Collateral Evolution of Applications and Databases

Dien-Yen Lin Iulian Neamtiu
University of California, Riverside
Riverside, CA 92521, USA
{dienyen,neamtiu}@cs.ucr.edu

ABSTRACT

Separating the evolution of an application from the evolution of its persistent data, or from the evolution of the database system used to store the data can have collateral effects, such as data loss, program failure, or decreased performance. In this paper, we use empirical evidence to identify challenges and solutions associated with the collateral evolution of application programs and databases. We first perform an evolution study that identifies changes to database schemas in two popular open source applications. Next, we study the evolution of database file formats for three widely-used database management systems. We then investigate how application programs and database management systems cope with these changes, and point out how collateral evolution can lead to potential problems. Finally, we sketch solutions for facilitating and ensuring the safety of application and database evolution, hence minimizing collateral effects.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering; version control*; H.2.4 [**Database Management**]: Systems—*Relational databases*; H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*

General Terms

Measurement, Reliability

Keywords

Collateral evolution, coupled software transformation, software evolution, empirical study, schema evolution, schema migration, Mozilla, SQLite

1. INTRODUCTION

An increasing number of applications are shifting from storing data in custom file formats towards storing data using a database management system (DBMS). This enables an application to manage data in a more flexible and safe manner, while at the same time rendering the data easier to query. An example of migration towards using a DBMS as storage back-end is SQLite. SQLite is a lightweight, zero-config, server-less SQL engine encapsulated into a library that can be simply linked into an existing application and used via the SQL API. This lightweight yet powerful approach has proved to be very successful, and, as a result, SQLite is now used in software products from Apple, Google, Adobe, and Mozilla, as well as cellphones, PDAs, and MP3 players [5]. However, when evolving applications that use a DBMS instead of custom file formats, the developers are now tasked with having to evolve not only the application, but also its database schema, and the DBMS file format. When the evolution of these three components is not synchronized, the result is a potentially incorrect system.

The problem of maintaining global (system) consistency when its coupled subcomponents evolve is known as *coupled transformation*; a general characterization of coupled software transformations, broader than the scope of our work, has been provided by Lämmel [17]. Padioleau et al. [23] use the term *collateral evolution* in the context of the Linux kernel: since device drivers use services provided by the kernel and kernel support libraries, whenever the kernel or support library interfaces change, device drivers must change too, to adapt to the newest interface; if device drivers fail to do so, the result is potentially incorrect behavior.

Taking a cue from these prior works, we use the term collateral evolution to denote potential inconsistencies that arise when a database and the application programs using that database do not evolve in sync. We have access to a rich evolution history for both application programs and DBMSs, but we are missing studies that identify how these entities evolve, what the challenges of collateral evolution are, and what are some possible solutions. Therefore, we are now in a good position to study and tackle this problem.

In prior approaches, application and database evolution have usually been studied separately, i.e., software evolution in software engineering research and schema evolution in database research. In this work we make the case for an integrated approach to application and database evolution. While in this paper we only focus on relational databases, we believe the results presented here can be generalized and applied to other database models, e.g., object-oriented databases.

In Section 2 we provide a formal definition of the collateral evolution problem for applications and databases, in terms of incompatibilities between data formats expected by a data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWPSE-Evol'09, August 24–25, 2009, Amsterdam, The Netherlands.
Copyright 2009 ACM 978-1-60558-678-6/09/08 ...$10.00.

client and the format provided by a data server. This will permit us to exploit the analogy between schema and file format evolution, and phrase the associated challenges and solutions within the same framework.

In Section 4 we consider collateral evolution of applications and database schemas in the context of two applications, Mozilla and Monotone. We perform an empirical evolution study and find that database schema changes (e.g., table addition/deletion/renaming, or attribute addition/deletion/type change) are frequent. Our study points out that the source code of applications does not always evolve in sync with changes to the database schema, or it makes assumptions about schemas that can easily be violated.

In Section 5 we consider the collateral effects introduced by changes to database file formats. We consider the evolution of three widely-used DBMSs (SQLite, PostgreSQL and MySQL) over their entire lifetimes. We find that albeit relatively infrequent, database file format changes are a reality. We also find that DBMSs are less application-friendly than we would think, since a significant amount of manual effort is still required for updating the database to the new file format.

In summary, this paper makes the following contributions:

- A schema evolution study on two realistic, widely-used applications.
- A database format evolution study over the complete lifetime of three major DBMSs.
- A presentation of challenges and solutions associated with changes to database schemas and database file formats.

2. A FORMAL DEFINITION OF COLLATERAL EVOLUTION

At a high level, the fundamental problem with collateral evolution is due to different expected formats for the underlying data between a data client and a data server.

Example 1. Suppose we use Mozilla version 1.145 to access a bookmarks database created with Mozilla version 1.144; in this case, Mozilla is the data client, and the bookmarks database is the data server. This operation might or might not succeed, depending on the database schema employed by two Mozilla versions, 1.144 and 1.145, respectively.

Example 2. Suppose we use SQLite version 3.0 to open a database file created with SQLite version 2.0; in this case, SQLite is the data client, and the file is the data server. Again, this operation might fail if the database file formats used in SQLite 2.0 and 3.0 are incompatible.

We now generalize these examples to formulate the collateral evolution problem. We use D to denote the data and $\mathcal{F}(D)$ to denote the data format, i.e., its type. For the Mozilla example, the data type $\mathcal{F}(D)$ is the schema for the bookmarks database. For the SQLite example, the data type $\mathcal{F}(D)$ represents the file format used to store the database on disk. We use $\mathcal{F}_C(D, X)$ to denote the format expected by client C, version X, and similarly, $\mathcal{F}_S(D, Y)$ to denote the format provided by the server S, version Y. Whenever the data client or the data server evolves, we might end up with a potentially incorrect collateral evolution if the data type expected by the client differs from the type provided by the server. Formally, the incorrect evolution is captured by the following definition:

Application programs	Start date	End date
Mozilla	November 2005	May 2009
Monotone	April 2003	May 2009

Database management systems	Start date	End date
SQLite	May 2000	May 2009
MySQL	May 1995	May 2009
PostgreSQL	May 1995	March 2009

Table 1: Time span for the schema evolution study.

Definition *(Potentially incorrect collateral evolution)* Let X and Y be the data client and data server versions that result from collateral evolution. Let $\mathcal{F}_C(D, X)$ be the format expected by the data client C, and let $\mathcal{F}_S(D, Y)$ be the format provided by the data server S. The collateral evolution is potentially incorrect if $\mathcal{F}_C(D, X) \neq \mathcal{F}_S(D, Y)$.

Note that prior works have used similar terminology and notation, e.g., our correctness condition is related to Visser's $A \cong B$ condition [29] (where A is a called a *source type* and B is called a *target type*). Similarly, Lämmel and Lohmann [18] use the $d \vdash x$ notation and condition, akin to our $\mathcal{F}_C(D, X) = \mathcal{F}_S(D, Y)$, to express validity of evolution in the context of XML and DTDs.

In Sections 4 and 5 we discuss how application programs and DBMSs cope with collateral evolution. While the correct approach is to use a migration function $M_{Y \to X}$ to render the client and server formats compatible, this is not always the case for the programs we considered, and, as a consequence, collateral evolution can lead to errors.

3. APPLICATIONS AND METHODOLOGY

We studied schema evolution for two open source applications: the Mozilla project and the Monotone version control system. We chose these applications because they have long release histories and employ large database schemas, comprising dozens of tables. Mozilla stores browsing history, input forms, cookies, etc. into the database. Monotone uses the database to store file revisions, deltas, and branch information. Both Monotone and Mozilla hard-code the database schemas in the application's C++ source code. Therefore, each new source code release has to evolve in sync with, or migrate, the on-disk database.

Older versions of Mozilla had been using Berkeley DB for data storage. Due to license problems, the lack of query facilities and limitations on multi-process access, the Mozilla developers have gradually phased out Berkeley DB and phased in SQLite. Monotone has been using SQLite from the very start (release 0.1).

In March 2007, Mozilla switched the version control system for Mozilla 2 development from CVS to Mercurial; we used both CVS and Mercurial to extract the last 42 months' worth of source code revisions (see Table 1). For Monotone we downloaded all the releases from the project's website

(0.1–0.44), which corresponds to its entire six-years lifetime. For each Mozilla revision (and Monotone version, respectively), we extracted the associated database schemas, i.e., the tables and their attributes, from the C++ source code. With the tables and schema at hand, we manually computed all the changes to data tables and attributes. Next, for those changes we considered problematic, we examined the Mozilla and Monotone application code by hand, to identify the mechanisms used for (1) making changes to database schemas, and (2) ensuring that application and database schema evolve in a synchronized manner.

We also investigated changes to database file formats. While these changes come from the DBMS provider, rather than being introduced by application programs, they can nevertheless be problematic. We studied the official manuals and migration guides that come with SQLite, MySQL and PostgreSQL. Based on these documents, we characterized the evolution of file format changes over the entire lifetimes of each of the three DBMSs (Table 1), as well as the mechanisms the DBMSs use for accessing and migrating the on-disk files when file formats change.

4. SCHEMA CHANGES

In the relational database model, a database consists of *tables*, also known as *relations*. Each table is structured into *attributes* (named columns) and *records* (rows). A *table schema* consists of a table name and the set of its attributes, and a *database schema* is the union of all the table schemas in the database [6].

Following the conventions introduced in Section 2, we use $\mathcal{F}(D)$ to denote the schema of a database D; we investigate potentially-incorrect collateral evolution of an application program's schema definition $\mathcal{F}_C(D, X)$ relative to a database's schema definition $\mathcal{F}_S(D, Y)$.

While traditional database applications were designed with a fixed schema in mind, (i.e., future versions of an application will keep the schema unchanged) this assumption is no longer valid today. All major DBMSs allow the administrator to alter a database schema via table- and attribute-level operations such as additions, deletions and renamings. Moreover, specialized migration and integration tools such as Prism [4], South [2], Django [1] and DB-MAIN [26] can facilitate schema changes. These programs help describe database schema changes, assess the impact of changes, and help migrate the data from the old database into the new database.

In the remainder of this section, we first present the results of an empirical study on how database schemas change over time in Mozilla and Monotone, and then talk about challenges and solutions associated with each possible change.

4.1 Empirical Study

Tables 2 and 3 show a summary of table and attribute changes in Mozilla and Monotone. As we can see in Table 2, the most frequent table-level modifications are table schema changes, followed by table additions, table deletions and table renamings. Similarly, as shown in Table 3, the majority of attribute changes consist of additions and deletions.

In Table 4 we present a detailed account of changes for each table used in Mozilla and Monotone; empty cells correspond to the value 0. The first column contains the table name; if the table has been renamed over the period we considered, we show the old and new names, e.g., moz_anno

Program	Table changes			
	schema change	add	delete	rename
Mozilla	42	20	4	5
Monotone	11	9	8	1

Table 2: Table changes.

Program	Attribute changes		
	add	delete	other
Mozilla	58	30	3
Monotone	11	9	4

Table 3: Attribute changes.

has been renamed to moz_annos. The second column contains the number of changes to a table schema, and an empty table cell signifies no changes to that table's schema; for example, moz_bookmarks_folders had one change to its schema, while moz_bookmarks_roots had no schema change. An entry '1' in the third column indicates that the table was not present in Mozilla from the beginning, but rather was added at some point in the time frame we studied; for instance moz_cache_owners was added, while moz_cache was present in the initial version. Similarly, an entry '1' in the fourth column indicates that the table was deleted at some point during our study, e.g., moz_chunks. An entry '1' in the fifth column indicates that a table was renamed, and the new name appears on the next row. Finally, columns 6–11 show the number of attribute changes for a certain table, e.g., the roster_deltas table in Monotone has undergone one attribute addition, one attribute deletion, two changes to attribute initializers, one key change, zero attribute type changes and zero attribute renamings.

4.2 Challenges and Solutions

We now proceed to presenting challenges associated with application and database co-evolution using concrete examples from Mozilla and Monotone. While the developers address some collateral evolution challenges, many issues still remain unsolved; these issues can lead to data loss, application crash or performance degradation.

4.2.1 Mozilla

The handling procedures for schema changes in Mozilla differ from subsystem to subsystem, reflecting Mozilla's decentralized development process. Based on our study, Mozilla subsystems use two main approaches for dealing with schema changes; we term these approaches *version-oblivious* and *bidirectional*.

Version-oblivious evolution. The first standard mechanism used in Mozilla is to simply ignore the collateral evolution problem and assume that, if a database exists, its schema version matches the schema version of the application. We illustrate this approach in Figure 1. The nsNavBookmarks :: Init routine is in charge of initializing (or creating, if it does not exist) the moz_bookmarks_folders table. In revision 1.28 of module nsNavBookmarks, the table schema contains two attributes, id and name. In revision 1.29, the schema changes: a new attribute, type is added. As we can see on line 5, Mozilla creates the table if it doesn't exist already, e.g., if this is the first Mozilla run for this user. However, if the table does exist, the **if** condition on line 7

Table name	Table changes				Attribute changes					
	schema change	add	delete	rename	add	delete	type change	rename	init change	key change
Mozilla										
metaData		1								
moz_anno	3			1	6	3				
(ren. to moz_annos)										
moz_anno_attributes		1								
moz_anno_name		1	1							
moz_bookmarks_assoc	7			1	14	8			1	
(ren. to moz_bookmarks)										
moz_bookmarks_folders	1	1	1		1					
moz_bookmarks_roots		1								
moz_cache										
moz_cache_owners		1								
moz_chunks		1	1							
moz_classifier	3	1			7	5				
moz_cookies	5	1			3	2				
moz_downloads	7	1			10	3				
moz_formhistory										
moz_history	6			1	4	2		1		
(ren. to moz_places)										
moz_historyvisit	1			1	1	1				
(ren. to moz_historyvisits)										
moz_hosts		1								
moz_inputhistory		1								
moz_items_annos	1	1			2					
moz_keywords	1	1			1	1	1	1		
moz_memhistory										
moz_sub_chunks	3	1								
moz_subs	3	1			9	6				
moz_tables2		1								
moz_webappsstore	1	1		1	1					
(ren. to webappsstore)										
password		1	1							
Monotone										
branch_epochs		1								
db_vars		1								
file_Certs	1			1	1					
(ren. to revision_certs)										
file_deltas										
files										
heights		1								
incoming_queue			1							
manifest_certs	1				1					
manifest_deltas										
manifests										
merkle_nodes			1							
netserver_manifests			1							
next_roster_node_number		1								
posting_Queue	2		1		2	5				
privateKeys	1		1		1					
publicKeys	1				1					
revision_ancestry		1								
revision_roster		1	1							
revisions		1								
roster_deltas	2	1			1	1			2	1
rosters	1	1			1					
schema_version			1							
sequence_Numbers	2		1		2	3		1		

Table 4: Schema changes details for Mozilla and Monotone.

```
1  nsNavBookmarks::Init(DBConn)
2  {
3    ...
4    nsresult rv;
5    DBConn->TableExists("moz_bookmarks_folders",
6                        &exists);
7    if (! exists ) {
8      rv = DBConn->ExecuteSimpleSQL(
9           "CREATE TABLE moz_bookmarks_folders ("
10          "id INTEGER PRIMARY KEY, "
11          "name LONGVARCHAR, "
12          "type LONGVARCHAR)");
13     NS_ENSURE_SUCCESS(rv, rv);
14   }
15   ...
16   rv = DBConn->CreateStatement(
17          "SELECT id, name, type
18           FROM moz_bookmarks_folders
19           WHERE id = ?1",
20          getter_AddRefs(mDBGetFolderInfo));
21   NS_ENSURE_SUCCESS(rv, rv);
22 }
```

Figure 1: Mozilla: version-oblivious (incorrect) evolution.

is false, and Mozilla will not re-create the table or migrate the schema. Further down, on line 16, Mozilla tries to run a query assuming the new schema (note the presence of attribute type on line 17). If the query fails because the attribute does not exist, Mozilla returns an error. Note that the query is guaranteed to fail if the on-disk table is in the old format, e.g., if Mozilla has just been updated.

Bidirectional schema migration. A second approach for coping with collateral evolution is to determine, prior to accessing the database, both the version X of the application and the version Y of the database schema, and then perform the schema migration $M_{Y \to X}$. Note that we make no assumption on whether $X > Y$ or $Y > X$, i.e., the migration can be either an upgrade or a downgrade.

In Figure 2 we illustrate bidirectional migration with a code snippet from the nsNavHistory module. In this approach, each application program version defines a PLACES_SCHEMA_VERSION macro that corresponds to the current application-level schema version X (line 1). The nsNavHistory::InitDB routine then reads the database-level schema version Y (line 7). If the database schema is older than the application schema, i.e., $X > Y$, the routine brings it up to date using step-wise migration (lines 12–31) consisting of a suite of transformations $Y \to Y + 1 \to \ldots \to X$. The step-wise migration approach is similar to that employed in the O2 object-oriented database system [15] and the Ginseng dynamic updating system [21]. If, however, $X < Y$, the routine will perform a backward schema migration (lines 33-39). This situation occurs when the application program is older than the database schema, i.e., when trying to open a database created with a newer version of Mozilla.

To illustrate how schema migration is actually performed, we present a simplified version of the MigrateV6Up routine. When table moz_places evolves from revision 1.144 (July 21, 2007) to 1.145 (July 26, 2007), the attribute user_title is deleted. In Figure 3 we show the migration code that updates the old version, 1.144, to version 1.145. We see

```
1  #define PLACES_SCHEMA_VERSION 8
2
3  nsresult nsNavHistory::InitDB(DBConn)
4  {
5    ...
6    PRInt32 DBSchemaVer;
7    rv = DBConn->GetSchemaVersion(&DBSchemaVer);
8
9    if (PLACES_SCHEMA_VERSION != DBSchemaVer) {
10     if (DBSchemaVer<PLACES_SCHEMA_VERSION) {
11       // Upgrading
12       // Migrate up to V3
13       if (DBSchemaVer < 3) {
14         rv = MigrateV3Up(DBConn);
15       }
16       // Migrate up to V5
17       if (DBSchemaVer < 5) {
18         rv = ForceMigrateBookmarksDB(DBConn);
19       }
20       // Migrate up to V6
21       if (DBSchemaVer < 6) {
22         rv = MigrateV6Up(DBConn);
23       }
24       // Migrate up to V7
25       if (DBSchemaVer < 7) {
26         rv = MigrateV7Up(DBConn);
27       }
28       // Migrate up to V8
29       if (DBSchemaVer < 8) {
30         rv = MigrateV8Up(DBConn);
31       }
32     } else {
33       // Downgrading
34       // Downgrade v1,2,4,5
35       // v3,6 have no backwards incompatible changes.
36       if (DBSchemaVer > 2 && DBSchemaVer < 6) {
37         // perform downgrade to v2
38         rv = ForceMigrateBookmarksDB(DBConn);
39       }
40     }
41   }
42 }
```

Figure 2: Mozilla: bidirectional schema migration.

that the application correctly renames the existing table to moz_places_backup (lines 4–6), recreates moz_places with the new schema (lines 10–13), copies only the new schema's fields, omitting user_title (lines 16–20) and finally drops the old table.

4.2.2 Monotone

In Monotone, collateral evolution is dealt with (and schema migration is performed if necessary) in a centralized routine. This is in contrast to Mozilla, where each subsystem has its own mechanism for coping with co-evolution.

In Figure 4 we present a code snippet from Monotone's describe_sql_schema function. Similar to nsNavHistory::InitDB in Figure 2, this routine checks whether collateral evolution has occurred, and informs the user whether the database is usable, migration is required, etc.

Prior to starting the application and accessing the database,

```
1  MigrateV6Up(DBConn)
2  {
3    // 1. rename moz_places to moz_places_backup
4    DBConn->ExecuteSimpleSQL(
5    "ALTER TABLE moz_places RENAME TO"
6         "moz_places_backup");
7    ...
8    // 2. create moz_places w/o user_title
9    //    and its index
10   DBConn->ExecuteSimpleSQL(
11   "CREATE TABLE moz_places("
12        "id, url, title, rev_host, visit_count,"
13        "hidden, typed, favicon_id");
14   ...
15   // 3. copy all data into moz_places
16   DBConn->ExecuteSimpleSQL(
17   "INSERT INTO moz_places "
18   "SELECT id, url, title, rev_host,"
19        "visit_count, hidden, typed, favicon_id"
20   "FROM moz_places_backup");
21   ...
22   // 4. drop moz_places_backup
23   DBConn->ExecuteSimpleSQL(
24   "DROP TABLE moz_places_backup");
25 }
```

Figure 3: Mozilla: step-wise schema migration.

Monotone retrieves the on-disk schema version Y and compares it with the application schema X (line 3). As we can see on line 4, if the schema in the database is at the same version as the application, i.e., $X = Y$, the function returns SCHEMA_MATCHES. If the database schema is older than the version expected by the application (line 6), i.e., $X > Y$, the function returns SCHEMA_MIGRATION_NEEDED and a migration procedure (which we will explain shortly) will be invoked. Finally, Monotone checks for the situation when the database schema is newer than that expected by the application, i.e., $X < Y$. In this case, Monotone assumes that the database cannot be safely used, and the return value is SCHEMA_TOO_NEW (line 8). The remaining cases are just sanity checks. Therefore, in contrast to Mozilla, Monotone only supports unidirectional schema migration.

In Figure 5 we present a simplified version of the Monotone schema migration code. If, for instance, the database schema corresponds to application version 0.34, but the current application version is 0.44, Monotone will perform the conversion $0.34 \rightarrow 0.44$ using step-wise upwards migration. Monotone has a predefined array migration_events (lines 1–8) that contains migration functions. For example, the function migrate_add_heights_index (line 3) performs the conversion $0.34 \rightarrow 0.40$, while the function migrate_to_binary_hashes (line 5) performs the conversion $0.40 \rightarrow 0.44$. The reason Monotone "skips" some intermediate versions is that not all Monotone releases change the schema, e.g., versions 0.34 and 0.40 do change the schema, but versions 0.35–0.39 and 0.41–0.43 do not. The function migrate_sql_schema (starting on line 10) first retrieves the current database schema version (line 14), and then applies the schema migration procedures in sequence until it reaches the most current version, signaled by a 0 entry at the end of the migration_events array.

```
1  string describe_sql_schema( sqlite3 * db)
2  {
3    switch ( classify_schema (db)) {
4      case SCHEMA_MATCHES:
5        return "(usable)";
6      case SCHEMA_MIGRATION_NEEDED:
7        return "(migration needed)";
8      case SCHEMA_TOO_NEW:
9        return "(too new, cannot use)";
10     case SCHEMA_NOT_MONOTONE:
11       return "(not a monotone database)";
12     case SCHEMA_EMPTY:
13       return "(database has no tables!)";
14   ...}
15 }
```

Figure 4: Monotone: schema version check.

```
1  const migration_event migration_events [] = {
2                  // version 0.34 to 0.40
3    migrate_add_heights_index,
4                  // version 0.40 to 0.44
5    migrate_to_binary_hashes,
6    ...
7    0             // end
8  };
9
10 void migrate_sql_schema( sqlite3 * db, ...)
11 {
12     migration_event const *m;
13     ...
14     for ( find_migration (db); m; m++)
15       {
16         ...
17         migrate_func(m, db);
18         ...
19         printf ("migrated to schema        }
20 }
```

Figure 5: Monotone: forward schema migration.

4.3 Table Additions and Deletions

While technically table additions and deletions are part of schema changes, which we covered in Section 4.2, we believe that these table changes pose unique challenges that differentiate them from schema changes associated with attribute additions and deletions, hence they require further treatment.

Assume that in the transition from version X to version $X + 1$, a table T_1 is added, and a table T_2 is deleted. If the application is updated but the database schema is not migrated, version $X + 1$ of the application expects to find T_1 (which is not there), while table T_2 still remains in the database, although it not being used any longer.

We found that table deletions are common in Mozilla; we point out one problematic case. Table moz_anno_name is deleted when module nsAnnotationService is updated from revision 1.15 (June 19, 2006) to revision 1.16 (Dec. 18, 2006); revisions 1.16 and later do not use it any longer. However, there is no DROP TABLE moz_anno_name command in versions 1.16 and later. This leads to moz_anno_name becoming an "orphan" table—not used, but taking up space.

```
1  migrate_rosters_no_hash ( sqlite3 * sql ,
2                            upgrade_regime & regime)
3  {  ...
4     sqlite3_exec ("DROP TABLE roster_deltas");
5     sqlite3_exec ("CREATE TABLE roster_deltas
6
7       // id had type ``not null '' in previous version
8       id primary key,
9       checksum not null,
10      base not null,
11      delta not null");
12    ...
13    set_regime( upgrade_regen_rosters , regime);
14 }
```

Figure 6: Monotone: handling changes to default values.

The T_1 (addition) scenario is equally problematic: if no migration is performed, and the application version Y assumes T_1 is present in the database, then trying to open the table will lead to an error. However, we have not observed this scenario in the applications we studied.

4.4 Other Attribute Changes

Our study has found that, although additions and deletions constitute the majority of attribute changes (we covered them as part of schema changes in Section 4.2), attributes can also change in less obvious ways, and these changes can lead to collateral evolution problems.

4.4.1 Renaming

In Figure 7, we illustrate how errors can be introduced when attributes are renamed. We present the nsNavBookmarks module update from revision 1.68 (Dec. 16, 2006) to revision 1.69 (Feb. 07, 2007). The top half (lines 1–10) contains the old code, while the bottom half (lines 12–22) contains the new code. In revision 1.68 and earlier, table moz_keywords contains an attribute place_id ; in revision 1.69, the attribute is renamed to id. The handling code first checks to see if the table exists (line 4); if it does not exist, the table is created with the new schema. If the table does exist (line 21 and after), the old table schema is used, and herein lies the problem: the old schema does not contain a id attribute, so any database query that asks for it will fail.

4.4.2 Default Initializer Changes

Attributes can have default initial values, e.g., id integer **default** null. If data client C assumes default value D_X for a certain attribute, while data server S assumes default value D_Y, then the collateral evolution is problematic when $D_X \neq D_Y$.

In Figure 6 we present an example of correct handling of changes to default attribute values extracted from Monotone. In Monotone version 29, the table roster_deltas contains an attribute id with default value **not** null. In the next version, Monotone 30, the **not** null default value is dropped, i.e., the new definition is id primary key. To properly handle this situation, Monotone first drops the roster_deltas table (line 4) and recreates it with the correct schema(lines 5–11). Finally, the procedure upgrade_regen_rosters (line 13) will re-populate the table at the new version.

```
1  // old version , revision 1.68
2  nsNavBookmarks::InitTables(DBConn)
3  {  ...
4     if (! exists ){
5      DBConn->ExecuteSimpleSQL(
6       "CREATE TABLE moz_keywords("
7       "keyword VARCHAR(32) UNIQUE,"
8       " place_id INTEGER)"));
9      }
10 }
11
12 // new version , revision 1.69
13 nsNavBookmarks::InitTables
14             (mozIStorageConnection* DBConn)
15 {  ...
16    if (! exists ){
17     DBConn->ExecuteSimpleSQL(
18      "CREATE TABLE moz_keywords("
19      "id INTEGER PRIMARY KEY AUTOINCREMENT,"
20      "keyword TEXT UNIQUE)"))
21     }
22 }
```

Figure 7: Mozilla: attribute type change and attribute renaming.

4.4.3 Type Changes

While they are not part of traditional schema change operators [27], attribute type changes [19] can present challenges for collateral evolution.

In Figure 7, we show an attribute type change introduced in Mozilla, using the same nsNavBookmarks update from revision 1.68 to revision 1.69. In the old version, table moz_keywords has an attribute keyword of type VARCHAR (32). In the new version, keyword has type TEXT instead. The runtime behavior in light of this type mismatch is dependent on the SQL engine used. Mozilla relies on the fact that SQLite does not enforce the length of a VARCHAR, effectively representing VARCHAR and TEXT in the same way [3]. However, when using a different SQL engine, the type mismatch can result in a runtime error. Therefore, the correct behavior would be to first check the schema version and use schema migration in case of version mismatch.

4.4.4 Key Changes

Another potentially problematic attribute change consists of changes to an attribute's key status. On lines 7 and 8 of Figure 6 we show how the attribute id becomes a primary key in the table roster_deltas. By definition, each row in the table has a unique value for the column associated with the primary key. If the database schema is not migrated when an attribute becomes primary key, we violate the uniqueness assumption. A similar key change appears in Mozilla (Figure 7) where the attribute place_id is renamed to id and becomes a primary key.

Even in situations when a key change does not lead to a database inconsistency, it can still affect database operations, e.g., in terms of performance. For example, as per Oracle 10g's manual, the size of the redo log depends on whether a modified attribute is a primary key or not. Therefore, if an attribute becomes primary key but the schema is not migrated, (or vice versa, an attribute is no longer

primary key), the DBMS will either do too much (or not enough) logging.

5. FILE FORMAT CHANGES

While table and attribute changes are under application program developers' control, the developers are also faced with changes they have little control over, i.e., the database file format. The file format is hidden from the application, but DBMS producers often choose to modify it, to offer improved performance, reduce storage size, or implement a new standard [28, 20, 24].

Following the notation introduced in Section 2, we use $\mathcal{F}(D)$ to denote the file format for a database D; we investigate potentially-incorrect collateral evolution of a DBMS' notion of the file format, $\mathcal{F}_C(D, X)$ relative to the on-disk file format, $\mathcal{F}_S(D, Y)$.

Upgrading a DBMS, e.g., using SQLite 3.0 instead of 2.0 can lead to a wide range of issues for the application programs that link with SQLite. Even though the application program and the on-disk files are unchanged, the new SQLite version can either fail to read the files created in version 2.0 format, or, if it can read them, certain commands will fail to execute [28]. For most of today's DBMSs, the standard procedure for dealing with changes in file format is to use the old version (in our case, 2.0) to "dump" the database into a batch file of SQL commands, then upgrade the DBMS and recreate the database using the new DBMS and the new file format (in our case, 3.0).

In the remainder of this section we present our findings after studying the entire lifetimes of SQLite, MySQL and PostgreSQL. First, we present details on the frequency of file format changes. Second, we identify challenges posed by these changes and solutions for dealing with these challenges.

5.1 Empirical Study

SQLite is being developed at a rapid pace, though its file format is quite stable; most of the changes to the file format occurred in early versions [28]. The first version of SQLite was released in May 2000, and the latest version in May 2009. As shown in Table 5, over its 9 years lifetime, SQLite has changed the file format 13 times, though only 3 of those changes require a dump-based migration; two other changes, not mentioned in the table, do not affect regular SQLite users, but affect those performance-critical applications providing their own OS abstractions. The correspondence between file format changes and the release numbering scheme used by SQLite is simple: major changes to the first release digit, e.g., 2.8.14 to 3.0.0 are incompatible, whereas changes to the second digit, e.g., 3.0.8 to 3.1.0 are meant to be backwards-compatible.

MySQL was first released in 1995, and the last release, 6.0alpha, came out in May 2009. MySQL uses a simple scheme to signal file format changes: whenever the file format changes, the major version number is incremented. As shown in Table 5, over its 14-year existence, MySQL has had 5 file format changes (from series 1.x.x to series 6.x.x).

PostgreSQL's first release was 0.01 (May 1995); the latest release was 8.3.7 (March 2009). PostgreSQL signals changes to the file format by changing the second digit in release number. For example, 7.0 and 7.1 are major releases with potentially incompatible file formats, while 7.1.x and 7.1.y should be compatible. Over its entire 14-years lifetime, we counted 21 file format changes (Table 5).

Program	Time frame (years)	File format changes
SQLite	9	3 (13)
MySQL	14	5
PostgreSQL	14	21

Table 5: File format changes.

5.2 Challenges and Solutions

Collateral evolution of DBMSs and file formats can lead to initialization-time or run-time errors. For example, when attempting to use PostgreSQL 8.0 to open a database created with PostgreSQL version 7.3, the server fails with the error "`FATAL: database files are incompatible with server`". Another example is using SQLite 2.2.0 to manipulate a database file created with SQLite 2.1.x. While the file can be loaded, there is a restriction on commands that can be used to manipulate the file [28]. In particular, due to changes in the underlying representation, the command `INTEGER PRIMARY KEY` is disabled; after upgrading the file format to 2.2.0, the command `INTEGER PRIMARY KEY` can be used normally.

To avoid such errors, MySQL [20] and PostgreSQL [24] documentations clearly state that the procedure for updating between major (i.e., file-format changing) releases is to back up the existing data, "dump" the DB contents to a SQL script containing the commands needed to recreate all the database records from scratch, upgrade the DBMS, and run the script to recreate and populate the database at the new format.

Interestingly, SQLite [28] is much more user-friendly in this regard. Over its 9-year lifetime, SQLite has made only three incompatible file format changes that require the user to perform the "dump" procedure. Of the remaining 11 changes, all are backwards-compatible, i.e., the new version can read (and sometimes auto-convert) a file created by an old version. Moreover, some file format changes even provide a limited form of forward-compatibility, e.g., files created with version 3.2.0 can be read by older DBMSs (3.1.4–3.1.6, but not older that 3.1.4).

We believe that the seamless file format conversion mechanisms used in SQLite should be adopted by other DBMS producers as well. This would relieve the DBMS users and application programs program developers of tedious, manual `dump`-based migration method when file formats change.

6. FUTURE DIRECTIONS

As pointed out throughout our paper, collateral evolution of application programs and databases presents challenges to application developers and DBMS producers alike. Even though our findings could prove valuable to both categories, we believe further research is needed into minimizing the effects of collateral evolution and the effort spent by developers and users in migrating databases.

While certain changes to schemas are inherently incompatible with prior versions, a mechanism for seamlessly migrating the database schema both forward and backward (such as the one used by Mozilla, presented in Section 4.2.1, Figure 2) should become standard practice.

Looking further ahead, we note that the schema and database migration mechanisms we considered so far assume offline migration. That is, the application program does not

use the database while it performs schema migration, and the DBMS is shut down while a dump/import is in progress. However, many applications such as mission-critical systems, transaction processing, or online service providers cannot afford to halt the system while a schema/file format update is in progress; for these categories of applications, updating the program while providing continuous service to clients is essential [13]. On-the-fly schema evolution and file format migration have seen little investigation, and current implementations are far from practical [14]. We plan to use the results of this collateral evolution study to help open the way towards on-the-fly database updates, in the spirit of *dynamic software updating* [21].

7. RELATED WORK

Schema modification operators (SMO) [27, 12] and schema evolution primitives [7] allow schema changes using table-level operations (e.g., CREATE, DROP, RENAME, COPY, MERGE, PARTITION, JOIN, DECOMPOSE) and attribute-level operations (e.g., ADD, DROP, RENAME). Our study focuses on a subset of SMO-style changes to tables because not all SMO operators are supported in today's DMBSs. On the other hand, our treatment of changes to attributes is more fine grained i.e., we study and assess the impact of type and key changes, which are not part of SMO.

Curino et al. [11] performed a study on how Wikipedia's schema has evolved between April 2003 and November 2007. They use macro-classification and micro-classification of changes. The micro-classifications correspond to SMO syntax, which is a superset of the changes we investigate. The micro-classifications include attribute key and attribute type changes, but omit default initializer changes. We present detailed, per-table attribute changes, rather than per-database as they do. Our approach facilitates understanding of how individual tables change, and whether schema changes are localized to a few tables, or spread throughout the database. Schema matching [25] and ontology mapping [16] try to address the problem of accessing data where the client and server formats are different by providing matching or mapping functions between the two formats. We could envision using a schema matching procedure $M_{X \leftrightarrow Y}$ that converts between the data client and data server representations on-the-fly. While helpful for avoiding runtime errors when the data client and data server's format are no longer compatible due to evolution, matching/mapping approaches do not convert the underlying data, i.e., they do not perform schema migration. Moreover, when the application and the underlying data are out of sync, the functionality is likely to be limited (i.e., storing a newly-added attribute), therefore these approaches provide limited help with collateral evolution.

A solution similar to schema matching or schema mapping is to use *lenses* [8, 9], an approach for bi-directional transformation between strings belonging to different languages. Lenses have the advantage that composing bidirectional migration functions $M_{X \to Y} \circ M_{Y \to X}$ yields the identity function; this is accomplished by saving deleted attributes off to the side in a dictionary. While lenses can reduce or eliminate errors introduced by collateral evolution, just like schema matching and mapping, this is only a transient solution. Because lenses do not perform the actual schema migration, the data server's version remains Y and the data client's version remains X; however, eventually the database has to be migrated.

Lämmel and Lohmann studied the problem of format evolution and coupled schema transformation in the context of transforming XML documents when their underlying DTDs change [18]. Their transformations for refactoring DTDs are similar to SMOs and our schema changes. Their consistency condition, $d \vdash x$ expresses the fact that an XML document x is valid according to a DTD d. This condition is similar to our $\mathcal{F}_C(D, X) = \mathcal{F}_S(D, Y)$ requirement, though their treatment of possible transformations, as well as well-formedness and validity conditions is more rigorous than ours. Visser [29] formalized the coupled transformation problem as constructing a two-level transformation between a source type A and a target type B witnessed by conversion functions between A and B. Their types (and type transformation operations, called refinements) stem from data refinement theory [22] and are more general than types of relational database schemas (and schema change operations, respectively). Their $A \cong B$ condition is similar to our $\mathcal{F}_C(D, X) = \mathcal{F}_S(D, Y)$ requirement.

Cleve and Hainaut [10] present an approach for *co-transforming* application code to stay in sync with schema evolution. Their solution is to (1) define a set of primitives that allow mapping between different versions of database instances and databases schemas, and (2) based on these mappings, generate wrapper functions that allow data migrations, or permit data clients and servers to interact without actually migrating the data. They present a case study of automatically generating wrappers in COBOL for two data reengineering cases of medium-sized database applications. Rather than focusing on the semantic of transformations and automated wrapper generation, our study provides a fine-grained characterization of database and application changes to two realistic systems over several years and many releases. The empirical evidence gathered in our study could be used to identify, and construct a library of, frequent schema transformations.

8. CONCLUSIONS

In this paper we analyze the collateral evolution of applications programs and databases, motivated by the recent trend towards greater adoption of DBMSs as data storage back-ends in regular applications. We perform a schema and file format evolution study on widely-used open source programs. Based on this study, we identify challenges introduced by the collateral evolution of applications and the databases they use. We point out how current co-evolution approaches are inadequate, and provide possible solutions. We believe that our study, as well as the challenges and solutions identified in this work can help make collateral evolution of applications and databases easier and safer.

Acknowledgments. We thank Dietrich Ayala of Mozilla Corporation for explaining why our Mozilla type change example is actually harmless, and the anonymous referees for their helpful comments on drafts of this paper.

9. REFERENCES

[1] Django Software Foundation. Django Schema Evolution. http://code.djangoproject.com/wiki/SchemaEvolution.

[2] South: Intelligent schema migrations for Django. http://south.aeracode.org/.

[3] SQLite FAQ. http://www.sqlite.org/faq.html#q9.

[4] The PRISM Project. http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Prism.

[5] Well-known users for sqlite. http://www.sqlite.org/famous.html.

[6] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing mapping composition. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 55–66. VLDB Endowment, 2006.

[8] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419, 2008.

[9] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM.

[10] A. Cleve and J.-L. Hainaut. Co-transformations in database applications evolution. In *GTTSE*, pages 409–421, 2006.

[11] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In J. Cordeiro and J. Filipe, editors, *ICEIS (1)*, pages 323–332, 2008.

[12] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, 2008.

[13] A. Deshpande and M. Hicks. Toward on-line schema evolution for non-stop systems. Presented at the 11th High Performance Transaction Systems Workshop, September 2005.

[14] T. Dumitras, J. Tan, Z. Gho, and P. Narasimhan. No more hotdependencies: toward dependency-agnostic online upgrades in distributed systems. In *HotDep'07: Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, page 14, Berkeley, CA, USA, 2007. USENIX Association.

[15] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec. Schema and database evolution in the o2 object database system. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 170–181, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[16] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18(01):1–31, 2003.

[17] R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, Nov. 2004.

[18] R. Lämmel and W. Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[19] B. S. Lerner. A model for compound type changes encountered in schema evolution. *ACM Trans. Database Syst.*, 25(1):83–127, 2000.

[20] MySQL. Upgrading MySQL. http://dev.mysql.com/doc/refman/4.1/en/upgrade.html.

[21] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 72–83, New York, NY, USA, 2006. ACM Press.

[22] J. N. Oliveira. Transforming data by calculation. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 134–195. Springer, 2007.

[23] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *EuroSys*, pages 59–71, 2006.

[24] PostgreSQL. Migration between releases. http://www.postgresql.org/docs/8.0/interactive/migration.html.

[25] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[26] REVERsa. DB-MAIN. http://www.db-main.com/.

[27] B. Shneiderman and G. Thomas. An architecture for automatic relational database system conversion. *ACM Trans. Database Syst.*, 7(2):235–257, 1982.

[28] SQLite. File format changes. http://www.sqlite.org/formatchng.html.

[29] J. Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electron. Notes Theor. Comput. Sci.*, 200(3):3–23, 2008.