2012 IEEE 36th International Conference on Computer Software and Applications

Automatic Adaptation of Software Applications to Database Evolution by Graph Differencing and AOP-based Dynamic Patching

Yang Song[1], Xin Peng[1], Zhenchang Xing[2], Wenyun Zhao[1]
[1]School of Computer Science, Fudan University, Shanghai, China
[2]School of Computing, National University of Singapore, Singapore
{10210240033, pengxin, wyzhao}@fudan.edu.cn
xingzc@comp.nus.edu.sg

***Abstract*—Modern information systems, such as enterprise applications and e-commerce applications, often consist of databases surrounded by a large variety of software applications depending on the databases. During the evolution and deployment of such information systems, developers have to ensure the global consistency between database schemas and surrounding software applications. However, in such situations as Enterprise Application Integration (EAI), databases are shared by a number of software applications contributed by different independent parties, and the developers of those applications often have little or no control on when and how database schema evolves over time. As a result, databases and software applications may not always remain in sync. Such inconsistency may lead to data loss, program failures, or decreased performance. The fundamental challenge in evolving and deploying such database-centric information systems is the fact that databases and their surrounding software applications are subject to independent, asynchronous, and potentially conflicting evolution processes. In this paper, we present an approach to automatically adapting software applications to the evolution of their underlying databases by graph-based schema differencing and aspect-oriented dynamic patching. Our empirical study shows that our approach can automatically adapt software applications to a number of common types of database schema evolution, which accounts for over 87.5% of all schema evolution in the subject system. Our approach allows database schema maintainer to evolve database schema more freely without being afraid of breaking surrounding software applications; it also allows application developers to catch up database schema evolution more quickly without diverting too much from their main business concerns.**

Keywords-software evolution; database evolution; software maintenance; application adapter

I. INTRODUCTION

Modern information systems, such as enterprise applications for resource planning and customer relation management, e-commerce applications, web-based information systems, often consist of databases surrounded by a large variety of applications depending on the databases. Databases enable an application to manage data in a safer and more reliable manner, while at the same time rendering the data easier to query.

On the other hand, the use of databases implies the design and implementation of the application's business logic heavily depends on the data access interfaces provided by the underlying databases, such as the entities and their attributes that can be accessed by the application and the relevant integrity constraints. For example, the business logic of order generation in an online shopping system will be designed and implemented around a sequence of database operations such as reading product information, adding a new order record, and update order items.

During the evolution and deployment of such database-centric information systems, developers have to evolve not only the databases, but also the surrounding applications depending on the databases. For example, if the database schema evolves, such as an entity has been renamed, a one-to-one relationship has been changed to one-to-many, or a column has been dropped, the software applications should be adapted accordingly. When databases and applications do not remain in sync during the evolution and deployment of the system, their inconsistencies may lead to data loss, program failures, or decreased performance [2], [16] .

Such inconsistencies become more problematic in the context of Enterprise Application Integration (EAI), in which databases are shared by a number of software applications contributed by many independent parties, and the developers of those applications often have little or no control on when and how database schema evolves over time. Furthermore, during application integration, an updated database schema may be erroneously integrated with an older version of the application.

The problem of maintaining global consistency between databases and surrounding applications during system evolution and deployment can be considered as a special case of the general co-evolution problem of different components of an information system. The fundamental challenge lies in the fact that those components are subject to independent, asynchronous, and potentially conflicting evolution processes.

Existing work has usually investigated database and application evolution separately. In software engineering research, several approaches have been proposed to support the co-evolution of framework APIs and client applications reusing those APIs [13], or to support the collateral

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.21

evolution of Linux kernel and device drivers [22]. On the other hand, schema and data evolution research in database community has usually been considered as a special case of the general problem of heterogeneous schema mapping and data integration [9], [24], [25], [26]. In both communities, little attention has been paid to the co-evolution of databases and their surrounding applications and the graceful deployment of the evolved system (including databases and applications).

In this paper, we present an approach, called AppCatchupDB, to supporting the automatic adaptation of software applications in the face of the evolution of the underlying databases. AppCatchupDB uses graph-based schema differencing technique to automatically analyze and detect the differences between the database schema expected by an application and the schema supported by a database. It then uses Aspect-Oriented Programming (AOP) technique to intercept SQL queries issued by the application. Given the queries intercepted and the differences between the expected and supported database schemas, AppCatchupDB applies heuristic-based graph transformation rules to infer appropriate adapter for adapting the intercepted queries. Finally, AppCatchupDB exploits AOP technique again to patch the inferred adapters into the application at runtime.

Our contribution in this paper is three-fold. First, different from existing work in the area of software engineering and database [8], [10], [13], our approach focus on automatically infer and dynamically apply co-evolution patches that application developers used to manually generate and statically apply in order to adapt their applications to work with the evolving database. Second, we summarized eleven common types of database schema evolution and formalize those types using graph grammar and transformation. Third, we implemented our approach in an open-source software tool and report the promising results of our initial evaluation of the proposed approach and tool support.

The remainder of this paper is organized as follows. Section II presents a motivating example that illustrates the impact of database schema evolution on software applications and how our approach help application adapt to such database schema evolution. Section III presents our approach step by step. Section IV presents our empirical study. Section V discusses some issues related to our approach and empirical study. Section VI summarizes related work on graph differencing technique and database evolution and co-evolution. Section VII concludes our work and outlines future work.

II. MOTIVATING EXAMPLE

Digital Campus is a modern information and application integration platform deployed and run in an university. The platform consists of a shared database and a set of software applications depending on it.

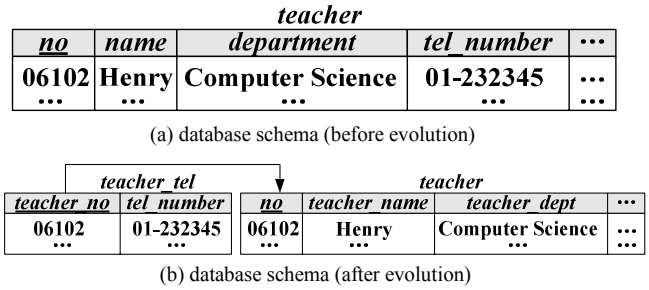
The course selection application is one of the applications depending on this shared database. This application allows students to browse information about courses and teachers, select desired courses and check their course grades. It accesses the shared database to retrieve information about students, courses and teachers and update the information.

Figure 1(a) shows a fragment of the initial database schema for teacher. This schema assumes that a teacher has at most one telephone number (i.e. one-to-one relationship). Functions and services of the initial course selection application have been implemented based on this initial database schema. Table I presents two queries that the course selection application issues to retrieve relevant information about teachers. Clearly, the provision of these functions and services at runtime is based on the data access interfaces provided by the shared database, including table and column names and their relevant integrity constraints.

TABLE I. FUNCTIONS OF THE COURSE SELECTION APPLICATION

Function	Database Operations
browse teacher information	select name, department, tel_number from teacher where no=?
browse teacher name of one department	select no, name from teacher where department = ?

As more and more teachers have more than one telephone number, the DBA team decides to evolve the database schema to a newer version as shown in Figure 1(b). In this new version, a separate table ***teacher_tel*** is introduced to enable recording multiple telephone numbers for a teacher (i.e. one-to-many relationship). Furthermore, some columns of the ***teacher*** table such as ***name*** and ***department*** are renamed to comply with newly adopted naming conventions.



(a) database schema (before evolution)

(b) database schema (after evolution)

Figure 1. An example of database schema evolution

Unfortunately, this new database schema will render some functions of the course selection application invalid. For example, the "browsing teacher information" function can no longer successfully retrieve teacher's information, since the columns ***name***, ***department***, ***tel_number*** no longer exist in the table ***teacher*** in the updated database schema.

In this case, if the application cannot be adapted to the new database, it will fail to provide required services. Our AppCatchupDB can automatically detect and patch the inconsistencies between the expected database schema of

the application and the actual schema of the shared database. The produced patches can adapt the SQL queries in TABLE I to those in TABLE II. The new SQL queries can work on the updated database schema and thus the two functions of the application can perform well as before.

TABLE II. ADAPTED SQL STATEMENTS

Function	Database Operations
browse teacher information	select teacher.teacher_name as name, teacher.teacher_dept as department, teacher_tel.tel_number as tel_number from teacher, teacher_tel where teacher.no=? and teacher.no = teacher_tel.teacher_no
browse teacher name of one department	select teacher.no as no, teacher.teacher_name as name from teacher where teacher.teacher_dept = ?

III. THE APPROACH

Figure 2 presents an overview of our approach AppCatchupDB. AppCatchupDB works in between an application and a database at runtime. It intercepts the SQL queries that the application issues using AOP technique and adapts those queries (if necessary) so that they can be successfully executed on the database.

AppCatchupDB assumes that both the expected database schema of the application and the supported database schema of the database can be obtained from the system. In fact, it is reasonable to assume that an application is released and deployed together with the database schema (e.g. in the format of XML files) that it expects. On the other hand, the actual database schema can be obtained, for example, from the data dictionary of the shared database.

At the first time the application issues an SQL query to accesses the database, AppCatchupDB queries the application and the database, and check whether the expected schema version of the application matches the supported schema version of the database. If the expected and supported schema versions match, AppCatchupDB simply passes the issued query to the database.

Otherwise, it attempts to adapt the issued query so that it can be executed on the supported version of the database. To that end, AppCatchupDB first apply (only once) graph-based schema differencing to compare the expected schema version of the application and the supported schema version of the database and to detect the differences. Then, given an intercepted query, AppCatchupDB parses it into a tree model and applies graph transformation rules (predefined according to the types of database schema changes) in order to adapt the intercepted query to the supported schema version of the database. As a result, the adapted SQL queries can be successfully executed on the database even though the expected schema version of the application is different from the supported schema version of the database.

A. Detecting Schema Changes

Given the expected schema version of the application and the supported schema version of the database, AppCatchupDB applies a simple graph-based schema differencing technique to detect the differences between the two schema versions.

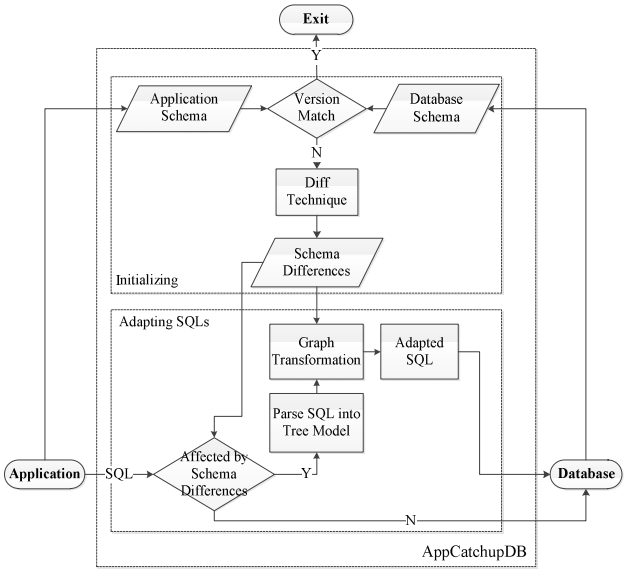


Figure 2. Framework of our approach

AppCatchupDB considers database schema as a undirected graph G (V, E), The node set V consists of all the tables of the schema. The edge set E consists of the foreign keys constraints between tables. Given the two graphs G1 and G2, corresponding to the two schema versions of the database, AppCatchupDB first matches the nodes of G1 and G2 based on the names of the tables and the columns. Then, it considers unmatched nodes as added and removed tables and columns. After that, AppCatchupDB attempts to infer the renaming, joining, decomposing, merging and partitioning of tables based on the table matching results. Finally, AppCatchupDB infers the differences of two versions of columns according to the table matching results.

AppCatchupDB reports the differences between the two schema versions as a set of schema changes that can transform one database schema to the other as summarized in TABLE II. I. Our definition of these types of schema changes is consistent with the existing work in the area of database research [1], [8], [16], [18]. We choose these types of schema changes summarized in Table III because they are atomic and occur most frequently in database evolution [7]. Furthermore, they can be easily extended in the future.

B. Parsing and Modeling SQL Statements

Given an intercepted SQL statement, if the expected and supported schema versions mismatch, AppCatchupDB parses the SQL statement into a SQL tree model to facilitate the adaptation of the SQL statement. The current implementation supports the adaptation of the *insert, delete, select and update* SQL statements.

TABLE III. SCHEMA CHANGES APPCATCHUPDB CAN AUTOMATICALLY DETECT AND ADAPT

Type of evolution	Schema Changes
Add Table *R*	_ → R(**A**)
Drop Table *R*	R(**A**) → _
Rename Table *R* into *T*	R(**A**) → T(**A**)
Merge Table *S*, *T* into *R*	S(**A**), T(**A**) → R(**A**)
Partition Table *R* into *S*, *T*	R(**A**) → S(**A**), T(**A**)
Decompose Table *R* into *S*, *T*	R(**A**, B, C) → S(**A**, B), T(**A**, C)
Join Table *S*, *T* into *R*	S(**A**, B), T(**A**, C) → R(**A**, B, C)
Add Column *C* into *R*	R(**A**) → R(**A**, C)
Drop Column *C* from *R*	R(**A**, C) → R(**A**)
Rename Column *B* in *R* to C	R(**A**, B) → R(**A**, C)
Move Column *C* from *R* to S	R(**A**, C) → S(**B**, C)

AppCatchupDB applies GOLD parser [19]to parse SQL statements, and the SQL grammar that it can handle is the ANSI 89 version of SQL that are widely used by Oracle, Microsoft and most other database developers. Gold parser builder can create a skeleton program and produce Matthew Hawkins Engine (an Abstract Syntax Tree (AST) parser implemented in Java) and can be applied easily as a .jar package.

SQL *select* statement consists of *Column, Into, From, Where, Group, Having and Order* clauses. *Column* clause has the values that are returned to the application. When AppCatchupDB builds a SQL tree model, some additional operations will be applied. For example, if there is a '*' notation in the *column* clause such as "Select * from *T*", our approach cannot obtain the names and the number of returned columns. Moreover, when the DBA adds a new column in *T* or changes the order of columns in *T*, the returned values will be different from those before. Thus we replace the "*" notation with the corresponding column names, because we think this is a bad smell and will cause troubles in schema evolution. Other clauses are built as tree models similar to the AST structure.

Insert, update and *delete* statement are simpler than *select* statement and their *where, from, column clauses* can be handled in the similar way to those of *select* statement.

C. Adapting SQL Statement

Let us now discuss how AppCatchupDB adapts the intercepted SQL statements.

1) Reserved Graph Grammar and Graph Gransformation

First, we give an overview of the Reserved Graph Grammar (RGG) [3] and graph transformation that provide theoretical foundation for the adaptation technique used in our AppCatchupDB approach.

The Reserved Graph Grammar [4] is a context-sensitive graph grammar formalism, which is expressed in a node-edge format; a node is organized with a large rectangle called super-vertex, and one or more small rectangles called vertices embedded with super-vertex presented in Figure 3. .

A RGG consists of a set of graph grammar rules, also called *productions*. A production has two graphs called *left graph* and *right graph* as shown in Figure 3. Given a graph to transform (called *host graph*), when the right hand side of a production is matched in a host graph, it will be replaced by the left hand side of the production which is called *R-application*. On the other hand, the reverse process is called *L-application*. Graph transformation can be realized by applying a sequence of productions on a given graph. [5]

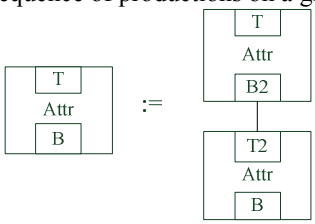


Figure 3. Node and production

The RGG offers a transformation mechanism [3], i.e. graph *transformation rules* that specify the transformation from an input graph to a different graph. Graph transformation is the application of a sequence of rules on a given host graph. The transformation process produces a new graph from the input host graph after applying transformation rules and terminates when no more transformation rules can be applied [6].

2) Adapting SQL Statement with Graph Transformation

Given the tree model of an intercepted SQL statement, if the tables and/or columns referred in the SQL statement have been changed between the expected and supported schema versions (see Section B), AppCatchupDB attempts to adapt the intercepted SQL statement.

AppCatchupDB first attempts to normalize the parsing tree model. AppCatchupDB use "as" grammar to redirect the values returned and the table queried. For example, if a column C in table T is renamed to D, we can modify SQL query like "select C from T" to "select D as C from T" to make the SQL query work as normal and return column name the same as before. As a result, this can easily avoid the impact of renaming. Moreover, it can make the returned value keep consistent to the application.

And then, AppCatchupDB applies a set of graph transformation rules to transform the input SQL tree model into a new tree model whose corresponding SQL statement can be successfully executed on the supported schema of the database. The applied graph transformation rules are defined for each type of database schema changes as follows.

Rename table/column: to adapt this evolution type is easy because we can use "as" clause to redirect the renamed one in the SQL statement.

Add table: if an added table is truly new one (not copied or decomposed) due to business logic, we do not care about it because it will not make impact on the SQL statement of the expected version explicitly.

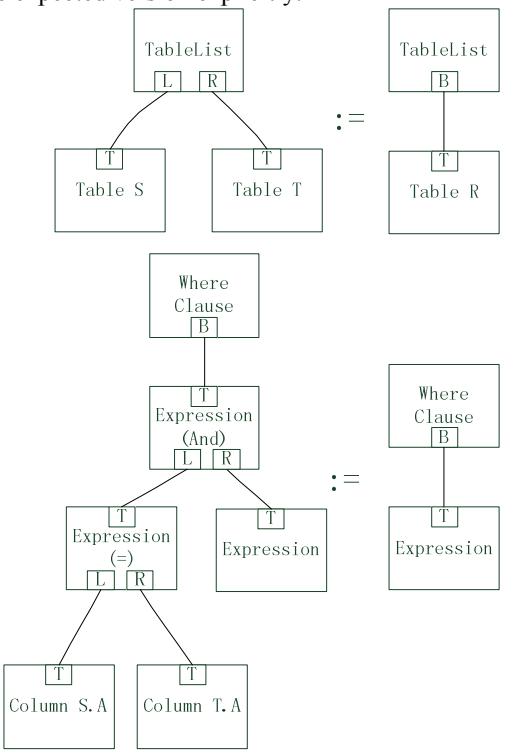


Figure 4. One of the Productions of Decomposing Table

Remove table: if a removed table is truly removed one, we will try to find those equivalent columns in other table.

Add column: like added table, however there are several situation that this evolution type impacts. If a *select* statement has a "*" in its *column* clause, we first replace "*" to the columns of the expected version of application, then the queried columns of the statement is not related to the added column.

Remove column: like removed table. If a *select* statement queries a removed column we return a null value instead of the value before or finding an equivalent column or ignoring it.

Decompose Table: if one table is decomposed to two tables, we need check which columns the new tables have to replace the old table by the new two tables. Then, we need add *where* conditions in most situation because when we query the new two tables, we often use their foreign keys to filter the data. Figure 4 presents the production of decomposing table: R(**A**, B, C) → S(**A**, B), T(**A**, C)

Join Table: like decomposed table, if two tables is joined into one table, we need replace the old two tables by the new one table. Then, we need adjust *where* conditions if necessary.

Move Column: moved column seems like that one table is decomposed to two tables first then one of the two is joined with another table.

Partition Table: if one table is partitioned to two tables, we need execute SQL queries on each table and union them.

Merge Table: about merging table, our policy is simple that only rename the old two tables.

We list graph transformation productions of decompose in Figure 4. . The *from* clause and *where* clause occur graph transformation. In *from* clause, the table ***R*** is replaced by the table ***S*** and ***T***, because in the supported version of database, table ***R*** is equal to table ***S*** joining ***T***. Meanwhile in *where* clause, a filter condition is added to keep foreign key constraints.

D. Dynamic Patching

We apply AOP technique to adapt SQL statements to the supported version of database at runtime. For example, in JDBC, we can apply AOP technique to patch SQL statements when application invokes the method "*Connection.prepareStatement*". After AppCatchupDB patches the SQL statements, AOP can execute adapted SQL statements in place of the old. When AppCatchupDB adapts SQL statements to the supported version of database at runtime, a report including evolution information and how to adapt SQL statement is produced for developer of DBA to check or find useful information.

Because AppCatchupDB works at runtime, we make some optimization on efficiency. Often the number of SQL statements of one application is limited and we can store the patching result for each to avoid duplicate processes. If a SQL statement is parsed, we add it to dynamic storage as a mapping table-a SQL statement in expected version of application maps a SQL statement patched. Moreover, if a SQL statement is different from those in the dynamic storage, we check its structure and if their structures are the same, we need only replace the parameters and save the time of graph transformation. In fact, this optimization is special useful during the situation of prepared statement of JDBC and can save much time.

IV. EVALUATION

We evaluate our approach on two open-source database-centric information systems. Our initial evaluation focuses on the applicability and efficiency of our AppCatchupDB approach and tool support.

The two subject systems are iBATIS JPetStore, which is a completely rewritten Pet Store application based on Sun's original J2EE Pet Store, and Oecrm an open-source Customer Relation Management (CRM) application. Pet Store has 25 SQL statements dealing with orders and products in its accessing module, and Oecrm has 26 SQL statements dealing with customers and activities in its accessing module.

TABLE IV. EVOLUTION INFORMATION AND IMPACT ON SQL QUERIES OF TWO CASE

	JPetStore (25 SQL templates)		Oecrm (26 SQL templates)	
Evolution change	**Change count**	**SQL statements affected**	**Change count**	**SQL statements affected**
Table adding	0	0	0	0
Table removing	0	0	0	0
Table decomposing	1	4	1	7
Table joining	1	4	0	0
Table renaming	1	5	0	0
Table merging	0	0	0	0
Table partitioning	0	0	0	0
Column adding	0	0	3	7
Column removing	3	6	1	5
Column renaming	5	9	1	4
Column moving	0	0	0	0

The schema of two versions of Pet Store has several schema changes such as renaming and dropping columns or some table restructuring (see Table IV). For example, the database schema of iBATIS JPetStore has a table called account including information of user's name, email, address, phone and others about account. There is a requirement that this table is decomposed to two: one has information about account, the other has information about authentication including name, email, phone.

We list evolution changes and their impact on the database schema as Table IV.

A. Applicability

To evaluate the applicability of our approach, we define the results of our approach after AppCatchupDB adapts and patches the intercepted SQL statements at runtime as followed:

Perfect patch: application can apply the patched SQL statement as before and the patched SQL statement is able to work on the supported version of database and return correct value to application.

Limited patch: application can apply the patched SQL statement as before, the patched SQL statement is able to work on the supported version of database but return null value (column removing happens) or inconsistent data due to schema changes.

Bad patch: the patched SQL statement is able to work on the supported version of database but application cannot apply the patched SQL statement as before because the initial parameters of the statement do not match it.

We make an evaluation on the patched SQL statements to classify the results as Table V.

There are ten and fourteen perfect patches which can work perfectly on the supported version of database, respectively. The applications are able to apply these as which in the expected version of application, and achieve correct values which they return. We find our approach can patch 62.5% and 73.7% SQL statements as perfect patch. Furthermore, the application developer does not need to care about these SQL statements affected by schema changes.

TABLE V. EVALUATION RESULTS OF TWO CASE

	JPetStore		Oecrm	
Result type	**Result count***	**Percent**	**Result count***	**Percent**
Perfect patch	10	62.5%	14	73.7%
Limited patch	4	25%	5	26.3%
Bad patch	2	12.5%	0	0%

*the total of results is not equal to 25 because others are not affected

There are four and five limited patches because removing column occurs. We cannot avoid the impact of removing column because it makes the supported version of schema lose data. The application can apply these patched SQL statements as the expected version but may achieve several "null" values returned.

Both perfect patch and limited patch are able to make the application work normally with the supported version of database schema. In this case study, we find our approach can patch 87.5% and 100% SQL statements as perfect patch or limited patch.

We find that there are two bad patches. The bad patches occur because one table is decomposed and if an update SQL updates all the columns of this table or an insert SQL inserts all the column of it, AppCatchupDB should separate the SQL statement into two. Moreover, in some kinds of visiting database, these two bad patches will be perfect patch because application can apply the two patched SQL statements as the old schema using some APIs like JDBC.

B. Efficiency

To evaluate the efficiency of our approach, we calculate the spending time of patching each SQL statement.

TABLE VI. TIME COST OF APPCATCHUPDB

N	T (ms)	$T_{average}$ (ms)
100	5704	2.28
1000	55339	2.21
10000	543515	2.17

In this case study, we apply our approach to patch the total 25 SQL statements in Pet Store for N times. If total spending time is T, the average spending time of patching each SQL statement $T_{average}$:

$$T_{average} = T / N / 25 \quad (1)$$

From TABLE VI. I, the extra spending time added by our approach on each SQL statement is about 2 ms. The time is irrelevant to the scale of data in database. So we argue that the efficiency of our approach is reasonable for application to apply. TABLE VII. I is the experiment setting.

TABLE VII. EXPERIMENT SETTING

Machine	RAM	3GB
	CPU(2x)	Intel Core2 P8700 2.53GHz
	Disks	320GB
OS	Version	Windows Vista Business SP1
Java	Version	1.5.0-10
MySQL	Version	5.0.67

We find that our approach can really patch the SQL statements in expected version of application to the supported version of database excellent in most situations and reduce much time of application developers and DBAs to co-evolve the application and database. When the database loses data during evolution, our approach will make some limited patched inevitably but it can produce some information in the log to note the developer and DBA.

V. DISCUSSION

AppCatchupDB implements a simple diff technique based on some heuristic rules. For example, when AppCatchupDB finds adding table or removing table, we need to analyze the columns of the tables to recognize whether decomposing or joining table happens. Moreover, according to the information of schema, we need to find whether the removed and the added tables/columns are renaming type. However, these heuristics work only if there are no dramatic changes between two versions of database schema. If many tables or columns are changed at the same time, our differencing technique cannot work well.

Dramatic schema changes will affect the quality of our differencing technique, then in turn on adaptations of SQL queries. However, our AppCatchupDB can be effectively applied in the situation where database schema remains stable largely but suffers certain amount of schema changes and can help developer to reduce some daunting work such as replacing strings or changing many statements due to those schema changes. In fact, our approach in this paper is developer-oriented to reduce much trivial work and can adapt SQL queries at runtime. In such cases, our differencing technique is good enough to detect precisely the schema changes between the two versions of database schema. In practical situation, dramatic schema changes hardly happen on the database between the two subsequent versions. Thus, we ignore dramatic schema changes currently, but in the future we will improve the diff technique to handle more complex schema changes.

In our empirical studies, the size of subject systems and the number of schema changes and SQL statements are limited. We choose two classic open source business applications: Pet Store which has 25 SQL statement templates and Oecrm which has 26 templates. As these two applications are not large systems, the statistical results may not be generalized to larger, more complex systems. However, they are representative database-centric information systems. And in the future, we plan to evaluate AppCatchupDB with more large-scale data-centric systems.

In our approach, we do not consider the concrete data existing in the database. In fact, potential information about database schema is able to be mined from the data existing in the database according to the researchers in database area [23]. This will help us to handle co-evolution of application and database. We will consider it in the future work.

VI. RELATED WORK

Evolution is an essential characteristic of software systems. We refer to the papers in database area about database evolution including relational and object-oriented databases [1], [7], [8], [20]. Lin et al. [8] indicate that database evolution mostly happens on schema changes and the changes of columns mostly occur on addition and deletion. Similar results are reported in a study by Sjoberg [1]. In [21], Lerner presents schema evolution types like merge, move, add link, link reverse, duplication.

DBAs can apply techniques like PRISM [7] to manage database schema evolution about data migration and query modification with Schema Modification Operators (SMO). Papastefanatos et al. [10], [11] conduct a series of researches on adaptive query formulation and propose a framework called Hecataeus to handle database schema evolution. AppCatchupDB focuses on the co-evolution of applications and underlying databases and works at runtime to adapt applications to evolution of database.

For the purpose of evolution analysis, some diff techniques such as Genericdiff and mysqldiff are proposed. Genericdiff [14] is a general framework for model comparison and can produce accurate comparison reports about model differences for diverse types of models. Mysqldiff [15] can compare the data structures of two MySQL databases and return the differences.

The necessity of online techniques for the co-evolution of applications and databases is also noticed by researchers. Deshpande et al. [17] argue that the capability of managing database schema evolution on-line without halting the system is essential for more applications like online services. Our approach provides such a technique that can work at runtime and dynamically adapt applications to database.

In the context of reuse-based software development, Xing et al. work on the co-evolution of reusable frameworks and client programs. Their approach API-CatchUp [13] tackles the API-evolution problem, which can automatically recognize API changes by using the differencing tool UMLdiff [12] and propose plausible replacements based on the detected API changes. AppCatchupDB focuses database evolution and functions at runtime to provide patches.

VII. CONCLUSION

In this paper, we have proposed an approach AppCatchupDB for automatically adapting software applications to the evolution of their underlying databases. AppCatchupDB employs graph-based schema differencing to detect inconsistencies between the database schema expected by application and the supported schema of database. Based on a set of graph transformation rules, AppCatchupDB can infer appropriated adapter for adapting the intercepted queries and patch the inferred adapters into the application at runtime using AOP techniques.

Our empirical study has confirmed that our approach can automatically adapt software applications to a number of common types of database schema evolution. We also find that the results of schema differencing greatly influence the applicability and efficiency of our approach. To facilitate the consistent evolution between database and application, application developers should be careful about their ways of database queries, for example, they should avoid bad smells like statements of "Select * from …".

We have noticed that our hypothesis and approach are still to be further validated with wider range of software database-centric information systems. In the future work, we will do more empirical studies with industrial cases and try to provide more precise adaptation policies for more complex situations.

ACKNOWLEDGMENT

This work is supported by National High Technology Development 863 Program of China under Grant No. 2012AA011202, the National Research Foundation for the Doctoral Program of Higher Education of China under Grant No.20100071110031.

REFERENCES

[1] D. Sjøberg, "Quantifying schema evolution," in *Information and Software Technology*, Vol. 35, No. 1, pp. 35-44, 1993.

[2] A. Maule, W. Emmerich, and D.S. Rosenblum, "Impact Analysis of Database Schema Changes," in *Proc. 30th ICSE, 2008,* pp. 451-460

[3] D. Q. Zhang, K. Zhang, and J. Cao, "A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages," *Computer Journal*, Vol. 44, No. 3, pp.187-200, 2001

[4] D. Q. Zhang and K. Zhang, "Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs," in *Proc. 13th IEEE Symp. Visual Languages*, Capri, Italy, 23-26 Sep. 1997, 284-291.

[5] J. Kong, K. Zhang, J. Dong, and G. Song, "A Graph Grammar Approach to Software Architecture Verification and Transformation," *Proc. 27th COMPSAC*, 2003, pp. 492-497.

[6] G. Zhao, J. Kong, K. Zhang, "Design Pattern Evolution and Verification Using Graph Transformation," in *Proc. 40th Hawaii International Conference on System Sciences*, 2007.

[7] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the prism workbench," *VLDB*, Vol. 1, No. 1, pp. 761-772, 2008

[8] D. Y. Lin and I. Neamtiu, "Collateral evolution of applications and databases," in *IWPSE 2009*, pp. 31-40

[9] P Vassiliadis, G Papastefanatos, Y Vassiliou, and T Sellis, "Management of the Evolution of Database-Centric Information Systems," In *Ist Inter-national Workshop on Database Preservation (PresDB 07)*, Edinburgh, Scotland, UK, 2007.

[10] G. Papastefanatos, P. Vassiliadis, Y. Vassiliou, "Adaptive Query Formulation to Handle Database Evolution," *Forum of the 18th Conference on Advanced Information Systems Engineering (CAiSE 2006)*, Luxembourg June 5-9, 2006

[11] G. Papastefanatos, P. Vassiliadis, A. Simitsis and Y. Vassiliou, "HECATAEUS: Regulating Schema Evolution. Data Engineering," in *ICDE 2010*, pp. 1181-1184.

[12] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *ASE*, 2005, 54-65.

[13] Z. Xing and E. Stroulia, "API-evolution support with diffcatchup," *IEEE Tran. Soft. Eng.*, Vol. 33, No. 12, pp. 818-836, 2007.

[14] Z. Xing, "Model comparison with GenericDiff," *ASE* 2010, 135-138

[15] Mysqldiff, http://adamspiers.org/computing/mysqldiff/

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

[17] 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.

[18] J. F. Roddick, "A survey of schema versioning Issues for database systems," in *Information Software Technology*, Vol. 37, No. 7, pp.383-393, 1995.

[19] GOLDParser Builder, http://goldparser.org/

[20] S. Wu and I. Neamtiu, "Schema evolution analysis for embedded databases," In *HotSWUp '11: Proceedings of the Third Workshop on Hot Topics in Software Upgrades*, Hannover, Germany, 2011

[21] B. S. Lerner, "A model for compound type changes encountered in schema evolution," *ACM Transactions on Database Systems (TODS),* Vol. 25, No. 1, pp. 83-127, 2000.

[22] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," ACM SIGOPS Operating Systems Review, Vol. 42, No. 4, pp. 247-260, 2008.

[23] J. P. Yoon and L. Kerschberg, "A framework for knowledge discovery and evolution in databases," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 5, No. 6, pp. 973-979, 1993.

[24] C. T. Liu, S. K. Chang, and P. K. Chrysanthis, "An entity-relationship approach to schema evolution," in *Proc. ICCI 1993*, pp. 575-578.

[25] C. T. Liu, P. K. Chrysanthis, and S. K. Chang, "Database schema evolution through the specification and maintenance of changes on entities and relationships," *Entity-Relationship Approach—ER'94 Business Modelling and Re-Engineering,* pp. 132-151, 1994.

[26] C. Turker, "Schema Evolution in SQL-99 and Commercial (Object-) Relational DBMS," *Database Schema Evolution and Meta-Modeling,* pp. 1-32, 2006.