transformations (Trans), Structure Refactoring (SR), Referential Integrity Refactoring (RIR), Architectural Refactoring (AR), Data Quality Refactoring (DQR) and Method Refactoring (MR). *Trans*, *SR* and *DQR* occurred the most frequently. These three categories cover more than 80% of schema changes across all 10 projects and over 95% across 7 projects. *AR* occurred in 8 projects, and *RIR* in only three, which covered only a very small part. *AR* and *RIR* were not often used, and *MR* did not occur in any of the 10 projects. It is interesting to note that *Tikiwiki* and *Roundcube* have much higher percentages of *DQR* compared to the other applications. This was due to substantial changes happened in certain versions of these two applications. For example, revision 966 of *Tikiwiki* removed the default values for most of the columns, affecting 550 (77.5% of all *DQR*) schema changes.

To understand the concrete type of each atomic change, we classified all collected schema changes *w.r.t.* their low-level change categories (Section 2). Table 6 shows the distributions for all atomic changes. We have highlighted (in **boldface**) the five most frequent low-level categories in each project. A1 (add table), A2 (add column) and A8 (change column datatype) were the most active atomic schema change types across the 10 projects. A4 (drop table), A6 (drop column), A7 (rename column), A18 (Add column default value) and A19 (drop column default value) occurred in all projects, but relatively infrequently in several projects. Although A10 (add key), A11 (drop key), A16 (add index), A17 (drop index), A20 (change column default value) and A22 (make column not null) appeared in all 10 projects, they were infrequent in many projects. A5 (rename table) and A21 (make column not null) happened only occasionally in most of the projects. In particular, A12 (add foreign key) and A13 (drop foreign key) rarely appeared and only showed up in 3 projects. This is because *foreign key constraints* were rarely used in schema definitions for performance concerns. The remaining six atomic change types, A3 (add view), A9 (drop view), A14 (add trigger), A15 (drop trigger), A23 (add stored procedure) and A24 (drop stored procedure), never occurred in any of the 10 projects — there were no view-related, trigger-related and procedure-related definitions in the schema files. It is interesting to

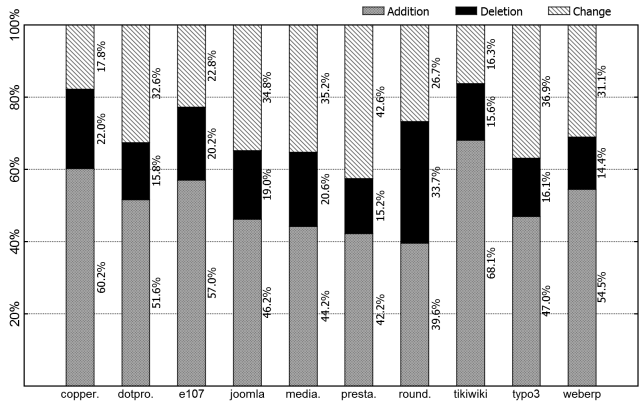

Figure 6: Distribution of addition/deletion/change on schema.

observe that *Prestashop* has much higher percentage of A8 (*change column datatype*). This was because revision 476 changed the column datatype from INTEGER to INT(10) (to save storage space), covering 26.2% of all schema changes.

We are also interested in the frequencies of *addition*, *deletion* and *change* operations on schema content. Figure 6 depicts the distribution. It shows that *addition* was the most frequent schema operation in 9 of the 10 projects. It accounted for about 40% of the operations in all 10 projects and over 50% in five projects. Moreover, *addition* and *change* accounted for around 80% of the operations, while *deletion* operations occurred less frequently across most projects.

Results 1) Three high-level schema change categories, *Trans*, *SR* and *DQR*, covered most schema changes; *AR* occurred relatively infrequently in some of the projects. 2) At the low-level, *add table*, *add column* and *change column datatype* were the most frequent atomic change types. 3) The data also confirms that referential integrity constraints (such as foreign key and trigger) and procedures (such as stored procedure) are indeed rarely used in practice. 4) *Addition* and *change* accounted for most of the schema evolution.

3.3 RQ3: How much application code has co-changed with a schema change?

Without a careful and laborious manual analysis, it is difficult to calculate the precise impact caused by a schema change. Neither is it feasible to perform the manual analysis at scale. Thus, we designed our study to *estimate* impact using the related information from project repositories. *Co-change analysis* has been effectively used on traditional software artifacts through large-scale experiments [19]. To confirm the validity of this approach for our setting, we examined two questions via a careful, non-trivial manual study: 1) How many valid DB revisions contain the co-change information of schema and code? and 2) How much code-level change is truly caused by schema changes? The first question is to understand whether and how often schema and code changes are committed together, and whether co-change history information is useful. The second question helps further explore the accuracy of co-change information as a means for estimating the code-level impact of a schema change. To answer the questions, we selected uniformly at random *10%* (146) of the valid DB revisions from the total 1,464 valid DB revisions and manually analyzed the co-changed information. Here we use lines of changed source code to represent the application change.

First, we need to analyze and understand how co-change information is present in the evolution history. Suppose R is the set of all valid DB revisions; r is the current one under analysis and C_r represents all changes committed in this revision. Schema changes

Table 6: Distribution of atomic schema changes *w.r.t.* the low-level categories of schema change types.

Category	Ref.	coppermine	dotproject	e107	joomla!	mediawiki	prestashop	roundcube	tikiwiki	typo3	weberp
Trans	A1	**33.1**%	**22.2**%	**27.2**%	**19.8**%	**13.8**%	**13.3**%	**8.9**%	**20.6**%	**28.5**%	**25.2**%
	A2	**12.7**%	**19.4**%	**8.8**%	**9.8**%	**6.2**%	**5.8**%	**7.9**%	**12.7**%	5.2%	**8.4**%
	A3	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
SR	A4	0.8%	2.2%	**11.4**%	**5.4**%	1.3%	1.1%	1.0%	2.6%	2.0%	0.5%
	A5	0.0%	0.0%	0.9%	2.1%	0.4%	0.2%	0.0%	0.4%	0.4%	0.2%
	A6	**11.0**%	**9.3**%	7.0%	4.5%	4.3%	1.6%	6.9%	4.5%	0.8%	5.9%
	A7	5.1%	2.2%	3.5%	4.3%	3.1%	1.3%	2.0%	1.4%	1.6%	2.7%
	A8	**7.6**%	**18.3**%	**16.7**%	**27.5**%	**27.1**%	**40.2**%	**14.9**%	**11.6**%	**34.9**%	**23.6**%
	A9	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	A10	**10.2**%	6.8%	**8.8**%	**9.6**%	5.2%	**21.0**%	0.0%	**7.1**%	**6.0**%	**7.2**%
	A11	3.4%	2.9%	0.9%	4.6%	4.4%	5.2%	0.0%	4.9%	2.8%	4.7%
RIR	A12	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	1.0%	0.0%	0.0%	3.6%
	A13	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	1.0%	0.0%	0.0%	1.4%
	A14	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	A15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
AR	A16	0.0%	1.1%	0.0%	2.5%	**12.0**%	0.5%	**11.9**%	0.6%	0.0%	0.2%
	A17	0.8%	0.0%	0.0%	0.6%	**9.6**%	**6.9**%	5.9%	1.2%	0.0%	0.2%
DQR	A18	3.4%	0.7%	**10.5**%	1.7%	3.3%	1.3%	9.9%	**25.2**%	**7.2**%	**9.8**%
	A19	4.2%	0.7%	0.9%	3.6%	0.3%	0.2%	**13.9**%	0.5%	0.8%	1.3%
	A20	5.1%	**12.2**%	1.8%	0.9%	4.5%	0.9%	**9.9**%	2.8%	0.0%	4.7%
	A21	0.8%	1.4%	1.8%	2.8%	3.7%	0.3%	0.0%	1.9%	0.0%	0.2%
	A22	1.7%	0.7%	0.0%	0.3%	0.6%	0.2%	5.0%	1.7%	**9.6**%	0.5%
MR	A23	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
	A24	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

SC_r and code changes CC_r correspond to co-change information in revision *r* where $SC_r \subseteq C_r$ and $CC_r \subseteq C_r$. The actual code change RC_r is the evolved code completely caused by SC_r. We have $SC_r \neq \varnothing$ where both CC_r and RC_r can be empty. There are four possible co-change situations:

(S1) $CC_r = \varnothing$ and $RC_r \neq \varnothing$. That is, RC_r occurs before/after revision *r*, *i.e.*, they were not committed in the same revision.

(S2) $CC_r = \varnothing$ and $RC_r = \varnothing$. This shows that schema changes do not impact code. In this case, the co-change information is still effective although no co-changed code is provided.

(S3) $CC_r \neq \varnothing$ and $CC_r \cap RC_r \neq \varnothing$. In this case, the co-changed code was committed together in revision *r* and contained the actual code change. If $CC_r = RC_r$, all of the co-changed code was caused by schema changes. Otherwise, the co-change history included other changes and may lead to inaccurate information.

(S4) $CC_r \neq \varnothing$ and $CC_r \cap RC_r = \varnothing$. In this case, although the code changes were committed together, they were not related to the schema changes. That is, all information provided by the co-change analysis is incorrect.

Figure 7a shows the distributions of four possible co-change situations. Regarding the first question, S2 and S3 provide effective co-change information. Figure 7a shows that around 72% of all valid DB revisions provided useful co-change information. In addition, the data for S2 showed that about 22% of valid DB revisions did not need any code changes as they had no impact on code.

For the 2nd question, we use *precision* (defined below) to estimate how much of the co-change history contains useful information:

$$\text{Precision(r)} = \frac{|RC_r \cap\ CC_r|}{|CC_r|} \quad (3)$$

It is obvious that, for any revision r that belongs to S1 and S4, the precision is 0%. That is, about 27% valid DB revisions did not provide useful co-change information. For the other valid DB revisions, Figure 7b shows their precision data. Our manual study shows that over 80% of valid DB revisions belonging to S2 and S3 have precisions over 60%, and over 70% have precisions over 80%. In particular, about 56% have 100% precision.

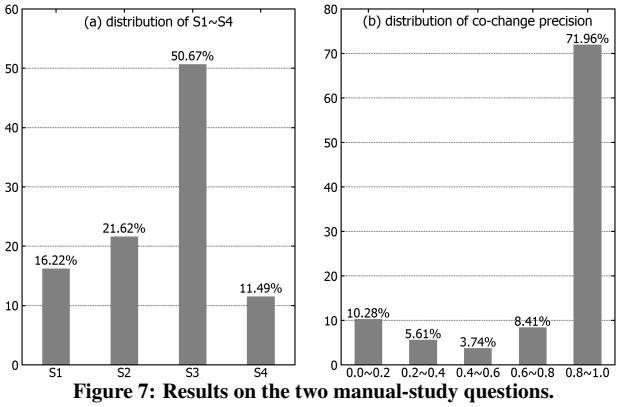

Figure 7: Results on the two manual-study questions.

Our results confirm that, as for traditional software artifacts, using co-change analysis to understand the impact of schema changes is indeed practical and effective. Hence, we study this question through co-changed source code lines within the same valid DB revision. We have designed and implemented a tool that automatically extracts the co-changed source code lines *w.r.t.* schema changes. Table 7 lists, respectively, the minimal, maximum, average and median number of source code lines co-changed with each atomic change and valid DB revision. For each atomic change, around 10∼100 lines were changed on average when it was performed. For each valid DB revision, which typically contains 2∼5 atomic changes, around 100∼1000 source lines were changed, which were quite significant changes.

We observed earlier that a valid DB revision usually involves multiple categories of schema changes. The summary statistics may not be informative enough since the impact on applications caused by different categories of schema changes can be quite different. To better understand the impact caused by schema changes, we calculate the source code lines co-changed with each category of schema changes separately. However, it is difficult to extract the amount of impact triggered by each category of schema changes without a manual analysis. As a practical alternative, we adopt *Multivariate Linear Models* to estimate the co-changed code lines

Table 7: Changed code size *w.r.t.* valid revisions and atomic changes.

Project	Per Valid DB Revision				Per Atomic Change			
	min	max(K)	$\bar{x}$	$\tilde{x}$	min	max(K)	$\bar{x}$	$\tilde{x}$
Coppermine	0	0.97	89	5	0	0.32	18	1
Dotproject	0	6.27	169	8	0	0.32	15	1
e107	1	4.35	169	17	0	1.45	40	6
Joomla!	0	52.6	890	27	0	0.98	47	9
Mediawiki	0	3.41	158	26	0	0.92	21	8
PrestaShop	0	5.50	174	20	0	1.11	20	8
RoundCube	0	1.71	136	10	0	0.55	23	2
Tikiwiki	0	154	716	18	0	20.2	53	5
TYPO3	0	52.5	1352	20	0	3.72	120	5
webERP	0	1.70	156	13	0	0.28	7	2

Table 8: Estimated changed code size *w.r.t.* high-level schema changes.

Project	$\hat{\beta_{Trans}}$	$\hat{\beta_{SR}}$	$\hat{\beta_{RIR}}$	$\hat{\beta_{AR}}$	$\hat{\beta_{DQR}}$	$\hat{\beta_{MR}}$	$\bar{R}^2$
Coppermine	157	10	0	0	18	NA	0.50
Dotproject	105	27	0	1	0	NA	0.80
e107	35	13	0	0	17	NA	0.32
Joomla!	243	54	0	0	41	NA	0.64
Mediawiki	112	16	0	4	0	NA	0.38
PrestaShop	103	29	0	0	17	NA	0.34
RoundCube	323	45	0	35	0	NA	0.64
Tikiwiki	232	36	0	50	13	NA	0.51
TYPO3	85	200	0	0	0	NA	0.32
webERP	35	33	0	8	0	NA	0.32

with each schema change category. Suppose y represents the total number of co-changed code lines in one valid DB revision; x_i represents the number of schema changes of high-level category i (i can be *Trans*, *SR*, *RIR*, *AR*, *DQR* or *MR*); β_i represents the corresponding co-changed code lines driven by schema change category i, we have

$$y = \sum_i \beta_i x_i \quad (4)$$

where $\beta_i > 0$. To guarantee all coefficients are positive, we use *Non-Negative Least Squares* to compute the coefficients. We also provide the *adjusted R-squared* to describe how well the calculated regression line fits the data set. Table 8 shows the results. Clearly the estimated impact (co-changed source code lines) caused by *Trans* and *SR* significantly exceeds the average in most cases. *Trans* exhibits the greatest impact. *SR* also has quite significant impact than the other four categories. *DQR* and *AR* have some effects on application code in certain situations. *RIR* has little or no impact on application code. Since *MR* did not appear in the revision history of any of the 10 projects, we cannot estimate its impact.

Results 1) Our detailed manual study on schema and code co-change history shows that more than 70% of all valid DB revisions contained effective co-change information, and among these, over 70% have precisions over 80%. Thus, our manual study confirms the validity and effectiveness of the co-change analysis. 2) Schema changes impact code greatly. For an atomic schema change, developers need to change about 10∼100 LoC on average; for a valid DB revision, developers need to change about 100∼1000 LoC. 3) Among the six high-level schema change categories, *Trans* and *SR* show more significant impact on application code that the others.

4. DISCUSSIONS

Our results for RQ1 and RQ2 show that schemas evolve frequently and extensively, and involve many change types. After deciding on a schema change, developers need to evolve the code consistently with the schema change. Our results for RQ3 further show that, with respect to a valid DB revision, developers typically need to change around 100∼1,000 source lines, a significant burden.

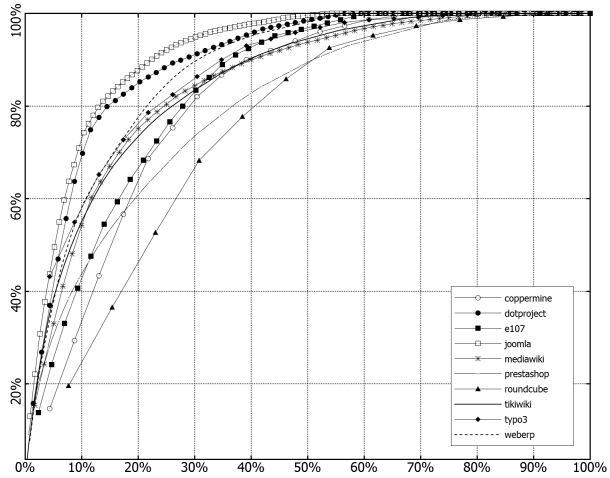

Figure 8: Coverage of schema changes by tables. The x-axis represents the percentage of total tables in schema, and the y-axis the percentage of total schema changes.

In addition, no tool currently exists that helps developers evolve database applications, which may explain why schema changes are often avoided [12]. However, when system requirements change, schema changes can become inevitable. To reduce manual efforts, it is desirable to develop assistive tools to aid database application evolution. We believe such tools should have the following functionalities:

(1) Before any potential schema change c, they can help find c's impacted code regions. This information can guide developers toward schema changes that minimize impact.
(2) After choosing an evolution strategy and schema change, they can help effectively locate all impacted code regions.
(3) Finally, they should also be able to guide developers how to evolve the code, such as recommending possible code changes. In certain restricted situations, they may even support automated program rewriting if the tools can capture sufficiently precise contextual information (although infeasible in general).

To satisfy the above requirements, we discuss two possible approaches. The first is to use program analysis to perform change impact analysis. For example, using structural code information, we may calculate dependencies (also called logic coupling [19]) of the program. Maule *et al.* [12] have addressed this problem for object-oriented programs that use relational databases. The technique may be adapted to realize the first two functionalities.

The second approach is to apply co-change analysis by mining association rules from evolutionary version histories. That is, the information recorded during the development and maintenance of the applications may be used to calculate the evolutionary coupling, which cannot be detected by program analysis [19]. The mined association rules can be used to suggest and predict likely further changes based on the already applied changes. Co-change analysis may be more useful for the evolution of database applications. First, the co-change history on schema and application code evolution in valid DB revisions is informative. We have observed that 1) over 70% of valid DB revisions provide useful co-changed source code lines that are related to the corresponding schema changes, and 2) over 70% valid DB revisions exhibit over 80% precision.

Second, the predictive power of a co-change analysis is closely related to the length of a project's revision history [18, 19]. Although valid DB revisions are relatively few over complete project revisions, they may be sufficient for evolving database applications. Schema changes are also centralized. That is, although schemas

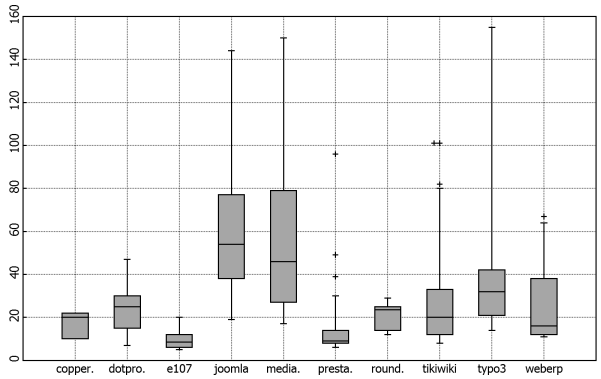

Figure 9: Distribution on the number of atomic schema changes across different tables. The x-axis lists the 10 database applications, and the y-axis the number of atomic changes for each different table.

evolve significantly, most schema changes concentrate in small parts of schemas — most components in schema definitions are stable. Figure 8 shows the coverage of schema changes by tables. We see that around 60%∼90% of the schema changes happened in 20% of the tables — most tables (around 40%) are rarely changed during the development lifecycle. The evolution histories of frequently changed tables are sufficient. The most frequently modified tables in total have covered over 80% of all the schema changes (*i.e.*, the total number of atomic schema changes occurred). Figure 9 shows the distribution. For each frequently changed table, its number of atomic schema changes is around 20. For *Joomla!* and *Mediawiki*, the two most mature projects with long evolutionary histories, their numbers are around 50. This provides evidence that the co-change analysis information can be useful in guiding code evolution.

Third, schemas and code in database applications correspond closely, meaning that, for each table defined in a schema, it exists code for manipulating the table. Thus, similar changes to the same table will likely have quite similar impact on the application code. As an example, consider Figure 10, which shows two similar schema-level changes and their corresponding code-level changes. Revision 1640 added the column ip_auto to table ipblocks (Figure 10a), and Figure 10b shows some selected corresponding changes to the affected PHP file `Block.php` from the column addition. A similar revision 2473 added another column ip_expiry to table ipblocks (Figure 10c), and Figure 10d illustrates the code-level changes.

As we can see, the code-level modifications for the two schema changes are quite analogous and match closely. This provides evidence that we may guide application code evolution by capturing and leveraging how schema and code co-evolve from revision histories.

In summary, it is desirable to build effective tool support to help evolve database applications, and the combination of *program analysis* and *co-change analysis* can be fruitfully exploited.

5. THREATS TO VALIDITY

Construct validity The construct validity of our study rests on the measurements performed, in particular related to the selection of valid DB revisions from the project evolutionary histories, identification of all atomic schema changes from the collected valid DB revisions, the co-change analysis to estimate the impact on applications caused by schema changes, and the calculation for code impact of different schema change categories.

Regarding selecting valid DB revisions, there are three potential threats. First, for each repository of the studied projects, we selected *trunk*, the mainline of the development process, to study. However, in some projects (such as *e107*), *truck* contains multiple branches of ongoing development lines (*e.g.*, *e107* versions 0.6 and 0.7 are developed simultaneously), whose evolution histories are not consistent. To reduce such noise, we selected only one branch that contains the longest evolution history of schemas. Second, based on our definition of a DB revision, we may omit tiny revisions where developers wrote external scripts to evolve the database and did not simultaneously update the schema files. Finally, since we manually filtered meaningless DB revisions based on our understanding, this may have introduced certain unavoidable bias.

Regarding identifying atomic schema changes, to better understand how schemas evolve, we attempted to recover the actual change trace of a schema using log information and our knowledge of database evolution to keep consistent with the original one. Unfortunately, some incomplete logs and our limited understanding of some projects may lead to misunderstandings. Similarly, the identification process may also involve our own human bias.

Regarding the co-change analysis between schema changes and application changes, we used source code lines co-changed with schema files in the same valid DB revision to estimate the impact caused by schema changes. From our manual validation, there are still about 27% of all valid DB revisions that did not provide effective co-change information, and about 20% of the cases with relatively low precisions based on our co-change strategy. However, our approach is likely the best compromise since if more adjacent revisions were regarded as impact caused by schema changes, additional noise, namely, irrelevant application changes, would be included in the co-change analysis, which may greatly reduce precision. More sophisticated impact analyses for database applications are available (such as [12]). In addition, we designed a difference extraction tool to automatically calculate the source lines co-changed with schemas. The implementation of our tool may contain errors. We utilized end-to-end regression testing throughout our tool's development to mitigate such a threat as much as possible. We thus are confident about our measurements.

Regarding calculating code impact by different categories of schema changes, we use multivariate analysis to estimate them based on the results of our co-change analysis. Although it may not indicate the precise code changes triggered by some schema change type, it provides high-level evidence that some schema change types exhibit more impact than others.

External validity Threats to external validity are concerned with whether the results are applicable in general. We selected 10 open-source database applications with different characteristics, such as schema and application size, and application domain. We obtained general findings for all projects and specific findings for individual projects. However, most projects are web-based applications, and many are implemented in *PHP*. It would be desirable to analyze more varieties of database applications (such as scientific database applications), developed in different programming languages to confirm our general conclusions.

6. RELATED WORK

The basic tasks in database application evolution, as shown in Figure 1, involve 1) schema evolution, 2) co-evolution of schema and data, and 3) co-evolution of schema and programs. We structure the discussion of related work accordingly.

Schema evolution There are several previous studies [8, 11, 14, 17] on schema evolution. Our work is different in several ways: 1) Previous studies mainly focus on how schemas evolve, while we not only include this aspect in our study, but also, and more importantly, how code co-evolves with schema changes; 2) We studied 10 popular open-source database applications from different

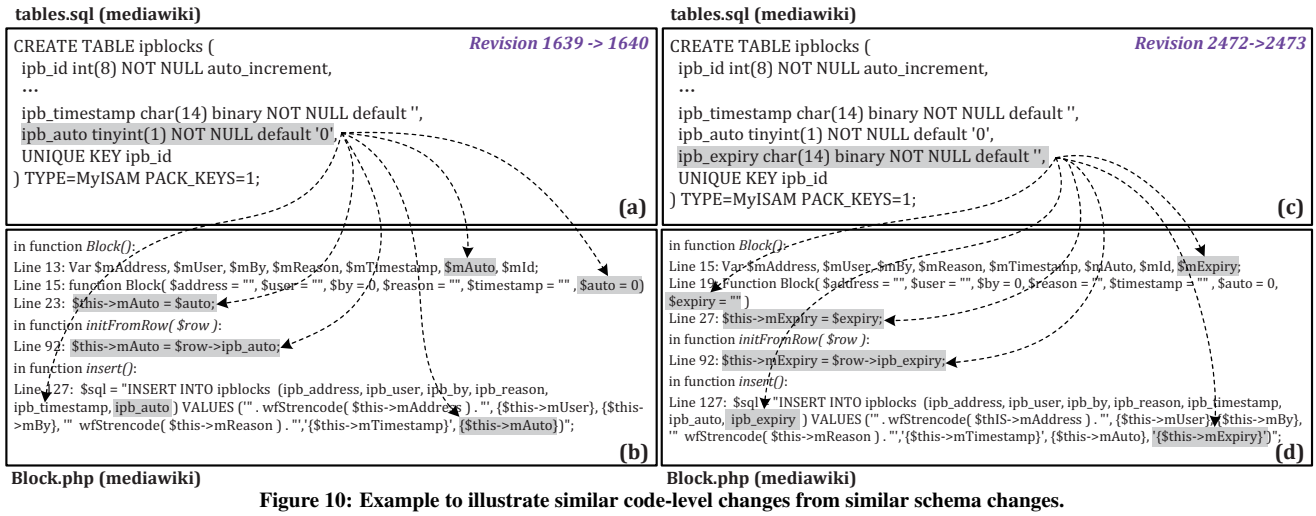

Figure 10: Example to illustrate similar code-level changes from similar schema changes.

domains with varying sizes, while previous work studied only a small number of projects; and 3) We studied the complete schema evolution history from their initial version to the current version, while previous work mainly focused on a period of the evolution.

Lin and Neamtiu [11] conducted a study of *collateral evolution* on two database applications, Firefox and Monotone. They considered the synchronization problem: when the definition of a database schema changes, how the schema migration code embedded in the application evolves to keep consistent with the new schema. In addition, they also studied the incompatibility problem of file formats between database and application code, caused by modifications of the DBMS specification on internal file formats.

Curino *et al.* [8] presented an empirical study of schema evolution on Wikipedia from April 2003 to November 2007. They analyzed basic information of schema evolution, such as schema size growth, and lifetime of tables and columns. They also provided both the macro and micro classifications of schema change types and the distribution of schema changes based on the SMO (Schema Modification Operator). In addition, they studied the effect of schema evolution on the front-end application by calculating the success rate of queries used in the previous version. The authors concluded that schema evolution may cause inconsistencies to the complete application. In contrast, our work tries to estimate the impact driven by schema changes at the code level from revision history and find possible solutions to assist code evolution.

Wu and Neamtiu [17] studied the schema evolution history of applications that use embedded databases. They designed a tool called *SCVD* (*Schema extraCtion and eVolution analysis for embedded Databases*) that can automatically extract the schemas embedded in source code and help people understand schema evolution. They used the tool to analyze the schema evolution history of embedded databases on four C++ open source projects. In our work, we manually extracted all possible schema changes from schema evolution history to make the result more accurate.

Sjøberg [14] also performed a schema evolution study on a health management system over 1.5 years. They found that most frequent changes are column additions/deletions and table additions/deletions. In our work, we also found additional interesting results, for example, 80% schema change happened in around 20% tables, *etc.*

Schema and data co-evolution There is much work in the database community on this topic [4,6,7,9,13]. Curino *et al.* [7] is a representative one. The authors designed a schema evolution tool, *PRISM*, to assist DB administrators in evaluating a schema change's effect at the schema level, optimized translation of old queries to work on new schemas version and automatic data migration. An updated version of PRISM, *PRISM++*, was implemented to support integrity constraint evolution and automatic query and update rewriting through structural schema changes and integrity constraint [6].

Schema and program co-evolution Little work exists that studied this topic. Change impact analysis is a good approach to support such kind of co-evolution. Maule *et al.* [12] proposed a program analysis-based approach to perform change impact analysis on object-oriented applications caused by schema changes in relational databases. They used a combination of program slicing and k-CFA data flow analysis to extract all possible insertions, updates, queries and stored procedure executions in database applications and implemented a tool called *SUITE*. Karahasanovic [10] mainly focused on object-oriented databases. In our work, we discuss the possibility of using the information of co-change analysis to guide developers to rewrite programs, which might be another solution for this direction. More importantly, our focus in this work is on a large-scale empirical analysis of such co-evolutions to gain insights, possibly for developing future assistive tools.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented the first large-scale study of how code co-evolves with schema changes in database applications using 10 popular projects. Our study has exposed new, interesting quantitatively information, which can be used as guidelines to develop assistive tools to help programmers evolve database applications. There are a number of interesting directions for future work. First, we plan to conduct a more comprehensive study with more applications and more varieties to increase the external validity of our findings. Second, we are interested in investigating techniques to increase the accuracy of estimating the code-level impact of schema changes, *e.g.* co-change analysis with Granger causality [3]. Finally, using our study results, we plan to develop tools to help developers evolve database applications.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback on an earlier version of this paper. This work was supported in part by the National Natural Science Foundation of China under Grant No. 60973149, the College Industrialization Project of Jiangsu Province under Grant No. JHB2011-3, Scientific Research Foundation of Graduation School of Southeast University Grant No. YBJJ1313, and United States NSF grants 0917392 and 1117603.

9. REFERENCES

[1] S. W. Ambler and P. J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley, 2006.

[2] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[3] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In *IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[4] A. Cleve. Program analysis and transformation for data-intensive system evolution. In *IEEE International Conference on Software Maintenance*, pages 1–6, 2010.

[5] A. Cleve, T. Mens, and J.-L. Hainaut. Data-Intensive System Evolution. *Computer*, 43(8):110–112, 2010.

[6] C. Curino, H. Moon, and A. Deutsch. Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++. *VLDB Endowment*, 4(2):117–128, 2010.

[7] C. Curino, H. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *VLDB Endowment*, 1(1):761–772, 2008.

[8] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in Wikipedia – toward a Web information system benchmark. In *International Conference on Enterprise Information Systems*, 2008.

[9] J. Hick and J. Hainaut. Database application evolution: A transformational approach. *Data & Knowledge Engineering*, 59(3):534–558, 2006.

[10] A. Karahasanovic. *Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation*. PhD Dissertation, Department of Informatics, University of Oslo, 2002.

[11] D.-Y. Lin and I. Neamtiu. Collateral evolution of applications and databases. In *Joint ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol)*, pages 31–40, 2009.

[12] A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *International Conference on Software Engineering*, pages 451–460, 2008.

[13] Y.-G. Ra. Relational schema evolution for program independency. In *International Conference on Intelligent Information Technology*, pages 273–281, 2004.

[14] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.

[15] A. Spiers. `http://adamspiers.org/computing/mysqldiff/`.

[16] Subversion. `http://subversion.apache.org/`.

[17] S. Wu and I. Neamtiu. Schema evolution analysis for embedded databases. In *Workshop on Hot Topics in Software Upgrades*, pages 151–156, 2011.

[18] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[19] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.