# The Case for Co-evolution of Applications and Data

Jialing Pei[1] jialing@csail.mit.edu
Michael L. Brodie[1] mlbrodie@mit.edu
Ricardo Mayerhofer[2] ricardo.mayerhofer@b2wdigital.com
Michael Stonebraker[1] stonebraker@csail.mit.edu

## ABSTRACT

Database design is the process of designing an initial shared database for some collection of applications, and then evolving this design as business conditions change. In "the wild" changes occur often, usually once a quarter or more.

There are three approaches to such evolution:

**Data First**: This is the approach advocated by all textbooks on DBMS technology. Here, the data base is always kept in 3rd normal form, and substantial application maintenance will be required as the schema keeps changing. Such maintenance will lead to application decay. In this case there is no database decay, but perhaps substantial application decay.

**Application First**: In this approach one tries to minimize application maintenance, while letting the schema of the database decay. Hence, application decay is minimized, but substantial database decay may occur.

Obviously, a mix of these two strategies is possible, which we call *co-evolution*. In this case one uses a holistic view to minimize the composite decay of applications and data.

We claim that the third strategy is the best option. To prove this, we examined the evolution of an application over a 6-year time frame. This application processed checkout requests for a large South American retailer. In the course of 36 evolution steps, we found that 10 followed an application first strategy, 10 followed a data first tactic and 16 followed a co-evolution approach.

As a result, any design tool must be capable of suggesting all three tactics, depending on the circumstances.

This paper presents the details of our study.

## 1. The Systems Coevolution Challenge

### 1.1 Conventional Application Systems Evolution Methodologies

Once an enterprise application system is made operational it must be continually evolved to accommodate new business requirements. Conventional systems design, development, and evolution methods typically focus on producing an "optimal" or "good" initial design, i.e., an 3NF schema that is optimized for the expected workloads and "good" applications. The "optimal", initial design is operation for <20% of the life cycle. The initial design of B2W's checkout was operational for 30% of the 74-month history. There are many methods and tools for optimal initial designs. The far more important period, i.e., >80% of the life cycle, must address constant requirements evolution in terms of schema, applications, and workloads changes. Our research [1][2][5][6] indicates that conventional methods for the dominant evolution period attempt to minimize application maintenance typically dominated by a dominant application often without examining all possible target schemas and almost always without knowledge of all other applications. While optimal for the dominant application, this approach is at the expense of the other applications, the schema and the database resulting in application, schema, and database decay, i.e., increase in complexity and future maintenance costs and decrease in understandability (i.e., maintainability) and performance. All observed application systems exhibit this decay due to the lack of solutions for application and database coevolution for organizational and methodological reasons[1][2].

### 1.2 The Coevolution Challenge

An application system has three layers

- Applications (A)
- Schema (S)
- Database (D)

To address the above challenge, we are interested in schema changes that alter the semantics of a schema that is addressed by schema mapping tools that then require changes to both applications, through application rewriting, and the database, through database migration.

- Applications: A-> A' application rewriting
- Schema: S->S' schema mapping
- Database: D->D" database migration

### 1.3 Existing Tools Support for Application Systems Evolution

A review of over 200 enterprise database evolution management products [4] indicates that most popular evolution products, e.g., InfoSphere, Hibernate, Ruby on Rails, and JPA compliant tools deal with the database-programming language interface, schema mapping, and database migration. The only evolution management products that deal with coevolution (application and database rewriting) are Object or Entity Frameworks. Currently, these tools are one-way, either "Application First" / "Code First" that support modifying applications and producing ORM-based schema fragments, but not mapping or migrating the current schema or database to those fragments [two products: Oracle Java Persistence API (JPA): Microsoft Entity Framework CodeFirst in Microsoft's SSDT (SQL Server Data Tools)] or "Database First" that take a database change and generate ORM-based database access scripts but do not alter existing applications [one tool jOOQ]. Our approach is "Database First"; however, unlike current entity framework tools, our coevolution approach evolves both existing schemas and applications.

### 1.4 Application Systems Evolution Costs

To accommodate continuously evolving business requirements applications systems must be evolved by modifying the schema and the corresponding applications. Each evolutionally change involves two costs: **maintenance** - the short-term cost required to implement the change, and **decay** – a measure of the future cost of maintaining and understanding the schema and the applications. For this study, we used the number of lines of code as a metric. Maintenance was estimated in terms of the total number of lines of code required to implement the change (added + deleted). Decay was estimated in the change in the number of lines of code (added – deleted). These simple metrics are in direct proportion to the cost of current and future cost of systems maintenance. They will be extended in future studies to include other relevant factors, e.g., systems performance for a given workload.

### 1.5 Optimizing Applications Systems Evolution

Our approach is to semi-automate the identification of all target schemas and for each semi-automatically rewrite all impacted applications, estimating for each target schema the application and database maintenance and decay costs to selecting the target schema and rewritten applications that optimizes the developer defined **evolution tradeoff equation** by which the developer balances application and database maintenance and decay costs.

## 2. B2W's Checkout Application Change History

Checkout is a large application developed in ~2008 on an Oracle database that operated until 2016 when it was fully migrated from Oracle to a microservices architecture on Solr, MongoDB, and other data managers.

Checkout ran B2W's front office handling the customer journey from creating a cart to order placement. It consisted of 67 modules the most significant being: cart, customer, inventory, freight, promotion, installment, payment, navigation, order, gift-wrap, buy-later, and account.

We analyzed schema changes related to modules that use Oracle tables and not those that use files (e.g., installment, payment, navigation and promotion used Drools and DRL files, or cart that used memory during a session with no persistence).

Checkout had 45 relational tables in Oracle; 13 used by in freight; 1 by order; 2 by customer; and 5 by stock.

The 74-month evolution history of checkout in a GIT repository contained 60 commits, each involving one or more changes. 45 commits corresponded to 13 Oracle tables involving 8 of checkout's 67 modules.

## 3. Co-Evolution Analysis of checkout: 4 Cases

There are four patterns of database and application decay.

**Little or no Evolution or Decay**

- ~70% of the system (61 of 67 modules (92%), 33 of 45 tables (73%) did not change)
- No database or application decay

**Application Decay**

- ~2% of the system (1 module (1.5%), 1 table **customer** (2%))
- No database decay; modest (21) customer modules changes, hence application maintenance and decay costs; decay indicates continuously increasing application maintenance.

**Database Decay**

- ~2% of the system (1 module (1.5%), 1 table **order** (2%))
- Significant database decay (29 additions) resulting in 53 pages of XML schema with most elements no longer in use. Since no elements were changed or deleted there is likely no application decay. This could not be confirmed since only back office modules, not in the GIT history, access order

**Database and Application Decay**

- ~25% of the system (3 modules (4%), 12 tables (27%))
- Three of **freight's** 7 modules and 12 of the 13 tables were changed. 10 tables were operational versus simple data tables. Database decay increased cumulatively from 0 to 110. The corresponding application changes increased application decay from 0 to 4,826. Hence, cumulatively, freight and its database tables decayed significantly.

Figure 1: checkout Co-Evolution Patterns

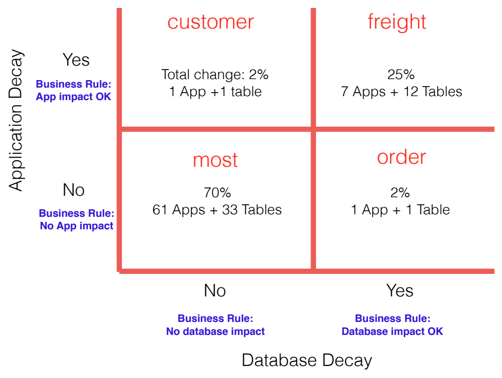


Figure 1 indicates how changes, hence maintenance and decay costs, impact portions of the system, with 25% of the system subject to the most change.

## 4. Detailed Analysis of Four Evolution Cases

The following charts illustrate the how changes impact maintenance and decay (quality) of the four cases. Figure 1(d) indicates how change reduced quality (causes decay) of portions of the system. The quality of freight improved (reduced decay) with little database decay. Other changes did not negatively impact the quality.

Figure 1d: checkout Total Decay

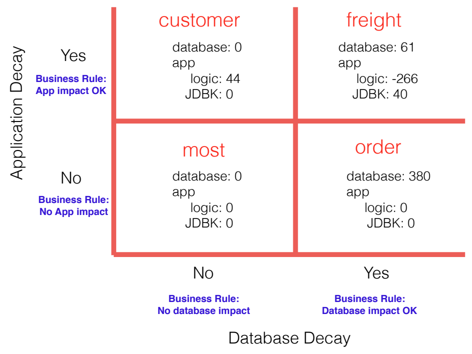


Figure 1(m) indicates how maintenance (the cost of change) is distributed across the system. The quality of freight improved (reduced decay) with most database decay.

Figure 1m: checkout Total Maintenance

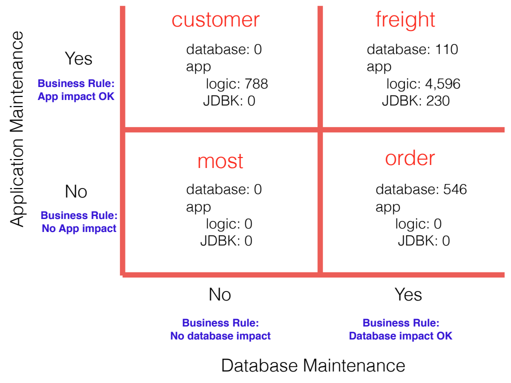
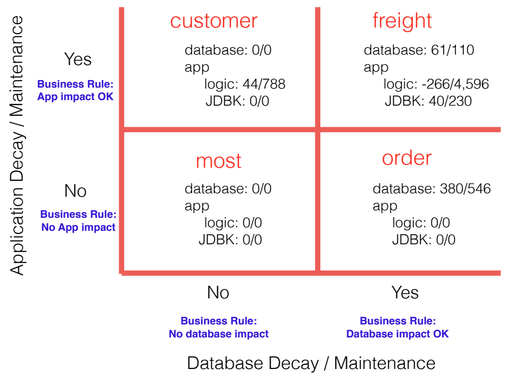


Figure 1(d-m) compares short-term (maintenance) and long-term (decay) costs across the system. Most (70%) of the system did not change in quality. Small portions (2%) decrease a small amount in quality at a largest maintenance cost (customer apps decayed 44 at a cost of 788; order database quality decreased 380 at a cost of 546). Freight, significant portion (25%) of the system, reduced in database quality (61) at a modest cost (110), while the application increased in quality (-266) at significant application (4,596) and JDBC (230) maintenance costs.

Figure 1d-m: checkout Decay / Maintenance



## 5. Quantitative Analysis of Maintenance Costs Due to Schema Change

Figure 3 compares the relative maintenance costs of JDBC application versus database changes for freight. Modest (110) schema changes cause modest (240) JDBC changes. Customer had no JDBC changes. Order had no application changes but large (550) database schema change costs.

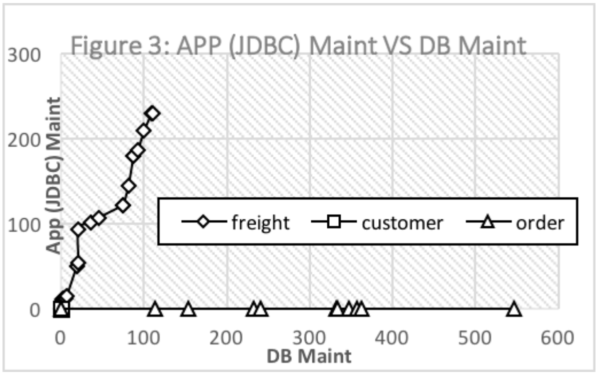


Figure 4 compares the relative maintenance costs of application logic versus database changes. For freight, modest (110) schema changes cause large (4,596) logic changes. Customer had no database changes but some (788) logic changes. Order had no application changes but large (550) database schema change costs.

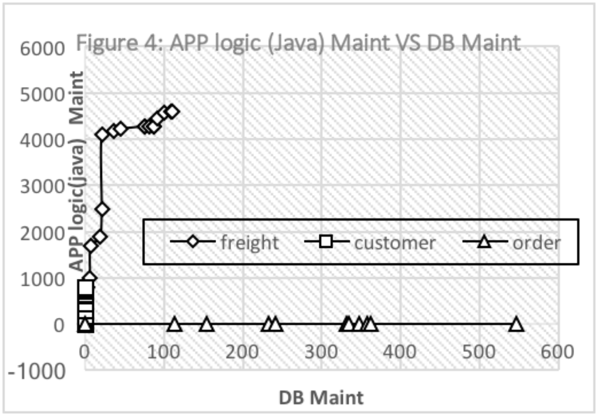


Figure 5 compares the relative maintenance costs of schema changes over time for freight. Schema changes came in bursts of 3, 5, then 8 that corresponded to changing the data on which freight was computed.

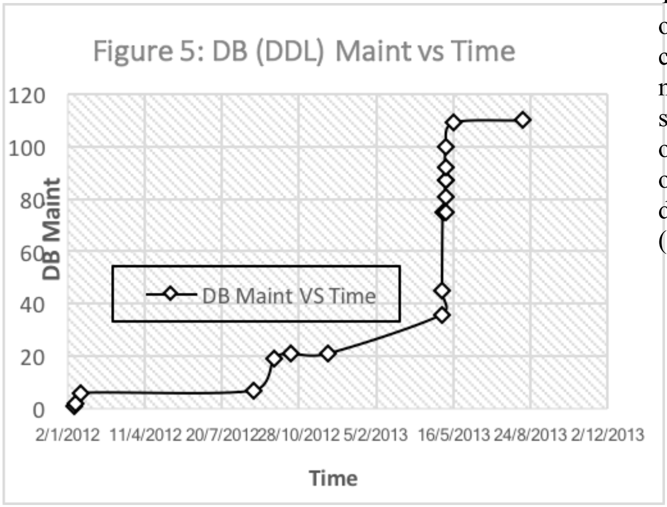


Figure 6 compares the relative maintenance costs of JDBC changes over time for freight. Comparing figures 5 and 6, we see that small DDL changes corresponded to small yet larger JDBC changes.

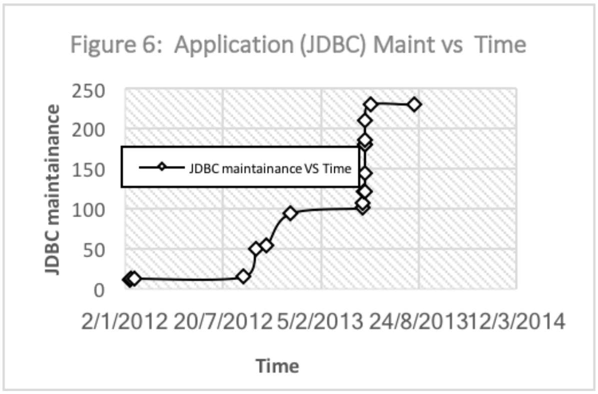
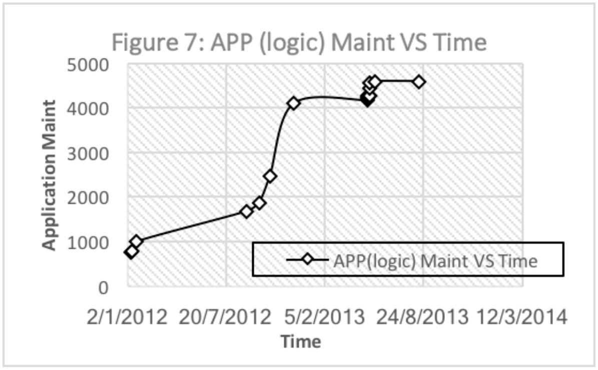


Figure 7 compares the relative maintenance costs of application logic changes over time for freight. Comparing figures 5, 6, and 7, we see that small DDL changes corresponded to small yet larger JDBC changes, and very large application logic changes.



## 6. Changes in Quality

The following charts illustrate the impact of changes on systems quality (i.e., simplicity (versus complexity), understandability, and maintainability) measured in terms of total lines of code in the resulting system. Figure 2 illustrates the three cases of interest in our co-evolution corresponding to the three quadrants of interest (in Figure 1) represented by customer (no database change), order (no app change), and freight (database and app change).

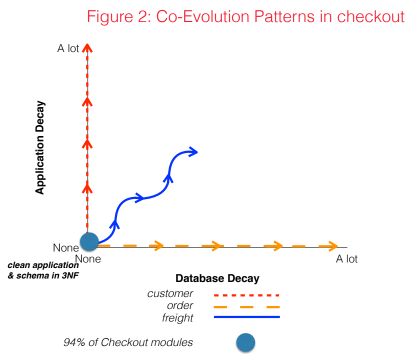


Figure 8 illustrates the impact of change on database (schema) quality (decay) over time. Order changed the most, again in bursts of 1, 5, and 3 changes, only one of which was large (120). While the freight application changed considerably, the corresponding schemas very little (55).

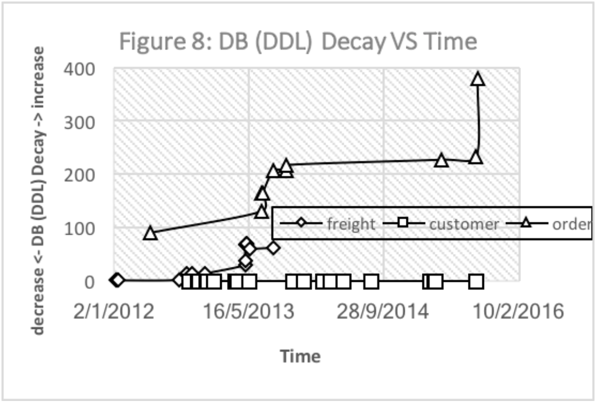


Figure 9 illustrates the impact of change on application logic quality (decay) over time. By business rules, customer and order application did not change in quality. The freight application became more complex (410) then was significantly improved (simplified) by changing freight computation from cost + zip code to region. It became more decayed by making freight computation more complex. Comparing Figure 9 and 8, we see that small schema changes lead to significantly larger application changes.

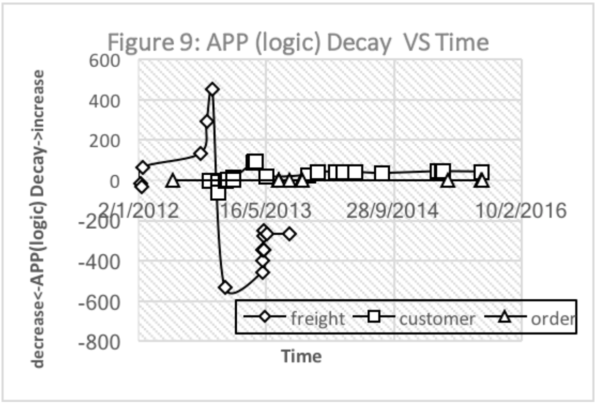
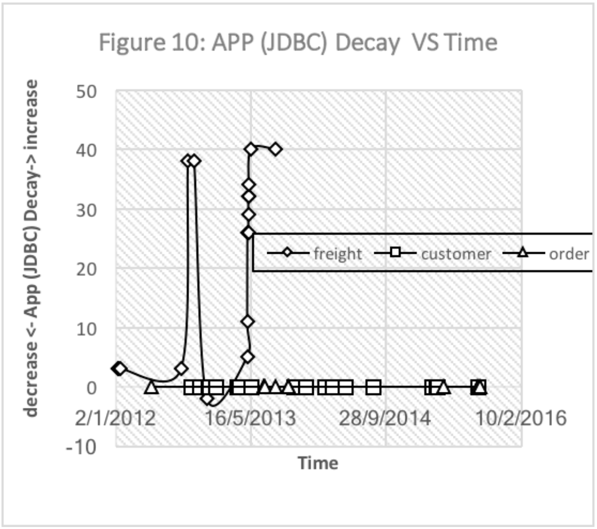


Figure 10 illustrates the impact of change on application JDBC quality (decay) over time. Comparing Figure 10 and 8, we see that small schema changes led to modest (40, 40) JDBC changes) but little decrease in quality.



## 7. Change Generally Leads to Decay

Changes made to checkout generally led to decay, i.e., increased the code base, potentially decreasing understandability and increasing system complexity, leading to increased database and application maintenance. Consider schema (DDL) changes (fig 8) and related application (logic fig 9 and JDBC fig 10) changes. The customer was never changed so zero DDL and JDBC decay. Apparently, a module changed its logical use of customer, unrelated to a schema change. Order was modified continuously by adding attributes until the schema was 50 pages of XML. checkout created an order in memory and wrote the result to Oracle, other than a write, no applications accessed order, so there was no applications decay. The most interesting changes were with freight. The major change was made between 10/17/12 and

12/4/12, when freight significantly simplified its calculation of delivery method offers from being based on zip-code to being based on region and actual cost. This was implemented by dropping two tables and adding two new tables with a few more attributes. Hence, schema complexity rose slightly (first small green rise in fig 8) requiring 40 lines of JDBC change (fig 10) that reduced the complexity of JDBC code (40 fewer lines) but required ~1K of logic to be changed. Since the freight calculation was considerably simpler (first significant green drop in fig 9) hence the application decay decreased (800 fewer lines of code). Starting at 5/2/13, a similar change pattern occurred. The schema change was simple with a small increase in decay (second greed up blip in fig 8), this time requiring significant application (JDBC) change (40 additional JDBC lines) hence decay. Unlike the previous freight calculation change, this change required more logic lines, thus increased decay and the complexity of the freight application by 200 likes of java.

Changes to customer and order were constrained by management rules (do not impact customer information; do not impact order data) freight was not constrained by a management rule so the technically optimal changes were made. Preventing data or application changes for freight would led to greater database and application decay.

This limited evidence is consistent with our survey [1][2]and hypotheses that change invariably leads to increased application and database decay measured in increased complexity, lines of code, and maintenance costs. However, some changes can lead to decreasing decay measured in simpler schemas and applications (lines of code, complexity, and maintenance costs.

## 8. Rate of Decay

Compared with our survey of database and application evolution in industry, checkout was relatively stable. A database of comparable size and complexity to checkout undergoes between 1 and 2 changes/table/month for 20% of the tables. In comparison, checkout had ~1 change per table per month for 10 tables (22%). Of the 45 changes freight, dominated with 25 changes to 9 (operational) tables; hence 0.05 changes/table/month. Order had 1.2 changes/table/month with 29 add-only changes to one table in <24 months. Customer had no schema changes and 21 application changes in 74 months hence 1 change every 3.5 months. These three of 67 (4%) modules accounted for most of the schema changes.

## 9. Checkout Schema Change Characteristics

- **Small schema changes have large application impacts:** Most (66%) schema changes on tables accessed by freight were small, i.e., 110 DDL lines total, the largest being 30 lines, led to significant application changes, i.e., 5,000 lines, of which 4,700 involved application logic (Java) and 300 database accesses (JDBK). Many (33%) were just the opposite: moderate database changes led to few application logic changes. There were two basic patterns (with no obvious interpretation).
- **Bursty** (see Figures 3-10): database decay changes are grouped in 3, 4, and 6 changes at the same time with the majority (8) changing at one time; thus, offering the potential of reordering within the group.
- **Simple**: Most schema changes were simple (largely automatable) involved adding, dropping, or renaming an attribute or a table; the DML changes were largely predictable (automatable) from the DDL changes. Two changes were simple involving adding tables and redirecting foreign keys to reference them. These seem to be automatable with human guidance. There were no complex schema changes in checkout.

## 10. Checkout Co-Evolution Consistent with Research Hypotheses

The above results were consistent with the co-evolution hypotheses and survey results described in [1][2], namely:

1. Due to necessary changes in business requirements, applications and databases must be continuously changed. Continuous change generally leads to increasing database and application decay measured in increased lines of code hence increased complexity and maintenance costs, and reduced understandability and maintainability. However, some changes do lead to decreasing decay, increased quality, measured similarly.
2. Decay is focused on a small number of key operational applications (<10%) and tables (<20%); hence, change seems to be localized to key business entities whose requirements change over time.
3. A typical rate of change is approximately 1-2 schema changes per month per table for <20% of the tables related to <10% of the applications.
4. For each change, choices are made, often implicitly, by the business and the developers,

that directly impact database and application decay.

5. Businesses typically minimize application versus database decay, i.e., applications take priority over the database.

## 11. Conclusions from Checkout Analysis

Our current conclusions from the checkout analysis are as follows:

Schema changes tend to be independent and localized

- **Independent**: Changes arise and must be resolved in no predictable order.
- **Localized**: Changes arise for <20% of tables that correspond to the key business entities that evolve over time; hence, changes tend to happen to the same applications and tables.

Our Co-Evolution method appears to be able assist system evolutions for most schema changes:

- **Fully automatable:** A small number of simple schema changes (rename, change data type) are fully automatable
- **Semi-Automatable**: Most schema changes require varying degrees of human guidance from “confirm or deny”, to “make a choice”, to “provide the correct answer”

Systems changes always involve complex tradeoffs between database decay and application decay as well as other evolution metrics. There are three cases:

1. **Minimize application decay**: Most enterprises attempt to minimize application decay to minimize short-term costs (focus on current change versus long term systems quality), schedules (implement time critical business changes), and due to placing priority on applications over data (applications are individually funded, managed, and measured on delivering value). This typically leads to significant database decay, which in turn can lead to *collateral application decay*. [**Our solution was motivated by and focuses on this most common case**: To minimize application decay for the given application and a schema change, our solution can find a schema design that minimizes related database decay and *collateral application decay* since we consider all impacted applications].
2. **Minimize database decay**: Business circumstances may require minimizing or eliminating database decay for a portion of the schema. Hence, there are only application changes, not schema changes. This typically leads to application decay. [**Our solution**: If an application change is expressed in terms of the database elements accessed, we can identify all applications that access those schema elements and through the database-program binding, all of the code locations that might be impacted, thus aiding redesign.] Warning: This is the general of altering an application, presumably there is research on this. Option: Non-destructive (add only) schema changes. Explore schema changes that do not impact any application, e.g., add attributes.
3. **Minimize Systems Decay**: With no database or application decay preferences, developers must deal with the complexities of database and application decay. This is the general – unconstrained – case. [**Our solution**: In theory, there are system design and development objectives, ideally expressed in measurable factors (metrics). In practice, system design and development objectives are based on business requirements and development methods (e.g., the above cases). Hence, an enterprise should select theoretical and practical objectives for a given system, expressed in an **evolution tradeoff equation** of those metrics that express the complex tradeoff for that system. Our system will be driven to maximize the evolution tradeoff equation.

Example: B2W had clear guidelines for **order** (1. Minimize application decay to avoid breaking back-office app) resulting in 0 application decay and considerable database decay; for **customer** (2. Minimize database decay, except for add-only changes to avoid contacting customers for verification) resulting in 0 database decay and modest application decay; **freight** (3. Minimize systems decay to achieve a business benefit regardless of database or application impacts) resulting in small database and JDBC decay, but significant application decay but achieving a significant business advantage.

In systems evolution, small database changes, hence small amounts of database decay, correspond to large amounts of application decay and application maintenance. Hence, most system decay is application decay that our solution does not address.

- Most database changes, even if small and simple, can lead to significant application decay since most changes focus on changes

business logic that is expressed predominantly in applications.
- Our solution can assist: see Minimize systems decay above.

Hence, it makes sense to experiment with alternative schema designs to optimize the evolution tradeoff equation.

Database schema evolution changes are typically

- Small and simple – possibly simplified to minimize application decay. Our solution: Use to experiment with multiple alternative schema changes.
- Bursty: impact <20% of the tables and <10% of the applications. Option: collect schema change requirements to determine if they can be re-ordered to minimize decay

Theory versus practice: In practice,

- Most schema changes are simple and largely automatable, in contrast to more theoretical work [3] that treats all changes equally. Potentially 95% of practical cases can be covered by our solution.
- Schema mappings can be deduced from DDL changes (ALTER TABLE statements), thus avoiding complex theoretical mappings.

Database Evolution "In the Wild"

Logical (semantic) database schema changes should be made using three DDL statements: Create Table, Alter Table, Create View, and Drop Table. In practice (B2W), makes most changes by editing the original DDL, i.e., adding, dropping, and modifying attributes, apparently to keep schema definitions clean and in one place rather than finding all CREATE, ALTER, and DROP statements.

## 12. Potential Value of Our Coevolution Approach

We claim that our approach can minimize or reduce the rate of applications systems decay thus reducing the costs of evolution, prolog the optimal life of the system, while simplifying, by semi-automated methods, the majority of the life cycle of applications system evolution.

## 13. Future Coevolution Use Case Analysis Requirements

We would love to have another B2W systems application development history for which we would provide a detailed analysis of its evolution and corresponding maintenance and decay costs.

Based on the above analysis, we have developed methods to analyze GIT histories that we would like to apply to additional use cases.

We would like:

- Long term (checkout was 5 years) version control history (SVC, GIT) of an operational (transactional versus data warehouse) database and its applications.
- Interaction with B2W experts familiar with the use case to assist us in understanding the application, the modules, the entry points of key modules to build call graphs and to find the most volatile modules and tables.
- In your software development process for modifying a schema, do you retain a log of the DDL commands that modified the schema? If so, the sequence of DDL schema changes could be used to map source schemas to target schemas; which is usually a difficult task without those steps. Could we get that log?
- Over the life span of this use case was there one or more database and application design, development, and evolution methodologies used for this use case? Where there clear objectives, as in checkout, i.e., do not modify the customer tables, do not alter existing order elements, no database or application restrictions on freight?
- What were the key objectives (metrics) used for database and application design and evolution for this use case? Are there different metrics for management, application teams, database teams?

## 14. Research Questions

- Do hypotheses continue to be true or require refinement?
- Can significant improvements in reducing decay and lowering total cost of ownership be achieved by focusing on a small number of modules (<10%) and tables (<20%).
- Can changes be grouped without causing schedule challenges so that the change order can be optimized?
- In practice, what is the type and frequently schema changes: simple (automatable as in checkout) versus complex (requiring significant human guidance)?
- Can schema change sequences be re-ordered to minimize or reduce database and application decay?

- Understand the impact of schema changes. Do some schema changes always lead to larger application changes than others?
- What are good metrics for "good" database and application designs? Can we combine all conventional database and application objectives in our assessment, e.g., performance, total cost of ownership, understandability / maintainability, schedule? What metrics do/ should enterprises strive for? Is management open to improved long-term database and application metrics (whatever they are) in comparison with optimizing short-term objectives such as schedule, the cost of application maintenance, especially if automation can assist designers to identify relevant choices?
- Do DBAs and application developers specifically (knowingly) avoid / exclude good designs as they may lead to higher database or application maintenance?
- Does management avoid changes due to anticipated costs? If changes could be made simpler and lower cost through automation, might management support more frequent, possibly smaller changes?

## 15. Does Our Co-Evolution Methodology Lead to Optimization?

Premise: You can use our method of optimize systems change choices if by selecting the right sequence of such choices your total systems (application + database) maintenance and decay is lower than it would be for a different sequence of choices. This challenge arises frequently in industry, e.g., in Verizon, since large and medium high priority systems always have a significant number of pending change requests.

In the B2W checkout history, the observed change choices were limited both in number and in the order in which they arose in the business. However, we can use that real data to determine if reordering the choices would produce a different (e.g., lower) total decay (i.e., closer to 0, 0).

Challenge:

- How do you choose between two different co-evolution solutions (each with a specific estimated database + application decay)?
- What is the basis for making such a choice?

Method

- Classify the 16 freight schema changes into X distinct schema changes.
- Understanding the semantics of the X schema changes, experiment to see if they could be reordered (i.e., the intended changes could be implemented) to produce a lower total decay. If not, then there is no benefit for our method. Otherwise, what was the basis used to select a better sequence over a worse sequence?

Consider

- X application decay units correspond to Y database decay units; X app decay units cost Y database decay units

## 16. ACKNOWLEDGMENTS

We gratefully acknowledge the participation of B2W Digital in this research. B2W provided complete access to the checkout code, development history, and experts without which this research could not have happened.

## 17. REFERENCES

[1] Stonebraker, M., Deng, D., & Brodie, M. L. (2016). Database Decay and How to Avoid It (pp. 1–10). Proceedings of the IEEE International Conference on Big Data, Washington, DC. December 2016.

[2] Stonebraker, M., Deng, D., & Brodie, M. L. (2017). Application-Database Co-Evolution: A New Design and Development Paradigm. New England Database Day, (pp. 1–3) January 2017

[3] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," The VLDB Journal, 2013.

[4] Michael L. Brodie, Enterprise Database Evolution Management Products & Prototypes. November 28, 2016; revised August 17, 2017.

[5] M. Brodie, B2W Digital-MIT Collaboration to Minimize Database Decay on Schema Evolution, May 6, 2017

[6] M. Brodie, Database Research Engagement Between B2W Digital and The MIT Database Group, CSAIL, MIT, May 6, 2017.