Do we understand the wiring of our systems?

Position paper, submitted to the Tenth (2003) High Performance Transaction Systems Workshop

Sergey Melnik[1], Philip A. Bernstein
Microsoft Research, Redmond

Abstract

Today's large-scale transaction systems are complex integrations of many mission-critical applications and databases, tied to external systems via messages and protocols. Thus, there are many schemas, views, method signatures, mappings, etc. that need to be wired together seamlessly. The correctness of this wiring is paramount. However, do we always grasp the whole picture? Can we formally characterize the effects of optimizing the existing wiring or introducing a new system into it? We claim that we do not have a good way of addressing these questions. We illustrate our point using examples of schema evolution and migration of legacy data. We highlight the challenges and suggest a way of approaching the problem.

1. Motivation

Thousands of transactions that are executed every day run over integrated databases and applications. These systems are tied together using database views, mappings, mediators, and transformation scripts of various kinds. The data managed by transaction processing (TP) systems is typically mission-critical. This data originates from different sources, travels over multiple hops and changes from one representation into another. At the same time, new component databases get integrated into the system, legacy applications get migrated, and the existing architecture gets re-plumbed and optimized. And during all those "earthquakes", the TP system has to function reliably and correctly.

The more complex and indispensable our systems grow, the more surprising is the fact that we do not have a good way of formally characterizing the properties of the deployed databases and transformations between them. Most meta-data management tools and APIs utilized for maintenance of complex systems' plumbing are implemented in an ad-hoc fashion and provide little or no guarantees to the engineers who use them.

The solution we suggest for these kinds of tasks is model management. The work in [1] defines several high-level algebraic operators, called model-management operators, such as Merge, Diff, and Compose, which can be used for solving schema evolution, data integration, and other scenarios using short scripts. In [4], we describe the first executable prototype for model management and prove "by construction" that the operators are implementable and useful. However, we do not have a precise semantics for specifying the result of such scripts. Without it, the programming platform we are offering is not satisfactory.

Lack of formal semantics is not a unique problem of model management, but is symptomatic of our inability to precisely specify solutions to these kinds of problems. In fact, we claim that we do not understand properly the effects of even relatively simple transformations and schema updates. To illustrate our point, we consider two well-known scenarios, schema evolution and legacy migration. We demonstrate how little we can say about both scenarios formally using existing techniques. We then summarize the challenges and sketch a possible solution path.

---

[1] Sergey Melnik is currently finishing his Ph.D. at the University of Leipzig, Germany.

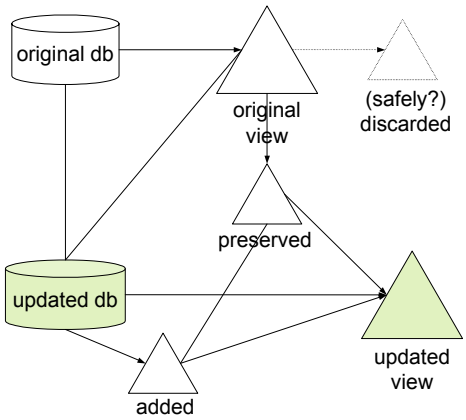

Figure 1: Schema evolution scenario

2. Schema evolution

The schema evolution problem arises when a change to a database schema breaks a view that is defined on it (see Figure 1). Our goal is to obtain an updated view through which we can query the updated database.

A typical state-of-the-art solution amounts to trial and error. The engineer creates an updated view, manually or semi-automatically, runs a series of canned tests against the updated view, and, once they all succeed, declares the task solved. Specifying the behavior of a mission-critical application by a test series is however problematic.

A better solution would be to provide a specification precise enough that an engineer could implement it unambiguously and tools could be built to analyze it, for example to identify possible undesired side effects. Ideally, the specification would even be executable – this would greatly simplify the solution. And in the best possible world, it would be declarative and amenable to automated rewriting and optimization.

The question is though, what properties would we want the updated view to have and how could we state them precisely? For example, should it be possible to query through the updated view all information that we were able to query through the original view? In general, the answer is no, since the updated database schema may be less expressive than the original one. How else can we then characterize the desired semantics of schema evolution?

On the one hand, the specification needs to say something about the amount of information we can see through the updated view. On the other hand, it needs to consider the structural properties of the view: some applications may depend on the fact that one of the view tables is called Account and its first field is called AccountID. That is, we have to consider both the "state-based" and the "structural" semantics to characterize the desired solution accurately.

While substantial competence has been gained by tool vendors with respect to structural transformations of meta-data artifacts, expressing the state-based semantics has proven to be hard. Using plain English, the state-based specification could be formulated roughly as follows: the updated view must expose all extra information that has been added to the updated schema while preserving the capability of answering queries that impact only the unchanged portion of the original schema. The challenge is to express this semantics formally in a declarative fashion, and make it operational.

Once we succeed in defining the above specification precisely, we can claim that this is the desired semantics of schema evolution. A formal approach would tread a path to studying other important properties of the schema evolution scenario, such as verifying whether applications

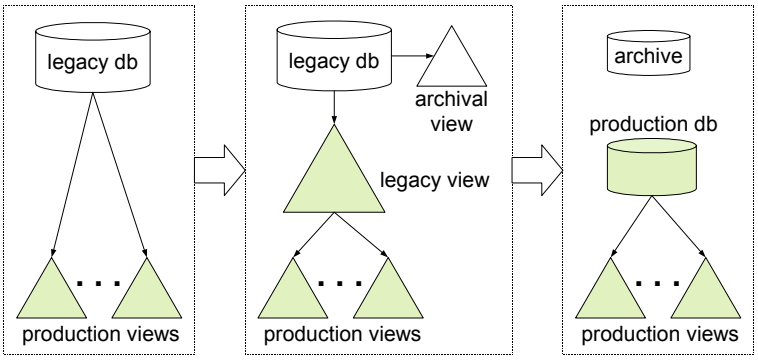

Figure 2: Legacy migration scenario

built against the original schema would still continue to work once all data has been migrated to the updated schema, whether the reverse data transformation required for that is lossless, etc.

In the next section, we briefly illustrate another important scenario, which is substantially more complex than the schema evolution scenario we sketched, and which suggests that the path to understanding our systems fully is going to be a long one.

3. Migration of legacy data

A critical issue that TP systems have to deal with is data migration from legacy databases. Migrating applications, data, and meta-data is an extremely complex task, which comprises many smaller individual sub-problems such as schema evolution, data integration, etc. We provide only a high-level picture to illustrate the challenges.

Consider the scenario of Figure 2. Assume that we run several TP applications against an existing legacy database (on the left). Each application accesses the database via a view or a wrapper. Now, the underlying DBMS platform or hardware becomes obsolete. Migration to a new production database (on the right) offers an opportunity to clean-up and revamp the database schema: the legacy features of the schema are removed and the existing data that is of historical interest only is physically relocated into an archive.

A possible methodology for data migration consists of two steps. First, we study the views deployed in production systems and identify a portion of the legacy database schema that is relevant for TP. We create a view over the legacy database. Through this view each of the applications continues to access the legacy database (middle part of Figure 2). At the same time, a portion of the legacy schema is extracted, which will keep the historical data. Finally, the legacy view created in the first step is cleaned-up and restructured into a new database schema to provide a better support for the features of the new production DBMS.

If the legacy database maintains mission-critical data, it would be highly desirable to precisely characterize the properties of the migration scenario upfront to avoid surprises along the way. A number of important questions need to be answered before we unscrew the first bolt: after migration, can we still answer each query and do each update through the production views? Can we rewrite all views and wrappers to access the production database directly, without the intermediate view? If we ever needed to provide an integrated view over the archival and production data, could we still do so? To achieve that, what properties must our migration scripts satisfy? Is our migration strategy optimal? Can we reformulate it in an equivalent way under the given business constraints?

A declarative specification with precisely defined state-based semantics could go a long way in helping us answer these questions. And, should it be executable, the implementation effort could be reduced dramatically.

4. Challenges

In this section we summarize the key challenges and requirements for formally describing the behavior of meta-data intensive applications. We build on the discussion in [2] and stress the limitations of the techniques deployed to date:

1. One approach is to define a special data model and constraint and transformation language. This is usually not useful for "le data model du jour," because it is hard to adapt to the languages used in practice, such as SQL views, XML Schema, XSLT, etc., since the adaptation may be lossy. We believe a better approach is to abstract out the specific features of languages and express the state-based semantics of tasks and scenarios in a generic (i.e. data model independent) fashion.
2. Existing techniques are too fine or too coarse. Many approaches introduce a number of fine-granularity restructuring primitives that are combined to describe more complex transformations. The expressive power obtained by combining such primitives is often inherently limited. Besides, formal specification of even relatively simple scenarios, such as database integration or schema evolution, becomes unwieldy due to the large number of features in practical data models that need to be taken into account.

   In contrast to that are state-based approaches based on the information capacity of a database schema, which is characterized by the set of possible database states conforming to the schema [3]. These approaches raise the level of abstraction much higher, so that it becomes possible to talk about relationships between databases without assuming specific schema and transformation languages. A major limitation of these approaches has been the focus on two kinds of relationships between schemas: equivalence and dominance. Either two schemas are equally expressive, i.e., their sets of instances are isomorphic, or one schema describes more instances than the other. These two kinds of relationships are insufficient [5]. For example, in our schema evolution scenario of Figure 1, the information capacity of all schemas and views may be incommensurate.
3. We need to be able to characterize operations with multiple inputs and inter-database constraints. For example, in database integration we need to consider the correspondences between the source databases, identify the overlapping portions, and resolve modeling conflicts.
4. Many databases possess implicit constraints that are not included in their schema definitions; we need a way of dealing with them. These constraints may be forgotten, not anticipated at design time, or simply not expressible in the schema language at hand. However, queries, view definitions, and transformations on these databases often make these constraints explicit (for example, an SQL view definition can be considered as a relational schema enriched with very complex constraints). We can and must take this information into account.
5. Ideally, the formal specification needs to be executable and optimizable (declarative). To make it executable, we need to tie syntax and semantics and consider both the structural and state-based semantics of applications.

5. Our approach

We advocate an algebraic approach to characterizing the behavior and the desired semantics of applications in complex scenarios. The focus of our previous work is on the structural semantics of the operators [1,4]. The state-based semantics is considered in [4] to a very limited extent using the notions of schema dominance, which by itself is too coarse, as we explained above.

Currently, we are exploring how the state-based semantics of the model-management operators can be defined precisely. In model management, the database schemas and other meta-data artifacts are called models. Views and database transformations are called mappings. To give formal definitions for the operators, such as Extract, Diff, Merge, Compose, and Intersect, we

interpret mappings as binary relations between the instances of models, i.e., between whole database states. Thus, a mapping can describe any conceivable database transformation.

To illustrate our approach, consider the operator Extract. It takes a model *m* and a mapping *map* and returns a submodel m_x of *m* that participates in the mapping. To explain the intuition behind the state-based semantics of Extract, imagine that *map* is a query over *m*. Thus, Extract treats *map* as a create-view statement and produces a minimal view schema m_x that can hold the result of the query. Notice that *map* may collapse several instances, or database states, of *m* into a single instance of m_x. That is, the extracted model m_x is in general less expressive than the source model *m*. However, we know more than the fact that *m* dominates m_x: by extraction, we obtain a model m_x that contains exactly those instances whose *m*-images are "distinguishable" under *map*.

Due to space limitations, we are not able to provide the formal definitions of the operators or describe their properties. As a teaser, we show below a formal specification of the schema evolution scenario of Figure 1 (s1 and s2 are original and updated database schemas, d1 and d2 are original and updated views, symbols with underscores such as s2_d2 identify mappings):

⟨s2′, s2′_s2⟩ = Diff(s2, s1_s2);
s2_d1 = Invert(s1_s2) ∗ s1_d1;
⟨dx, dx_d1⟩ = Extract(d1, s1_d1);
s2′_dx = s2′_s2 ∗ s2_d1 ∗ Invert(dx_d1);
⟨d2, s2′_d2, dx_d2⟩ = Merge(s2′, dx, s2′_dx);
s2_d2 = (s2_d1 ∗ Invert(dx_d1) ∗ dx_d2) ∪ (Invert(s2′_s2) ∗ s2′_d2);

Knowing the formal properties of the individual operators allows us to express the state-based semantics of many other important tasks. Our approach offers the following advantages:

- We can provide developers with a precise specification of the desired semantics.
- We can simplify application programming by implementing the individual operators. In fact, we already managed to make the scripts, such as the one above, executable for relational and XML schemas assuming a very simple transformation language [4].
- We can rewrite scripts into more efficient ones, or rewrite infeasible scripts into feasible ones using operator commutativity. For example, we can convert a schema into another schema language and extract a portion of it, or first extract a portion of a schema and then convert it.

6. Conclusion

We claim that we are lacking – and are in strong need of – a generic way of characterizing the properties of integrated databases and studying the effects of architectural changes in complex TP systems. We illustrated our point on two scenarios, and reviewed the challenging issues. We provided a brief sketch of a state-based, algebraic approach that we are pursuing.

References

1. P. A. Bernstein: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003
2. S. Davidson, P. Buneman, A. Kosky: Semantics of Database Transformations. In B. Thalheim, L. Libkin, Eds., Semantics in Databases, LNCS 1358, pp. 55–91, 1998
3. R. Hull: Relative Information Capacity of Simple Relational Database Schemata. SIAM J. Computing, 15(3), pp. 856-886, Aug 1986
4. S. Melnik, E. Rahm, P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. ACM SIGMOD 2003
5. R. J. Miller, Y. E. Ioannidis, R. Ramakrishnan: Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. Information Systems 19(1), pp. 3–31, 1994