Database Decay and How to Avoid It

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

***Abstract*—The traditional wisdom for designing database schemas is to use a design tool (typically based on a UML or E-R model) to construct an initial data model for one's data. When one is satisfied with the result, the tool will automatically construct a collection of 3rd normal form relations for the model. Then applications are coded against this relational schema. When business circumstances change (as they do frequently) one should run the tool again to produce a new data model and a new resulting collection of tables. The new schema is populated from the old schema, and the applications are altered to work on the new schema, using relational views whenever possible to ease the migration. In this way, the database remains in 3rd normal form, which represents a "good" schema, as defined by DBMS researchers. "In the wild", schemas often change once a quarter or more often, and the traditional wisdom is to repeat the above exercise for each alteration.**

In this paper we report that the traditional wisdom appears to be rarely-to-never followed for large, multi-department applications. Instead DBAs appear to attempt to minimize application maintenance (and hence schema changes) instead of maximizing schema quality. This leads to schemas which quickly diverge from E-R or UML models and actual database semantics tend to drift farther and farther from 3rd normal form. We term this divergence of reality from 3rd normal form principles *database decay*. Obviously, this is a very undesirable state of affairs, and should be avoided if possible.

The paper continues with tactics to slow down database decay. We argue that the traditional development methodology, that of coding applications in ODBC or JDBC, is at least partly to blame for decay. Hence, we propose an alternate methodology that should be more resilient to decay.

1. Introduction

There has been significant on-going research into schema design methodologies for at least the 40 years since Peter Chen wrote his pioneering paper [1]. There has been work on design paradigms [2], [3], schema evolution [4], [5], model management [6], [7], and reports on a variety of commercial offerings [8], [9], [10]. In our opinion, much of this work is misguided, as we explain in this paper, because it focuses on the wrong metrics. Specifically, the research work focuses on constructing and maintaining "good" schemas. Instead, it appears that most large enterprises actually care about minimizing application maintenance of existing production systems. That causes them to utilize "bad" schemas, and generally to allow "database decay". Our assertion is based on conversations with nearly twenty Database Administrators (DBAs) at three very large enterprises.

The purpose of this paper is to explain why decay occurs, and then to explore tactics that minimize database decay. In our opinion, the most significant contributing factor is the development methodology that is traditionally employed in large organizations, that of coding applications using ODBC/JDBC. Hence, we propose a different methodology that is more resilient to decay.

The rest of the paper is organized as follows. In Section 2 we explore the environment we see in most large enterprises. This serves to frame the reasons for database decay, which we explain in Section 3. Sections 4 and 5 then turn to antidotes for decay. In Section 4 we consider **defensive** database design and application construction, and show how this can help. Section 5 then explores a design paradigm that is more resilient to decay.

2. The Design Environment in Large Organizations

2.1. Longevity

In this section we explain the environment found in most large organizations. First, as has been pointed out by many authors, databases last for a long time. What is less well known is the frequency with which business conditions force them to change. New applications for the same data, changed business requirements for the existing applications and mergers and acquisitions (M&A) all contribute to frequent changes. As a figure of merit, we assume databases change once per quarter, although the data in [11] suggests it is even more frequent.

2.2. Decentralization

Researchers often assume that a database is controlled by a DBA, who is embedded in the application group that is writing or maintaining the applications that access a given database, as noted in Figure 1. We call this model "centralization" as all aspects of database design and deployment are controlled in one place. Although there are examples of this organization, especially with small-to-midsize applications,

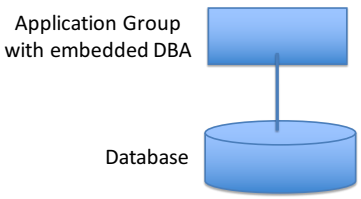

Figure 1. Assumed Enterprise Organization

it is unusual for large systems to be developed in this fashion. Instead there are several application groups, each reporting to a different boss, and responsible for writing some piece of the overall application, as noted in Figure 2. For example, an enterprise might have billing, procurement, and collections departments, each with a programming team. The DBA for the database could be in one of the participating departments or he could report to central IT. For generality, we show the DBA separate from the application groups in Figure 2. One should think of this as "decentralized" development.

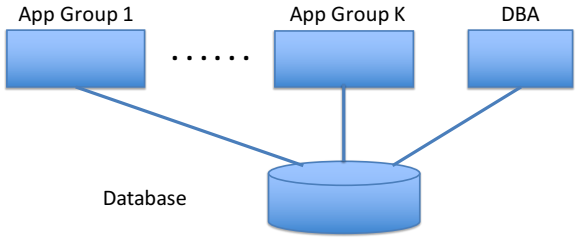

Figure 2. More Typical Enterprise Organization

Initially, there is a clean sheet of paper, and the various groups build their applications, often with different delivery dates. Moreover, at the time that the first department goes "live", other departments may not even have identified their requirements. This leads to a phased design and rollout. After initial deployment, the maintenance budget for each portion of the application is invariably controlled by the individual departments. Hence, any application maintenance due to a schema change must be coordinated with multiple departments, each with their own priorities.

In this paper, we assume a decentralized environment for application management.

2.3. Kinds of Changes

Another dimension in enterprise environments concerns the kinds of changes that occur over time. Changes can either preserve semantics or not. A typical example of the first type is to split apart a table, such as:

$$\texttt{Example-1}(key, att_1, \cdots, att_n)$$

into two tables:

$$\texttt{Example-2}(key, att_1, \cdots, att_k)$$
$$\texttt{Example-3}(key, att_{k+1}, \cdots, att_n)$$

Specifically, one collection of attributes might deal with an employee's personal life while the other deals with work-related matters. Semantic clarity would result from such a split, which can be readily inverted by performing a join. In this case, it is conceivable to make the change and map all queries and updates from the old schema to the new one. As such, application maintenance can be completely avoided, and the work in Prism++ [11] indicates how to do this when the schema changes in specific ways.

On the other hand, there are many changes that result from changed business conditions. For example, an enterprise might require that each part be bought from a single supplier. At some point the CFO might decide that the enterprise could save money by having different suppliers in the various regions. As such, a relationship between parts and suppliers that was previously 1−N has been changed to being M−N. In this case, application maintenance may be unavoidable.

As we will explain in the next section, DBAs tend to avoid making unnecessary changes. Hence, semantics preserving changes are usually optional and are avoided "in the wild". Hence, in this paper we will focus on schema changes which change the semantics of the applications.

In summary, we focus on decentralized application development and on schema changes which alter the semantics of at least some of the applications. In the next section, we explain why the traditional schema methodology fails under these conditions.

3. Why Conventional Database Design Fails

3.1. The Traditional Methodology

The conventional wisdom, which dates from the 1970's, is to perform logical data modeling using an Entity-Relationship (E-R) or UML tool. The result is a graph where entities with their attributes (as nodes) are connected by relationships with their attributes (as arrows). Once the DBA is satisfied with his graph, he can push a button in a variety of current day tools to produce the following relational schema:

- A table for each entity along with its attributes.
- A table for each M−N relationship along with its attributes.
- For each 1−N relationship, add a foreign key to the "N side" entity along with any attributes of the 1−N relationship.

Consider, for example the E-R diagram in Figure 3:

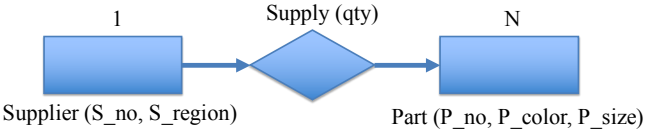

Figure 3. A Typical E-R diagram

Figure 3 shows two entities, Supplier and Part, along with a 1−N relationship indicating which supplier supplies which parts. In this case, we specify that the relationship is 1−N, i.e. each part is supplied by a single supplier. Any E-R tool will produce the following two tables for this diagram:

Supplier (S_no, S_region)
Part (P_no, P_color, P_size, S_no, qty)

Figure 4. Initial Tables

There is a table for each entity, and the relationship is encoded in the foreign key S_no with its attribute qty. This collection of tables is in 3rd normal form, and is thought by DBMS professionals to be a "good" schema. Hence, this relational schema is defined to a DBMS and the application is coded, presumably in JDBC or ODBC, against this schema. If, for some reason, the data or the business logic changes, then the E-R diagram is updated, causing the physical schema to change to a new one. However, over time the database remains in 3rd normal form by following this methodology. Application maintenance can be lightened or eliminated by using the view subsystem in present day DBMSs. Hence, the previous schema can be defined as a view so that previously written programs use the same schema they were written for, and thereby continue to run.

3.2. Why the Traditional Methodology Fails

In our simplified world, suppose applications are written in a decentralized way. For example, the attributes in the Part table could be specified by engineering. Suppliers could be chosen by finance, and procurement could be charged with picking the qty to be supplied.

A transaction from engineering could be:

T1: Add a new part to the Part table, with its associated attributes

A transaction from finance could be:

T2: Delete supplier XXX

And one from procurement could be:

T3: Change the qty supplied for Part XXX to YYY

It is straightforward to implement this collection of transactions as applications A1, A2 and A3. Of course, a real deployment would be more complex, but this simplified one will serve our purposes. The result is a collection of production applications running against a production database.

Now, suppose the company makes a strategic decision that each part that it purchases may come from more than one supplier, as long as the suppliers are in a different region. Maybe the goal is to stimulate competition; "in the wild" such modifications of business conditions occur regularly, as noted in Section 2. This change of business logic impacts our E-R diagram. Specifically, it changes the relationship, Supply, from 1−N to M−N.

Following the traditional wisdom, the DBA will change the E-R diagram appropriately, and E-R tools will produce a new physical design, as follows:

Supplier (S_no, S_region)
Part (P_no, P_color, P_size)
Supply (P_no, S_no, qty)

Figure 5. Final Tables

The standard wisdom continues by stating that one should define the initial tables (Figure 4) as views on top of the final set of tables (Figure 5). Notice that Supplier is unchanged while Old Part is a join of New Part and New Supply. It is straightforward to define these views in many SQL DBMSs. There are three problems with this approach:

Problem 1: The views, so defined, are unlikely to be updatable. This could happen because of semantic ambiguity or because of shortcomings in the view support system. For example, in SQL Server, one cannot insert into a view that is derived from multiple base tables. Other DBMSs have similar restrictions. In our example, T1 will fail on most DBMSs, because it is inserting a record into a view, and this operation is disallowed in all view subsystems that we are familiar with.

Problem 2: The semantics of the application may change. For example, T3 in the original schema updates the qty supplied for the only supplier of Part XXX. After the change, there may be multiple suppliers of Part XXX. Should the application change qty for all suppliers or only one. If only one, then which one? In effect, a human has to examine T3 to decide what to do.

Hence, the unintended consequence of these problems is that applications across the enterprise must be found and perhaps corrected. There are two dramatic consequences of this course of action.

Consequence #1. There is substantial risk. ALL of the applications must be found and corrected, and then all must be cutover at the same time. Usually, a few will be missed, causing downstream failures. Few DBAs want to take on this risk.

Consequence #2: There is often no budget for global maintenance. In other words, the organization making the change does not have budgeted resources to fix the issues in other technical groups, so it is impractical to proceed in this direction.

As a result, a very popular tactic is to leave the schema in Figure 4 intact. Instead, the changed business logic is supported without changing the schema. One way to accomplish this is to duplicate parts data in the Part table for each supplier that supplies the part. Therefore, if two suppliers both supply part XXX, then there will be two records in the Part table, one for each supplier, each with a different qty and region. In effect, we just dropped the primary-key-foreign-key (PK-FK) constraint on S_no in the Part table and converted the key of the Part table to be (P_no, S_no,

S_region). The result, which we call the **kluge**, is not in 3rd normal form, so professionals would consider this a bad design. However, it has a huge advantage:

The applications A1−A3 will continue to run.

Although we have moved to an inferior data base design, we have lowered or eliminated the need for application maintenance, so most enterprise DBAs would claim that this is the preferred strategy. Specifically, T1−T3 will continue to function, and there is no ambiguity about the execution of T3; the qty supplied is changed for all suppliers on Part XXX. To avoid application maintenance, however, a third problem must be addressed.

Problem 3: Applications have to avoid running certain problematic SQL commands. For example, suppose there is a transaction T4 as follows:

T4: Find the qty supplied for a given part

T4 will fail when we move from Figure 4 to Figure 5, because we must add an aggregation operation to the query in order to get the correct answer. In addition, the same aggregation is required in the replication alternative noted above. Hence, the absence of application maintenance is dependent on avoiding certain constructs. We term this **defensive programming** and we discuss this option in Section 4.

Note clearly that leaving the schema untouched results in an E-R diagram for the database that is invalid. In fact, there is no E-R diagram that will produce the kluge schema. The minute a DBA leaves an earlier schema intact, so as to avoid risk, the E-R diagram diverges from reality. We have talked to nearly twenty DBAs, and all report that they do not use E-R tools, because they do not reflect reality in the database. Instead all report performing database design on the tables themselves. Several DBAs indicated that they use E-R tools for the initial (green field) design, and then abandon them during the downstream maintenance phase.

We have observed that real DBAs go to considerable effort to avoid changing the initial schema. Hence, their true objective appears to be to minimize or eliminate application maintenance. This is very different than the objective of maintaining a good logical data base design. A consequence of this state of affairs is that few DBAs will make any optional changes. After all, this just entails risk. As a result, schema changes which preserve semantics are typically avoided, since they are optional. Therefore, we will look at database design very differently than has been done in the past, as will be shown in Sections 4 and 5.

In summary, our thesis is the following. DBAs avoid schema changes, if at all possible, since they entail risk. DBAs avoid semantics-preserving (and therefore optional) changes, since they entail risk. Essentially all modifications that change the semantics of the application will avoid maintenance for some SQL commands and not others. Therefore, defensive programming is highly desirable to avoid problematic SQL. Otherwise, a DBA will have to examine all transactions to see which ones can be run intact and which ones require maintenance. As a result, databases decay over time, and the schema drifts further and further from one that obeys the standards of good design.

3.3. Other Examples

In this section we give several additional examples of changes that cause problems.

3.3.1. Changing a Relationship from M-N to 1-N. Consider starting with the design in Figure 5. Suppose enterprise management legislates that there will be a single supplier for each part. Obviously, the extra suppliers must be identified and eliminated. This will almost certainly take weeks-to-months to accomplish; in the meantime, the schema cannot change. In this process, the extra records will gradually be eliminated. At the conclusion of this exercise, the schema can be converted to that of Figure 4. However, this is a dangerous step as Figure 5 will need to be defined as a view, and may not be updatable. Hence, a more likely solution is to leave Figure 5 intact, but modify the key of the Supply table from (S_no, P_no) to (P_no). Again, the E-R diagram for the schema is incorrect. However, all applications will continue to run.

3.3.2. Dead Attributes. Now, suppose one of the applications decides to add an attribute to the Supplier entity, say S_rating. The conventional wisdom is to add this attribute to the Supplier table, producing New_Supplier. Then, define Supplier as a view on New_Supplier. Now suppose the application stops using the S_rating attribute. The appropriate action to take would be to drop the attribute from the stored table. Depending on the system being used, this could require a (perhaps substantial) reorganization to remove the offending attribute. More seriously, other applications may fail because they read S_ratings, even if they don't actually use it in their logic. As noted in Section 2, such a change is rarely taken by DBAs. Instead, a typical action is to simply leave the attribute in the database, rendering it a "ghost" or "dead" attribute. Again, the E-R diagram becomes more distant from application reality.

Now suppose the application which deleted S_ratings decides to add a new attribute, say S_date_incorporated. As you can imagine, the easiest thing for the developer to do is simply reuse the S_ratings field. Again, the E-R diagram drifts further from the truth. After a few of these modifications, you can imagine that developers simply don't pay attention to the E-R diagram, as it bears no resemblance to reality.

Now consider a second application that uses the Supplier schema and wants to add a different attribute to Supplier, say S_terms, to indicate what payment terms are expected by the supplier. This application wishes to add an attribute to Supplier, which is now a view. AddAttribute is not supported by any view subsystem we are familiar with. Hence, this sequence of two schema operations fails. A common solution is to add a few phantom attributes to the schema, which may be used for downstream maintenance, such as in the above example.

3.3.3. Problems with Timing. Return to the schema of Figure 4. As noted earlier, there may be three departments, e.g., engineering, finance and procurement, that are responsible for the composite application. Engineering may be 3 months ahead of finance, in which case the Part table is populated, but Supplier and qty information has yet to be filled in. Obviously, the E-R diagram is currently incomplete, and certainly cannot be trusted.

Clearly, a complete application will have dependencies of this sort. To deal with this situation, there must be information stored somewhere about these dependencies and the timing for their resolution. It is certainly plausible to put such information into an E-R diagram, but one now has a cross between a project management system and a database design tool.

3.3.4. Mergers and Acquisitions. Now suppose the company in question buys another company, B, and merges the two Supplier databases. The obvious way to do this is to add a field to the Supplier table called "origin". Then, we define two views one with the suppliers from B and another view with the remainder. Effectively, the real database has a new key (S_no, origin). However, both views have an effective key of S_no, so both original applications can continue to run.

However, this elegant solution is rarely possible because of semantic mismatches between the two Supplier databases. For example, suppose the two Supplier tables have different semantics for S_region. For example, one table might use (City_name, State_abbreviation) while the second uses State_name. If we merge the two tables and define a view as suggested above, then the two original applications will continue to run. However, new applications on the merged table will assuredly fail. Although the data type of S_region can be a string, it is the union of the two originating data types. Any command with a predicate on S_region will thereby fail, unless application logic is made aware of the situation. Effectively this creates an attribute definition, completely isolated from the schema, an obviously undesirable state of affairs.

A more likely tactic would be to leave the two original tables in the database. The original applications continue to run. New applications must be coded against two tables rather than one; however, they are no more difficult to code than with the other solution. Moreover, an attribute definition entirely in user code has been avoided.

3.3.5. Performance Problems. If the current schema has performance problems either because of response time or throughput issues, then three popular tactics are often employed. First indexes can be added, dropped or changed. This has no impact on whether applications continue to run or not. Hence, it is a **benign** change. The second is to add a materialized view (MV) to the database that will deal with the offending response time issue. If such a MV exists, then this, too, is a benign change. Of course, there is no free lunch and the cost of updating the MV must be considered. Also in many systems MVs are asynchronously updated, so the MV is fundamentally out of date. The last option, that of changing the schema, is insidious. Usually, some join is the offending command, and an option is to pre-join the involved tables. In the schema of Figure 5, one might run the following query:

```
Select Sup.P_no, S.S_region
From Supply S, Supplier Sup
Where Sup.qty > 100 and Sup.S_no = S.S_no
```

This requires a join between Supply and Supplier, which may be deemed a performance problem. One can eliminate this join by converting the schema to that of Figure 6.

Supplier (S_no, S_region)

Part (P_no, P_color, P_size)

Supply (P_no, S_no, qty, S_region)

Figure 6. A Possible Replacement Schema for Figure 5

This schema eliminates the join, by duplicating S_region. Of course, the result is not in 3rd normal form and does not correspond to any E-R diagram.

3.3.6. Summary. In this section, we have discussed common schema changes that occur "in the wild". These were:

- 1−N to/from M−N relationships
- Dead attributes
- Timing problems
- Mergers and acquisitions
- Performance problems

These changes occur regularly, often once a quarter or more often. After a few of these changes the E-R diagram is divorced from reality and becomes useless as a design or documentation tool. Therefore, few DBAs pay any attention to such E-R diagrams, if they exist at all. This causes databases to **decay**, i.e. the schema for the database drifts further and further from any E-R diagram and further and further from a "good" design that obeys the various normal forms. After a while the schema becomes so **rotten** that no further changes can realistically be made, and a complete rewrite is the only way to move forward. Obviously, this is a lousy state of affairs. The remainder of this paper suggests tactics to slow down or avoid decay. In Section 4 we explore how to lower the amount of required application maintenance by using either **defensive schemas** or **defensive application development tactics**. In other words, in the presence of schema decay, these tactics result in applications that require less maintenance. Then we follow in Section 5 with a totally different application paradigm that should be more effective at decay prevention. We compare our approach to related work in Section 6.

However, first we present a non-solution, namely moving to a higher level of abstraction. For example, why not code against an E-R diagram rather than a collection of tables? Unfortunately, this does not solve any of the issues raised earlier in this section. Specifically, all of the problems either

were caused by the E-R diagram changing or in shortcomings of the relational view system. Neither issue is solved by changing the abstraction level. However, code maintenance would be easier, since it is a higher-level notation. A similar comment would apply to coding in some other higher-level notation, such as an object model.

Also, any such change would require an entire generation of DBMSs and tools to convert from ODBC/JDBC to something else. This is clearly a big issue.

Another possibility is to ask DBMS vendors for schema versioning. If this existed, it would certainly make changes easier. However, to avoid application maintenance, a SQL command on the old version must be automatically mapable to the new schema. Since this may or may not be the case, schema versioning will not be a magic bullet.

4. A Better Technique – Defensiveness

In this section, we focus on defensiveness as a tactic in the presence of the changes we noted in the previous sections. In Section 4.1, we consider application level tactics, following in Section 4.2 by schema level tactics.

4.1. Defensive Applications

A first possibility is to use some sort of master data management (MDM) scheme whereby the names of entities are stored in an MDM system along with their table names and attributes. Then, all applications would be expected to use MDM notation for all DBMS objects, and some of the view problems are potentially avoidable.

A second thought is to avoid brittle constructs, such as

```
Select *
From ...
```

which are challenging in the face of changes in the physical tables. A little prevention can avoid a lot of downstream sins. Also, if one is reading JSON data types, one should always code ultra defensively, since the composition of the object may well change.

As a final example, if one is retrieving a data element from the schema in Figure 4, such as the qty supplied for a given part:

```
Select qty
From Part where P_no = XXX
```

then this code will be brittle for changes such as the one in Figure 5. In this case, the retrieve has to be converted to an aggregate. However, if we code defensively, for example, using the following:

```
Select sum(qty)
From Part where P_no = XXX
```

then this code will execute correctly on a view defined from the real schema in Figure 5. Of course, coding defensively trades lower performance for better resiliency to changes. An alternate approach is to delegate all such queries to a companion decision support data warehouse, and keep them out of the operational system.

The conclusion of this section is to always write code that is resilient in the face of possible changes to the underlying tables.

4.2. Defensive Schemas

Obviously, the schema in Figure 5 is a more defensive one than the schema in Figure 4. At the expense of lower performance in the case when the relationship is 1−N, it will support changing the relationship to M−N without any code changes. For changes that seem plausible off into the future, one is better off with a defensive schema, since this avoids a much worse situation if the change occurs. In the case of the data in Figure 4 a "good" schema will have two tables. However, a more defensive schema (Figure 5) will have three tables. Of course, a defensive schema must inevitably give up on some automatic integrity constraints, such as the automatic enforcement of a 1−N relationship. Such data integrity issues must be handled in application logic, which is considerably more painful than inside the DBMS or by some sort of Business Rule Management system (BRMS).

The ultimate defensive schema is to decompose every table into a collection of binary tables of the form:

$$\{\textit{(key, attribute)}\}$$

This schema will have one table per attribute and will survive many changes, without the necessity of recoding. Such schemes are compatible with a key-value (KV) store, and we will call them "tall and skinny", as opposed to the "fat and wide" ones produced by an E-R methodology.

An alternate methodology is to encode all of the binary tables into a single triple store of the form:

$$\{\textit{(entity-id, entity-attribute, entity-value)}\}$$

Again, we get a "tall and skinny" representation. For clarity in the next section, we will consider the KV representation.

Of course, tall and skinny schemas pay a big performance price, since assembling a record with K non-key attributes will require K joins. This will be orders of magnitude slower than the single read required in a standard schema. Hence, we view a reasonable metric of defensiveness to be the number of tables in the schema. More tables means a more defensive schema, but of course offers lower performance. In effect, there is a delicate balance between performance, integrity control and schema defensiveness, which we crudely model in the next section.

4.3. A Defensiveness Model

Let N be the number of tables in the schema. As noted above the larger the N the more defensive the schema will be. Let P be the performance of the schema on the customer workload. This will have to be estimated, but in general will be monotonically decreasing in the number of tables in the schema. Let S be the survivability of the schema to changes

of the sort mentioned in this paper. It is an estimated number between 0 and 1.

Let the defensiveness of a schema be:

$$D = S * P * N$$

A good schema will have a large D, and DBAs should strive to maximize this quantity.

4.4. Surviving the Common Changes

Judicious replication when moving from a 1−N relationship to an M−N relationship can be supported by defensive programming. One simply assumes there might be a multiple record return and adds some way to cope with that occurrence, such as adding an aggregate.

Moving from M−N to 1−N can be accomplished by changing the key of relationship table and by defensive programming. Most previous code should continue to execute correctly.

The use of phantom attributes, dead attributes, and reuse of attributes will allow the addition and perhaps subsequent deletion of attributes to work correctly.

Lastly, avoiding table merge, when one enterprise buys another, will allow current programs to continue to run, generating problems for any new code.

Using defensive programming, defensive schemas and appropriate kluge schemas will allow many popular changes in business logic to NOT require application maintenance. This will clearly result in lower maintenance costs than using the traditional wisdom. Of course, there is no free lunch and the ultimate cost is database decay.

In the next section we consider an alternative approach to schema evolution, which will effectively force centralization. This technique will allow additional options, not possible just using defensiveness.

5. A Better Technique−A New Application Development Paradigm

5.1. Our Proposal

One problem that contributes significantly to database decay is the decentralization of application development. It is virtually impossible to figure out the implications of schema changes on applications, since they are in multiple departments in the enterprise. Obviously, one needs to decentralize such development, since the business logic has to be implemented by the relevant departments. In the traditional architecture, this means that SQL interactions are also decentralized.

Our proposal is to continue to decentralize development of the business logic, but to centralize SQL interactions. Figure 7 illustrates our proposal. Here we see the same K application groups writing business logic as was noted in Figure 2. However, instead of coding in ODBC/JDBC, we require them to specify DBMS interactions differently.

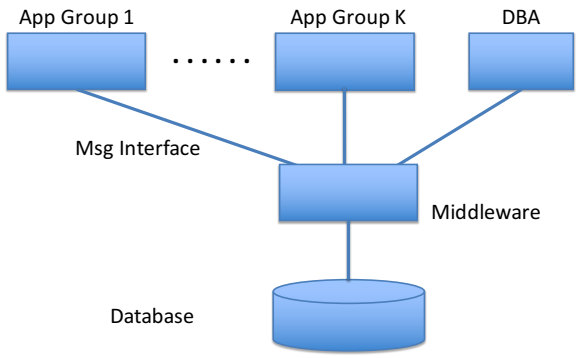

Figure 7. Our Proposal

Specifically, we require them to use a messaging interface as noted in Figure 7.

In Figure 7, an application must send a message to a middleware system in an agreed-upon format. Note that such a message can be asynchronous (message) or synchronous (RPC); there are advantages to both options, so we will choose the asynchronous option for our exposition. There are many possible formats for such a message; one can use a web services paradigm, a micro services paradigm, a remote procedure call paradigm, a stored-procedure DBMS paradigm, or use one of the popular messaging systems.

Our goal is to prevent database decay by making it as easy as possible to change the schema to a "good" replacement schema and then automatically map the old SQL commands to their replacements. As a result, we explore an interface using SQL to express DBMS semantics. However, nothing in our proposal disallows some other syntax for DBMS interactions.

Hence, one such message might be:

(Salary_Raise, John, 10,000)

Here, the intent is to change John's salary to $10,000. The middleware receiving this message in turn executes the following SQL on behalf of the user:

```
Update EMP (set salary = 10000)
Where name = 'John'
```

and returns a completion code in a separate message. As you can imagine, the receiver of the message must execute the purpose of the message by constructing the appropriate SQL. This design is, of course, not very flexible since an application developer with a new requirement must get a new message approved by the owner of the middleware system. However, it has one big advantage. If the schema changes, then the application logic that interacts with the DBMS is centralized in the middleware message processing system. As such, it is easier to find the code that must be changed and then "do the right thing". In effect, this architecture centralizes database management commands in what would otherwise be a decentralized application development world.

In the rest of this section we explore this approach in more detail. Our design methodology can be summarized as

follows:

1) The DBA uses his favorite mechanism/tool to build an initial schema, which we term S_1.
2) Users then send messages to middleware, and a DBA implements the actual SQL to support them. In Section 5.3 we indicate a tool that can help with SQL composition.
3) The schema is subsequently changed to S_2, and some maintenance may be required. Our goal is to make this maintenance painless enough that the DBA will choose to migrate S_1 to a "good" schema, rather than choose a poor one. In this way, he can prevent or slow down database decay.
4) In the general case, we are moving from S_i to S_{i+1}, and the general problem statement is the following: We are given an initial schema S_i, a collection of messages, $M_1, \cdots, M_k$ and corresponding SQL, $Q_1, \cdots, Q_k$. The goal is to identify the replacement SQL, Q_1', $\cdots$, Q_k', with as little effort as possible.

For simplicity, we assume that messages do not commit transactions as a side effect. A specific commit message must be generated from an application. Hence, there will be no discussion in this section of transaction boundaries.

Notice that there are two issues to deal with. First, there is the question of how to negotiate with the application developer to construct the correct SQL that specifies his business logic. This topic is discussed in Section 5.3. The second issue, namely how to automatically or semi-automatically construct the replacement SQL that must be run following a schema change is treated in Section 5.4. Section 5.2 specifies some preliminaries that are required for either solution.

5.2. The Assumed Structure

We assume a transaction processing environment in which the SQL is limited in scope. Decision support environments tend to have much more complex SQL, which may not be amenable to our proposal. Specifically, we assume the following:

Entry point: There is an entry point into the database provided by the application developer. In the example above this might be "*name = John*", which specifies a record (or records) in some table in the database. We assume that the application developer specifies whether there should be a single record matching the entry point or that there can be multiple records. There may be multiple entry points, but for simplicity, we assume there is only one.

Target: There is a target of the application, which specifies the record to be updated or retrieved. In the example above, this would be "*salary = 10000*". Again, this must be provided by the application developer, who also specifies whether he is expecting one record to qualify or multiple records. Not surprisingly, there may be multiple targets, but for simplicity, we assume there is only one. Hence, the entry point and target can be 1−1, 1−N, M−1 or M−N, and we call this the **application template**.

Join path: Since the entry point and the target may be different tables, there must be some join path that connects the two. We assume this must be a sequence of PK-FK relationships. Of course, there may be multiple such paths in the large schemas of enterprise applications.

We turn now to assisting the DBA with join path selection. We have built a prototype **data stitching** system called **Data Civilizer** [12], which examines the tables in a database and finds all PK-FK relationships. These relationships are edges in a graph whose nodes are tables. In addition, Data Civilizer can find approximate PK-FK relationships since data "in the wild" often contains errors. A human administrator can specify how many errors can be tolerated before the appropriate edge is invalid. This stitching system can be run at any time to build this graph anew, based on the current data in the tables and the current schema.

5.3. Initial SQL Construction

Data Civilizer also contains a discovery component, whereby the DBA can find columns that contain a specific value, among other features. This component is detailed in [12]. Using this component, the DBA can find possible entry points and targets and then negotiate with the application developer as to which one(s) meet his needs.

Following this, data stitching can be run to find all join paths that connect the entry point and target. Filtering out the ones which have the wrong application template yields the actual candidates. The DBA can negotiate with the application developer to decide which join path meets his needs, after which constructing the actual SQL can be performed automatically.

5.4. Replacement SQL Construction

We write this section assuming a "just in time" resolution system. For example, RDBMSs are routinely criticized for requiring a "schema first" methodology. More recent "NoSQL engines" support a more flexible "schema later" policy. In this section we defer the commitment to a schema until the first execution of a given application message. This offers ultimate run-time flexibility, and also supports the possibility of changing the schema in between executions of a given message.

After initial construction of the appropriate SQL, any message is marked as "production", and all subsequent executions merely perform parameter substitution into the stored SQL before execution. Over time, messages may be added or subtracted at will and the above procedure is followed.

At some later time, the schema is modified using a collection of schema directives and then a bulk copy operation is activated to populate the new schema from the old one. When the change is finalized, Data Civilizer is activated and marks all existing messages as "tentative" and rebuilds its

graph. At this point, we have the old graph, the new graph and the old SQL for the messages.

When a message is received by middleware in tentative mode, Data Civilizer first identifies the entry point and target in the new graph. In general, schema changes are fairly incremental. Hence, Data Civilizer can match all the tables in the old path to the new schema. Unless wholesale changes are made, several of the tables can be matched, and the entry point and target identified. If necessary, the DBA is asked to confirm automatic entry point and target identification. Then Data Civilizer calculates which of the following cases applies:

a) **The original path is unchanged.** Hence, there is no change to the SQL, and the transaction can be executed unchanged. Note that the path must be unchanged, as well as the application template.

b) **The original path has been removed but there is a unique replacement path.** In this case, data stitching produces only one path between the end points of the transaction. Moreover, the replacement path has the same application template as the original path. In this case, the DBA should be asked if the replacement path is equivalent to the previous one. If so, the query can be rewritten automatically and executed.

c) **There is a unique replacement path but the application template has changed.** In other words the replacement path does not have the same "shape" as the original. In this case, the semantics of the SQL will change, and the DBA must decide whether the changed semantics are correct or whether he needs to engage in defensive programming or rewrite the SQL to have a different effect.

d) **There are multiple replacement paths.** In this case, a human will be asked which one he wants to use, and this case devolves to one of the previous cases.

e) **There is no replacement path.** A human must decide what to do.

In this way we perform "just in time (JIT)" message resolution and processing. As a result, we have moved to an architecture which we call "schema at run time". The architecture in this section is a run-time schema in that the there is no commitment to a schema until the application is executed. In fact, the schema can change between executions of an application. This gives the ultimate in flexibility

However, in realistic scenarios, a DBA would like to know the impact of his schema changes before making them. This requires a second mode of operation, which we call "impact analysis", which is discussed in Section 5.5

5.5. Impact Analysis Operation

In impact analysis mode, the new schema is defined to Data Civilizer, but is not implemented. In this case, we have the old schema, the new schema and the SQL for most (or all) of the messages. In this case, Data Civilizer does not have the data to find PK-FK relationships, and these must be specified by a human. All messages are tested according to the algorithms in Section 5.4 to generate an impact analysis. If too much maintenance needed, then the DBA can suggest a different change or suggest that the change is too disruptive to be implemented.

5.6. Resolution of the Common Cases

If a relationship changes from 1−N to M−N, then we get case c) above. A human can decide whether the automatically generated SQL is correct or must be modified. Case c) also covers the transition from an M−N relationship to a 1−N relationship. However, there are transactions that can be automatically mapped. For example, a query will be unaffected when we move from M- N to 1−N. As a result, a human need only look at a subset of the transactions in this situation.

Dead attributes are covered by case a). In order to work correctly, we need all applications to be minimalist, i.e. to request only those fields in records that they actually use.

As a result, the thorny cases of changed semantics (case c) can be quickly identified and a human can decide what to do. This can be done incrementally at run time as messages are executed. Application groups who write the business logic and user interface code can be isolated from this process.

6. Related Work

There has been a great deal of work over the past two decades on database evolution. For example, work on schema matching (e.g., [7]) and migrating a database from old to new schema (e.g., Clio [13]) can be leveraged by Data Civilizer in database migration. However, unlike those solutions that support the development of "good" schemas, Data Civilizer assumes the new schema will obey the dominant enterprise objective of minimizing application maintenance. Hence, it may or may not be a "good" one.

There has also been extensive research on automating application rewriting due to database evolution, e.g., [11]. This work focuses on semantic-preserving schema changes. As we noted earlier, DBAs will generally not make such changes because of application risk. Hence, our work focuses on changes that do not necessarily preserve application semantics.

Our research is also focused on "in the wild" management and design practices for large enterprise databases. This is in contrast to [11] for example, which deals with a public-facing database. Although we recognize that it is much harder to obtain (often proprietary) experiences with enterprise data, we claim that it qualitatively changes the nature of the database evolution problem. Specifically, it alerted us to the importance of minimizing application maintenance which led us to a new development paradigm.

Although there are many tools based on more formal approaches, e.g., [1], [8], [11], [13] we encountered none of these in use at the enterprises we surveyed. Also, we chose to focus on a SQL-oriented approach to application design, because it easily exposes the SQL in use and allows us to more readily construct or validate replacement SQL when necessary. This is in contrast to approaches based on ORMs, web services, or higher level notations, e.g., LinQ [14], Ruby-on-Rails [15] or Hibernate [16].

7. Next Steps and Conclusions

In this paper we have indicated why database decay occurs, which ultimately results in a schema so tangled that further modifications become extremely difficult. Of course, defensive applications and schemas can help with maintenance issues, but do not impact the fundamental reasons for decay. It is plausible that moving to the application development methodology in Section 5 can slow down decay. We are currently exploring a collaboration with multiple "in the wild" application environments. These range from one with a few hundred tables to one with 1,800 tables. For now, Section 5 should be considered a proposal to be validated by future studies in a follow-on paper.

Acknowledgments

We gratefully acknowledge the assistance of the large enterprise database and application managers and DBAs who provided data for this research.

References

[1] P. P. Chen, "The entity-relationship model - toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.

[2] I. Jacobson, G. Booch, and J. E. Rumbaugh, *The unified software development process - the complete guide to the unified process from the original designers*, ser. Addison-Wesley object technology series. Addison-Wesley, 1999.

[3] P. A. Bernstein and S. Melnik, "Model management 2.0: manipulating richer mappings," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, 2007, pp. 1–12.

[4] C. Curino, H. J. Moon, M. Ham, and C. Zaniolo, "The PRISM workwench: Database schema evolution without tears," in *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, 2009, pp. 1523–1526.

[5] X. Li, "A survey of schema evolution in object-oriented databases," in *TOOLS 1999: 31st International Conference on Technology of Object-Oriented Languages and Systems, 22-25 September 1999, Nanjing, China*, 1999, pp. 362–371.

[6] S. Melnik, *Generic Model Management: Concepts and Algorithms*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 2967.

[7] P. A. Bernstein, J. Madhavan, and E. Rahm, "Generic schema matching, ten years later," *PVLDB*, vol. 4, no. 11, pp. 695–701, 2011.

[8] http://www.erwin.com/products/data-modeler.

[9] http://www.oracle.com/technetwork/developer-tools/datamodeler/overview/index.html.

[10] https://en.wikipedia.org/wiki/Database_Workbench.

[11] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *VLDB J.*, vol. 22, no. 1, pp. 73–98, 2013.

[12] 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, Seventh Biennial Conference on Innovative Data Systems Research*, 2017.

[13] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis, "Clio: Schema mapping creation and data exchange," in *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, 2009, pp. 198–236.

[14] http://www.tutorialspoint.com/linq/.

[15] http://www.rubyonrails.org/.

[16] http://www.hibernate.org/orm/.