Application-Database Co-Evolution: A New Design and Development Paradigm

Michael Stonebraker Dong Deng Michael L. Brodie
M.I.T. Computer Science and Artificial Intelligence Laboratory
{stonebraker,dongdeng,mlbrodie}@csail.mit.edu

ABSTRACT

In a recent IEEE paper [2] we discussed the concept of **database decay**. Specifically, as business conditions change in large enterprises, applications and/or the database schema must change. However, "in the wild" DBAs try very hard to minimize application maintenance by avoiding changing the schema. Instead they "kluge" the schema, thereby cause it to decay over time. In this short paper, we extend the discussion to consider **application decay**. If a DBA "does the right thing" and changes the schema to conform to well known database design principles, then extensive application maintenance is likely to be required. Patching applications will generally cause them to decay over time, as multiple patches, often by different programmers, will generate dirtier and dirtier code. Recent DBMS application frameworks support exactly this point of view. In our opinion, the two extremes (dirty data, clean application) and (clean data, dirty application) are both inferior to a strategy of co-evolution, which we present herein. We briefly outline a prototype co-evolution system and sketch its construction.

1. INTRODUCTION

In a recent paper [2], we presented the concept of **database decay**. In that study we interviewed more than twenty DBAs in large enterprises who confirmed that they do not follow the textbook advice on schema evolution when changes in business conditions require changes to existing databases and their applications. This advice specifies:

- Start with a schema in third normal form (3NF), typically constructed with an entity-relationship (ER) tool, such as erwin;
- Code the application against this 3NF schema using JDBC/ODBC;
- When business conditions change, change the ER model generating a new 3NF schema;
- Reconstitute the database into the new schema;
- Fix the applications to work with the new schema.

We call this advice **data-first**, because it ensures that the database is always semantically clean. However, it will require substantial changes to applications whenever business conditions change. Hence, the data-first tactic leads to **application decay**.

"In the wild", this data-first advice is almost never followed. Instead real-world DBAs try hard not to change the schema, so as to minimize application maintenance. In general, applications are developed in multiple, separate departments. For example, engineering may specify what parts are required for a specific product. Finance may decide what suppliers can sell parts to the enterprise. Lastly, procurement is often in charge of negotiating terms and conditions for suppliers who are being considered to supply a given part. In effect, three different departments code the three parts of this application; hence, application development is dispersed.

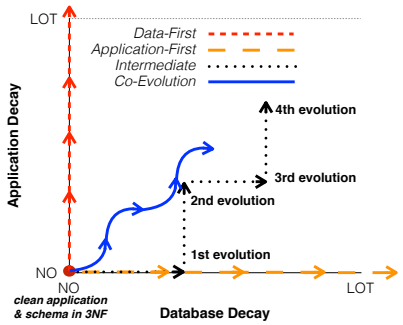

Figure 1: A Depiction of Application versus Database Decay

The typical reason for minimizing application maintenance is to reduce the cost (and risk) in finding and fixing applications across the enterprise. Moreover, maintenance budgets are often dispersed, in the above example to three different departments. As such, a change to business conditions requires a budget to fix applications in multiple departments, which usually does not exist.

As a result, DBAs, who are invariably risk-adverse, almost always choose to leave the existing schema in place or change it minimally, so as to minimize application maintenance. We call this tactic **application-first**, because it attempts to minimize or eliminate changes to application code. However, this tactic causes database decay. Over time the actual schema drifts farther and farther from a clean, 3NF schema. Ultimately the database becomes so rotten, that further changes are impossible and the entire application must be redone.

Figure 1 illustrates these two tactics graphically. We start with a "green field" (i.e. an initial clean database and an initial clean set of applications), which is at the origin in Figure 1. In the real world changes to applications and/or data bases occur frequently because of changes in business conditions, mergers and acquisitions, new government regulations, new business needs, etc. These occur once a quarter or more frequently. Hence, we must deal with frequent and inevitable changes. Data-first and application-first are two distinct tactics that we could follow. If a data-first strategy is employed, the database is modified to be in 3NF and is semantically clean. This causes applications to be patched, re-patched, and re-re-patched. Hence, the database remains clean while the applications increasingly decay and eventually become so rotten that they must be rewritten. Following a data-first strategy means moving vertically along the application decay axis in Figure 1.

If an application-first strategy is followed, then minimizing changes to applications requires the schema to be left intact and the semantics "kluged", inevitably violating 3NF and semantic cleanliness. Hence, application changes are minimized while the database is increasingly decayed. Following an application-first strategy means moving horizontally along the database decay axis in Figure 1. Our DBA study confirmed that the application-first tactic is almost exclusively applied in practice, thus explaining the prevalence of database decay in large enterprises.

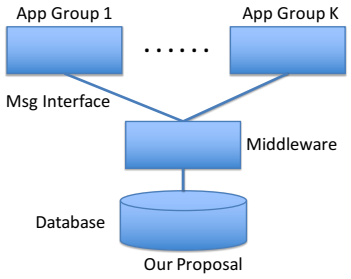

Figure 2: Our Proposed Development Protocol

Moving strictly vertically or strictly horizontally in Figure 1 is not likely to be the optimal strategy. Points in the interior of Figure 1 are likely to result in lower long-term cost of ownership of the database and its applications. We term this two dimensional movement in Figure 1 as **co-evolution** of the database and its applications. In the rest of this paper we briefly describe a system to enable co-evolution.

2. CO-EVOLUTION

Designing for co-evolution means finding modified database and application designs in the interior of Figure 1 that minimize some combination metric that encompasses both application and database decay. From a database perspective, there are many metrics, which optimize database evolution, e.g., distance from a 3NF schema. Similarly, using software engineering and/or other application development metrics, e.g., number of applications impacted, we can optimize application evolution. Applying these techniques separately produces a staircase-like graph of the state of database and application decay as illustrated by the dotted line in Figure 1. Our co-evolution strategy attempts to improve on these disjoint evolution tactics and stepwise changes by evolving the database and its applications together using a combined metric.

It is clear that co-evolution can be accomplished only if there is information on both applications and data available in a central place. As long as applications are coded in individual departments in a distributed fashion, it will be impossible to understand the impacts on applications of a given schema change or of an application change on the schema and other applications. Application development could be centralized; however, this moves the business logic away from the departments that understand it. This is not only politically infeasible; it is also undesirable due to the increased cooperation required between department, application, and database teams. In addition, user-interface issues are largely personal preference items, and best left in the hands of the users.

To navigate this situation, we propose changing the current gold-standard development paradigm of having application groups code directly against the database using ODBC/JDBC. It does not matter whether these accesses are coded directly or generated automatically using Object-Relational Mapping (ORM) or Entity Framework tools (e.g., Entity Framework from Microsoft or JPA from Oracle). In either case, ODBC/JDBC cannot be the interface between users and the DBMS.

Instead, our suggestion is to disallow application groups from coding database accesses in ODBC/JDBC or with ORM or Entity Framework tools. Hence, application groups should be responsible for business logic and user-interface code while DBAs are responsible for database access code. Application groups must negotiate a messaging protocol to send their task to a middleware program that is in the control of the DBA, as illustrated in Figure 2.

In this scenario, the DBA writes the code that implements the tasks specified by inter-process messages. This architecture is reminiscent of micro-services; database stored procedures, and RESTful middleware. The outcome of Figure 2 is a database of applications, i.e., database accesses, or micro-services, consisting of pairs:

(task-name, SQL-code-implementing-the-task)

This database of applications is in middleware and is controlled by DBAs. With such a database it is possible to discover the application impact of any proposed change to the database schema. Specifically, as an offshoot of the Data Civilizer [1] system, we are developing the following modules:

Inspector. This module accepts an original schema, a revised schema, and a collection of SQL database accesses for the original schema, as noted above. It outputs:

- The SQL code that will run unchanged against the revised schema,
- The SQL code that will not run, but which can be automatically converted to a unique replacement SQL code that will implement the semantics of the task, and
- The SQL code for which there is no unique replacement code, and for which a human must consider what to do.

Generator. This module accepts an original schema and a revised schema, as noted above, and generates additional schemas that might be interesting to consider. To do so, it makes use of a database of past schema changes and a collection of user-defined rules. Then it can call the Inspector to run its analysis for the each of these schemas.

Based on this collection of inputs, a human can decide to use data-first, application-first, or some intermediate strategy. For each possible change, he can assess the amount of database decay (e.g., the distance from 3NF) and application decay (e.g., the number of applications that require maintenance). Given a function of these variables we can generate the schema that minimizes the value of this function. In this way, it is possible to choose wisely.

To make use of this strategy, we need a person who is more than just a DBA. He must also be cognizant of the applications that use the database in order to make the above choice. We term this person an **application administrator** (AA) and suggest that enterprises adopt our architecture and utilize the services of an AA.

3. USE CASES

As a proof of concept, we are applying our application-database co-evolution strategy in a data exchange use case within a very large enterprise. The enterprise has more than 100 separate divisions each with their own database, in some cases as large at 18,000 tables. Data is shared between divisions by means of data exchanges that are implemented in terms of micro-services. The development of a micro-service is initiated by a user data access request defined in terms of desired information attributes. The data discovery component of Data Civilizer (discussed in [1]) identifies source database tables and attributes across the enterprise that might satisfy the user request. Data Civilizer then semi-automatically generates queries to extract the requested source data and form the basis of a new micro-service to implement the data exchange.

The micro-services that implements the data exchange will be stored as database access pairs, as noted above, consisting of the SQL and the corresponding data in the source database. Inevitable changes in business requirements will lead to the need for the co-evolution of the source databases and the data exchanges. Hence, this use case will enable us to study the effectiveness of our algorithms in a real world enterprise setting.

4. FUTURE WORK

We are working on methods to estimate and optimize application and database decay. Clearly the number of tasks that need maintenance is a very rough metric, as is distance from a 3NF schema. Based on such metrics, we hope in the future to construct a **recommendation system** that would better assist a DBA in making co-evolution choices. We also want to try out our system on additional enterprise applications "in the wild" to get a better feel for the metrics that would be appropriate.

5. REFERENCES

[1] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.

[2] M. Stonebraker, D. Deng, and M. L. Brodie. Database decay and how to avoid it. In *IEEE International Conference on Big Data*, 2016.