Automated Co-evolution of Conceptual Models, Physical Databases, and Mappings

James F. Terwilliger, Philip A. Bernstein, and Adi Unnithan

Microsoft Corporation
{james.terwilliger,phil.bernstein,adi.unnithan}@microsoft.com

Abstract. Schema evolution is an unavoidable consequence of the application development lifecycle. The two primary schemas in an application, the conceptual model and the persistent database model, must co-evolve or risk quality, stability, and maintainability issues. We study application-driven scenarios, where the conceptual model changes and the database and mapping must evolve in kind. We present a technique that, in most cases, allows those evolutions to progress automatically. We treat the mapping as data, and mine that data for patterns. Then, given an incremental change to the conceptual model, we can derive the proper store and mapping changes without user intervention. We characterize the significant subset of mappings for which automatic evolution is possible, and present our techniques for evolution propagation.

1 Introduction

Object-relational mapping tools (ORMs) have become a central fixture in application programming over relational databases. They provide to the application developer the ability to develop against a conceptual model, generally an entity-relationship model with inheritance. The conceptual model is coupled to a mapping that describes the relationship between the model and a physical database schema. The ORM uses this mapping to translate queries and updates against the model into semantically-equivalent ones against the relational database.

As application requirements change, the application's conceptual model may need to evolve. Consider an application with a model as shown in Figure 1. A new application version may require that new types be added to the type hierarchy, including the three shown in the figure. With ORM tools, these conceptual model artifacts must map to persistent storage. Thus, we must determine what the mapping should be and what changes should be made to the physical schema.

If the entire hierarchy in Figure 1 is mapped to storage using a consistent pattern — for instance, by mapping the entire hierarchy to a single table (as Ruby on Rails does [8]) or instead by mapping each type to its own table — then it is natural to map the new type using that same pattern (regardless of where in the hierarchy of Figure 1 one chooses to add the type). However, for more complex mappings, especially ones that do not employ a uniform mapping pattern, the answer is more complicated. In Figure 1, the choice of mapping and physical storage may differ for each of the three locations for adding a new type.

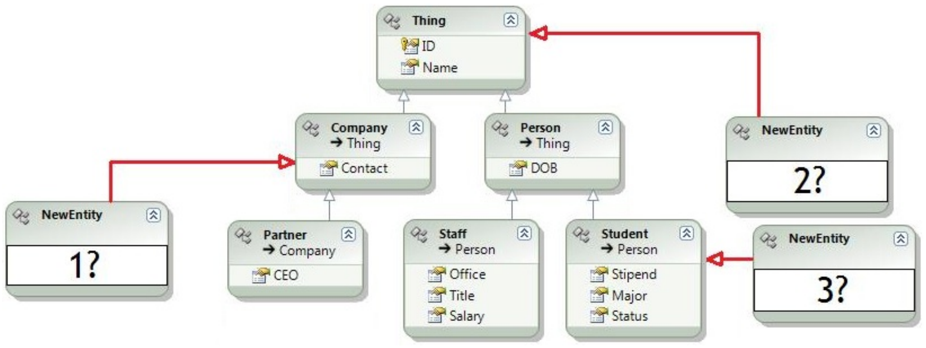

Fig. 1. A type hierarchy with three locations for adding a new type to the hierarchy

We assert that in many cases, the correct means for propagating a schema change from a conceptual model to a physical database is to follow the mapping. The contribution of this paper is a method to use the existing mapping to guide future incremental changes, even when the mapping scheme is not uniform across a hierarchy. If there is a consistent pattern in the immediate vicinity of the change, then that pattern is preserved after the change. As a special case, if an entire hierarchy is mapped using a single scheme, then it is mapped using that scheme for new artifacts. Given a list of incremental conceptual model changes and the previous version of the model and mapping:

1. Create a representation of the mapping called a *mapping relation* that lends itself to analysis, then
2. For each model change, effect changes to the mapping, to the store model, and to any physical databases that conform to the store model, and finally
3. Translate the mapping relation changes into changes to the original mapping

There are many ORM tools available today, including TopLink [7], Hibernate [4], Ruby on Rails [8], and Entity Framework (EF) [2]. Each of these tools has different syntax and expressive power for mapping conceptual models to databases, but none provides a method for propagating conceptual model changes into mapping or store changes. In the research project MeDEA, given an incremental change to a conceptual model, a developer chooses a rule to guide mapping evolution [3]. However, the new mapping need not be consistent with the existing mapping; if such consistency is important, the developer must maintain intricate knowledge of the mapping. By contrast, our technique provides the developer with an automated model design experience with changed artifacts mapped consistently with existing artifacts, which the DBA can optimize further if desired.

Section 2 extends the example in Figure 1 and introduces the concept of a mapping relation. Section 3 defines a schema change's "scope," whose mapping guides the processing of the change. Section 4 discusses the expressive power of O-R mappings and how to use the mapping relation to recognize patterns.

Section 5 describes how to use a mapping relation to propagate schema changes from a conceptual model to all other ORM artifacts. Finally, Section 6 analyzes related work, and Section 7 concludes the paper with some future directions.

2 Object-Relational Mappings and the Mapping Relation

Given the conceptual model from Figure 1, consider a mapping that combines several schemes for the hierarchy, as in Figure 2. The mapping between the conceptual model (a) and its physical storage (b) has the following characteristics:

- The types `Thing`, `Company`, and `Person` are mapped using the *Table-per-Type* (TPT) scheme, where each type maps to its own table and hierarchical relationships are modeled using foreign keys.
- The type `Partner` is mapped using the *Table-per-Concrete Class* (TPC) scheme relative to type `Company`, where each type still maps to its own table, but the child type `Partner` maps all of its properties derived from `Company`.
- The types `Person`, `Student`, and `Staff` are mapped using the *Table-per-Hierarchy* (TPH) scheme, with the entire sub-hierarchy mapped to a single table. Furthermore, the types map columns according to their domain, minimizing the number of columns needed in table `TPerson`.

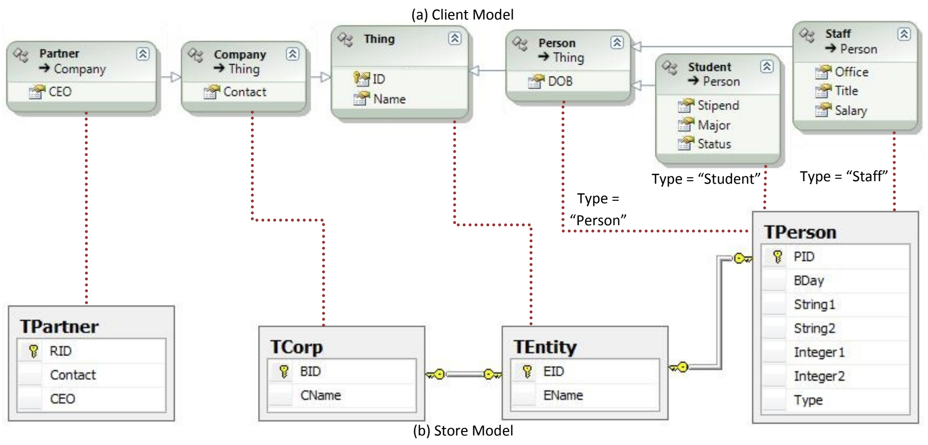

Fig. 2. Example client (a) and store (b) models, with a mapping between them

For this mapping, there is no single mapping scheme for the entire hierarchy. However, one can make some observations about the three different locations from Figure 1, specifically regarding the types that are "nearby":

- Location 1 has sibling `Partner` and parent `Company`, mapped TPC.
- Location 2 has siblings `Company` and `Person` and parent `Thing`, mapped TPT.
- Location 3 has parent `Student`, in a sub-hierarchy of types mapped TPH.

Using this informal reasoning, one can argue that types added at locations 1, 2, and 3 should be mapped using TPC, TPT, and TPH respectively. Formally speaking, we have two challenges to overcome. First, we need a definition of what it means to be "nearby" in a hierarchy. We address this in Section 3. Second, while TPC, TPT, and TPH are well-known and well-understood concepts to developers, they may not be expressed directly in the mapping language, so some analysis is needed to recognize them in a mapping. Moreover, finer-grained mapping notions like column mapping are not present in any available ORM.

Different object-relational mapping tools have different methods of expressing mappings. We assume in our work and our running example that the mappings are specified using EF, whose mappings need to be analyzed to identify mapping schemes like TPT, TPC, or TPH. In EF, a mapping is a collection of mapping fragments, each of which is an equation between select-project queries. Each fragment takes the form $\pi_{\boldsymbol{P}}\sigma_{\theta}E = \pi_{\boldsymbol{C}}\sigma_{\theta'}T$, where $\boldsymbol{P}$ is a set of properties of client-side entity E, $\boldsymbol{C}$ is a set of columns of table T, and θ and θ' are conditions over E and T respectively. Conditions θ and θ' may be of the form $c = v$ for column or property c and value v, c IS NULL, c IS NOT NULL, type tests IS T or IS ONLY T for type T, or conjunctions of such conditions.[1]

Our mapping evolution work uses a representation of an O-R mapping called a *mapping relation*, a relation $\mathcal{M}$ with the following eight attributes:

- $\mathcal{CE}, \mathcal{CP}, \mathcal{CX}$: Client entity type, property, conditions
- $\mathcal{ST}, \mathcal{SC}, \mathcal{SX}$: Store table, column, conditions
- $\mathcal{K}$: a flag indicating if the property is part of the key
- $\mathcal{D}$: The domain of the property

A mapping relation is a pivoted form of mapping, where each row represents a property-to-property mapping for a given set of conditions. As an example, consider the model pair and mapping shown in Figure 2. One can express the mapping in the figure using Entity Framework as follows:

- $\pi_{\textit{ID, Name}}\texttt{Thing} = \pi_{\textit{EID, EName}}\texttt{TEntity}$
- $\pi_{\textit{ID, Contact}}\sigma_{\text{IS ONLY Company}}\texttt{Thing} = \pi_{\textit{BID, CName}}\texttt{TCorp}$
- $\pi_{\textit{ID, Contact, CEO}}\sigma_{\text{IS Partner}}\texttt{Thing} = \pi_{\textit{RID, Contact, CEO}}\texttt{TPartner}$
- $\pi_{\textit{ID, DOB}}\sigma_{\text{IS ONLY Person}}\texttt{Thing} = \pi_{\textit{PID, Bday}}\sigma_{\textit{Type = "Person"}}\texttt{TPerson}$
- $\pi_{\textit{ID, DOB, Stipend, Major, Status}}\sigma_{\text{IS ONLY Student}}\texttt{Thing}$
  $= \pi_{\textit{PID, BDay, Integer1, String1, Integer2}}\sigma_{\textit{Type = "Student"}}\texttt{TPerson}$
- $\pi_{\textit{ID, DOB, Office, Title, Salary}}\sigma_{\text{IS ONLY Staff}}\texttt{Thing}$
  $= \pi_{\textit{PID, BDay, String1, String2, Integer1}}\sigma_{\textit{Type = "Staff"}}\texttt{TPerson}$

One can translate an EF mapping fragment $\pi_{\boldsymbol{P}}\sigma_{\boldsymbol{F}}E = \pi_{\boldsymbol{C}}\sigma_{\boldsymbol{G}}T$ into rows in the mapping relation as follows: for each property $p \in \boldsymbol{P}$, create the row $(E', p, \boldsymbol{F'}, T, c, \boldsymbol{G}, k, d)$, where:

[1] The formal specification in [6] allows disjunction in conditions. Here, we allow only conjunction because it simplifies exposition and because EF as implemented only allows conjunction. The techniques described in this paper are applicable when disjunction is allowed, with some adaptation.

- E' is the entity type that participates in the `IS` or `IS ONLY` condition of $\boldsymbol{F}$, or E if no such conditions exist
- $\boldsymbol{F'}$ is the set of conditions $\boldsymbol{F}$ with any `IS` or `IS ONLY` condition removed
- c is the column that matches p in the order of projected columns
- k is the boolean indicating whether the property is a key property
- d is a string value indicating the domain (i.e., data type) of the property

To translate an entire EF mapping to a mapping relation instance, one performs the above translation to each constituent mapping fragment. Table 1 shows the mapping relation for the models and mapping in Figure 2.

Table 1. The mapping relation for the models and mapping in Figure 2 (column $\mathcal{CC}$ not shown, since the mapping has no client conditions)

CE	CP	ST	SC	SX	K	D
Thing	ID	TEntity	EID	—	Yes	Guid
Thing	Name	TEntity	EName	—	No	Text
Company	ID	TCorp	BID	—	Yes	Guid
Company	Contact	TCorp	CName	—	No	Text
Partner	ID	TPartner	RID	—	Yes	Guid
Partner	Contact	TPartner	Contact	—	No	Text
Partner	CEO	TPartner	CEO	—	No	Text
Person	ID	TPerson	PID	Type=Person	Yes	Guid
Person	DOB	TPerson	BDay	Type=Person	No	Date
Student	ID	TPerson	PID	Type=Student	Yes	Guid
Student	DOB	TPerson	BDay	Type=Student	No	Date
Student	Stipend	TPerson	Integer1	Type=Student	No	Integer
Student	Major	TPerson	String1	Type=Student	No	Text
Student	Status	TPerson	Integer2	Type=Student	No	Integer
Staff	ID	TPerson	PID	Type=Staff	Yes	Guid
Staff	DOB	TPerson	BDay	Type=Staff	No	Date
Staff	Office	TPerson	String1	Type=Staff	No	Text
Staff	Title	TPerson	String2	Type=Staff	No	Text
Staff	Salary	TPerson	Integer1	Type=Staff	No	Integer

The rows in the mapping relation do not need to maintain `IS` or `IS ONLY` conditions because they are intrinsic in the mapping relation representation. The `IS` condition is satisfied by any instance of the specified type, while the `IS ONLY` condition is only satisfied by an instance of the type that is not also an instance of any derived type. In the mapping relation, the `IS` condition is represented by rows in the relation where non-key entity properties have exactly one represented row (e.g., `Thing.Name` in Table 1). The `IS ONLY` condition is represented by properties that are mapped both by the declared type and by its derived types (e.g., `Company.Contact` and `Partner.Contact` in Table 1).

3 Similarity and Local Scope

Our approach is to identify patterns that exist in the mapping in the local scope of the schema objects being added or changed. Before defining what local scope means, we first define what it means for two types in a hierarchy to be similar. Our notion of similarity formalizes the following notions:

– An entity type is most like its siblings.
– Two entity types X and Y, neither a descendant of the other, are more similar to each other than to their least common ancestor.
– If entity type X is a descendant of entity type Y, then X is more similar to any of Y's descendants than to Y, but more similar to Y than to any of Y's ancestors, siblings, or siblings' descendants.

We formalize these notions by assigning to each type in a hierarchy a pair of integers (m, n) relative to a given entity type E_0 that belongs to the hierarchy (or is just added to it) according to the following algorithm:

1. Assign the pair $(0, 0)$ to type E_0 and all of its siblings.
2. For each type E with assigned pair (m, n), if E's parent is unassigned, assign to it the pair $(m+2, n)$. Repeat until no new pair assignments can be made.
3. For each type E with assigned pair (m, n), assign the pair $(m + 1, n)$ to any of E's siblings that have no assigned pair. Apply this rule once for each type that has assigned pairs from step 2.
4. For each type E, if E has no pair and E's parent has the pair (m, n), assign to E the pair $(m, n+1)$. Repeat until no new pair assignments can be made.

Once the above steps have been completed, every type in the hierarchy will be assigned a pair. The *priority score* $\mathcal{P}(E, E_0)$ for an entity type E in a hierarchy relative to E_0 is computed from its pair (m, n) as $\mathcal{P}(E, E_0) = 1 + m - 2^{-n}$.

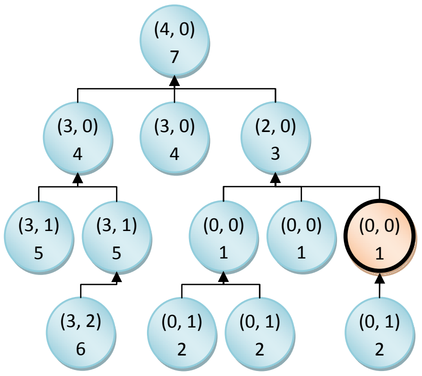

Fig. 3. An example of similarity pairs and the numeric order of hierarchy nodes relative to a given node (in bold outline)

Using the priority score, we can formalize the *local scope* $\Phi(E_0)$ of an entity type E_0 as follows. Let $\boldsymbol{H} = [E_1, E_2, \ldots]$ be the ordered list of entity types E_i in E_0's hierarchy such that $\sigma_{CE=E_i}\mathcal{M} \neq \emptyset$ (i.e., there exists mapping information; some types may be abstract and not have any mapping defined). List $\boldsymbol{H}$ is sorted on priority score, so $\mathcal{P}(E_i, E_0) \leq \mathcal{P}(E_{i+1}, E_0)$ for all indexes i. Then:

– If $|\boldsymbol{H}| \leq 2$, then $\Phi(E_0) = \boldsymbol{H}$.
– If $|\boldsymbol{H}| > 2$, then construct $\Phi(E_0)$ by taking the first two elements in $\boldsymbol{H}$, plus any elements with the same priority score as either of those elements.

This construction of local scope satisfies the informal notions we introduced earlier. For instance, if an entity type E has priority score x relative to E_0, then every sibling E_s of E also has priority score x unless E_s is an ancestor E_0. Thus, if $E \in \Phi(E_0)$, then any sibling E' that has associated mappings is also in $\Phi(E)$.

4 Mapping Patterns

Using the mapping relation and local scope function Φ, we can use the mapping itself as data to mine the various mapping schemes. A *mapping pattern* is a query Q^+ that probes for the existence of the requested mapping scheme and returns either `true` or `false`. The first set of patterns search for one of the three prominent hierarchy mapping schemes mentioned in Section 2, given a local scope $\Phi(E)$ for an entity type E:

Table-per-Hierarchy (TPH): Map an entity type E and its children to a single table T. Given local scope $\Phi(E)$, the TPH pattern tests whether the entities in E's scope map to exactly one store table. We test that all rows in the mapping relation matching the local scope have the same value for the mapped table ST:

$$Q^+_{TPH} \equiv (|\pi_{ST}\sigma_{CE \in \Phi(E)}\mathcal{M}| = 1).$$

Table-per-Type (TPT): Given an entity type E and a child type E', map them to tables T and T' respectively, with properties of E mapped to T and properties of E' not present in E mapped to T'. Given local scope $\Phi(E)$, we define the TPT pattern as a query that tests for two properties. First, any pair of entity types in scope will have non-overlapping sets of mapped tables. Second, if $\mathcal{A}$ is the least common ancestor of all entity types in $\Phi(E)$, then for each entity type in scope that is not the common ancestor, the non-key properties of $\mathcal{A}$ are not re-mapped (i.e., there are no matching rows in the mapping relation):

$$Q^+_{TPT} \equiv (\forall_{E',E'' \in \Phi(E)} \pi_{ST}\sigma_{CE=E'}\mathcal{M} \cap \pi_{ST}\sigma_{CE=E''}\mathcal{M} = \emptyset)$$
$$\wedge (\forall_{E' \in \Phi(E)} \forall_{P \in NKP(\mathcal{A})} (E' \neq \mathcal{A}) \rightarrow |\sigma_{CP=P}\sigma_{CE=E'}\mathcal{M}| = 0).$$

where $NKP(E)$ is the set of declared non-key properties for entity type E (i.e., do not include properties derived from ancestors).

Table-per-Concrete Class (TPC): Given an entity type E and a child type E', map them to tables T and T' respectively, with properties of E mapped to T and properties of E' (including properties inherited from E) mapped to T'. We define the TPC pattern as the same tests as TPT, except that all entity types in scope must map the non-key properties of common ancestor $\mathcal{A}$:

$$Q^+_{TPC} \equiv (\forall_{E',E'' \in \Phi(E)} \pi_{ST}\sigma_{CE=E'}\mathcal{M} \cap \pi_{ST}\sigma_{CE=E''}\mathcal{M} = \emptyset)$$
$$\wedge (\forall_{E' \in \Phi(E)} \forall_{P \in NKP(\mathcal{A})} |\sigma_{CP=P}\sigma_{CE=E'}\mathcal{M}| > 0).$$

If we find an instance of the TPH scheme, we can use column mapping patterns to distinguish further how the existing mapping reuses store columns. Column mapping patterns do not use local scope, but rather look at the entire mapping table for all entities that map to a given table; we expand the set of considered entities to all entities because the smaller scope is not likely to yield enough data to exhibit a pattern:

Remap by column name (RBC): If types E and E' are cousin types in a hierarchy[2], and both E and E' have a property named P with the same domain, then $E.P$ and $E'.P$ are mapped to the same store column. This scheme maps all properties with like names to the same column, and is the scheme that Ruby on Rails uses by convention [8]. Given hierarchy table T, the RBC pattern is:

$$Q^{+}_{RBC} \equiv (\exists_{C \in \pi_{SC}\sigma_{ST=T}\sigma_{\neg K}\mathcal{M}} |\sigma_{CP \in NKP(CE)}\sigma_{ST=T \wedge SC=C}\mathcal{M}| > 1)$$
$$\wedge (\forall_{C \in \pi_{SC}\sigma_{ST=T}\sigma_{\neg K}\mathcal{M}} |\pi_{CP}\sigma_{CP \in NKP(CE)}\sigma_{ST=T \wedge SC=C}\mathcal{M}| = 1).$$

That is, check if a store column C is mapped to more than one client property, and all client properties CP that map to store column C have the same name.

Remap by domain (RBD): If types E and E' are cousin types in a hierarchy, let $\boldsymbol{P}$ be the set of all properties of E with domain D (including derived properties), and $\boldsymbol{P'}$ be the set of all properties of E' with the same domain D. If $\boldsymbol{C}$ is the set of all columns to which any property in $\boldsymbol{P}$ or $\boldsymbol{P'}$ map, then $|\boldsymbol{C}| = \max(|\boldsymbol{P}|, |\boldsymbol{P'}|)$. In other words, the mapping maximally re-uses columns to reduce table size and increase table value density, even if properties with different names map to the same column. Said another way, if one were to add a new property P_0 to an entity type mapped using the TPH scheme, map it to any column C_0 such that C_0 has the same domain as P_0 and is not currently mapped by any property in any descendant type, if any such column exists. Given hierarchy table T, the RBD pattern is:

$$Q^{+}_{RBD} \equiv (\exists_{C \in \pi_{SC}\sigma_{ST=T}\sigma_{\neg K}\mathcal{M}} |\sigma_{CP \in NKP(CE)}\sigma_{ST=T \wedge SC=C}\mathcal{M}| > 1)$$
$$\wedge (\forall_{X \in \pi_{D}\sigma_{ST=T \wedge \neg K}\mathcal{M}} \exists_{E \in \pi_{CE}\sigma_{ST=T}\mathcal{M}} |\pi_{CP}\sigma_{CE=E \wedge ST=T \wedge D=X \wedge \neg K}\mathcal{M}|$$
$$= |\pi_{SC}\sigma_{ST=T \wedge D=X \wedge \neg K}\mathcal{M}|).$$

There is at least one store column C that is remapped, and for each domain D, there is some client entity E that uses all available columns of that domain.

Fully disjoint mapping (FDM): If types E and E' are cousin types in a hierarchy, the non-key properties of E map to a set of columns disjoint from the non-key properties of E'. This pattern minimizes ambiguity of column data provenance — given a column c, all of its non-null data values belong to instances of a single entity type. Given hierarchy table T, the FDM pattern is:

$$Q^{+}_{FDM} \equiv \forall_{C \in \pi_{SC}\sigma_{ST=T}\sigma_{\neg K}\mathcal{M}} |\sigma_{CP \in NKP(CE)}\sigma_{ST=T \wedge SC=C}\mathcal{M}| = 1.$$

Each store column C is uniquely associated with a declared entity property CP.

[2] *Cousin types* belong to the same hierarchy, but neither is a descendant of the other.

In addition to hierarchy and column mapping schemes, other transformations may exist between client types and store tables. For instance:

Horizontal partitioning (HP): Given an entity type E with a non-key property P, one can partition instances of E across tables based on values of P.

Store-side constants (SSC): One can assign a column to hold a particular constant. For instance, one can assign to column C a value v that indicates which rows were created through the ORM tool. Thus, queries that filter on $C = v$ eliminate any rows that come from an alternative source.

Strictly speaking, we do not need patterns for these final two schemes — our algorithm for generating new mapping relation rows (Section 5) carries such schemes forward automatically. Other similar schemes include vertical partitioning and merging, determining whether a TPH hierarchy uses a discriminator column (as opposed to patterns of `NULL` and `NOT NULL` conditions), and association inlining (i.e., whether one-to-one and one-to-many relationships are represented as foreign key columns on the tables themselves or in separate tables).

Note that each group of patterns is not complete on its own. The local scope of an entity may be too small to find a consistent pattern or may not yield a consistent pattern (e.g., one sibling is mapped TPH, while another is mapped TPC). In our experience, the developer is most likely to encounter this situation during bootstrapping, when the client model is first being built. Most mappings we see are totally homogeneous, with entire models following the same scheme. Nearly all the rest are consistent in their local scope (specifically, all siblings are mapped identically). However, for completeness in our implementation, we have chosen the following heuristics for the rare case when consistency is not present: If we do not see a consistent hierarchy mapping scheme (e.g., TPT), we rely on a global default given by the user (similar to [3]). If we do not see a consistent column mapping scheme, we default to the disjoint pattern. If we do not see consistent condition patterns like store constants or horizontal partitioning, we ignore any store and client conditions that are not relevant to TPH mapping.

5 Evolving a Mapping

Once we know that a pattern is present in the mapping, we can then effect an incremental change to the mapping and the store based on the nature of the change. The incremental changes that we support fall into four categories:

Actions that add constructs: One can add entity types to a hierarchy, add a new root entity type, add properties, or add associations. Setting an abstract entity type to be concrete is also a change of this kind. For these changes, new rows may be added to the mapping relation, but existing rows are left alone.

Actions that remove constructs: One can drop any of the above artifacts, or set a concrete entity type to be abstract. For changes of this kind, rows may be removed from the mapping relation, but no rows are changed or added.

Actions that alter construct attributes: One can change individual attributes, or “facets,’, of artifacts. Examples include changing the maximum length of a string property or the nullability of a property. For such changes, the mapping relation remains invariant, but is used to guide changes to the store.

Actions that refactor or move model artifacts: One can transform model artifacts in a way that maximizes information preservation, such as renaming a property (rather than dropping and re-adding it), transforming an association into an inheritance, or changing an association's cardinality. Changes of this kind may result in arbitrary mapping relation changes, but such changes are often similar to (and thus re-use logic from) changes of the other three kinds.

The set of possible changes is closed in that one can evolve any client model M_1 to any other client model M_2 by dropping any elements they do not have in common and adding the ones unique to M_2 (a similar closure argument has been made for object-oriented models, e.g. [1]). All of the rest of the supported changes — property movement, changing the default value for a property, etc. — can be accomplished by drop-add pairs, but are better supported by atomic actions that preserve data. For the rest of the section, we show the algorithms for processing a cross-section of the supported model changes.

Adding a new type to the hierarchy: When adding a new type to a hierarchy, one must answer three questions: what new tables must be created, what existing tables will be re-used, and which derived properties must be re-mapped. For clarity, we assume that declared properties of the new type will be added as separate "add property" actions. When a new entity type E is added, we run algorithm `AddNewEntity`:

```
1. AddNewEntity(E):
2.   $k \leftarrow$ a key column for the hierarchy
3.   $\boldsymbol{G} \leftarrow \gamma_{CX}\sigma_{CP=k \wedge CE \in \Phi(E)}\mathcal{M}$, where $\gamma_{CX}$ groups rows of the mapping relation according to their client conditions
4.   If $\exists_i |\pi_{CE}\boldsymbol{G}_i| \neq |\Phi(E)|$ then $\boldsymbol{G} \leftarrow \{\sigma_{CP=k \wedge CE \in \Phi(E)}\mathcal{M}\}$ (if there is no consistent horizontal partition across entity types, then just create one large partition, ignoring client-side conditions)
5.   For each $G \in \boldsymbol{G}$:
6.     If $Q^+_{TPT}(G)$: (if TPT pattern is found when run just on the rows in $G$)
7.       For each property $P \in$ Keys(E) $\cup$ NKP(E):
8.         Add NewMappingRow(GenerateTemplate($G, P$), $E$)
9.     If $Q^+_{TPH}(G)$ or $Q^+_{TPC}(G)$:
10.      $A \leftarrow$ the common ancestor of $\Phi(E)$
11.      For each property $P \in$ Keys(E) $\cup \cap_{e \in \boldsymbol{E}}$ NKP(E) where $\boldsymbol{E}$ is the set of all entities between $E$ and $A$ in the hierarchy, inclusive:
12.        Add NewMappingRow(GenerateTemplate($G, P$), $E$)
```

Function `GenerateTemplate`($\boldsymbol{R}, P$) is defined as follows: we create a mapping template T as a derivation from a set of existing rows $\boldsymbol{R}$, limited to those where $CP = P$. For each column $C \in \{CE, CP, ST, SC\}$, set $T.C$ to be X if $\forall_{r \in \boldsymbol{R}} r.C = X$. Thus, for instance, if there is a consistent pattern mapping all properties called `ID` to columns called `PID`, that pattern is continued. Otherwise, set $T.C = \otimes$, where $\otimes$ is a symbol indicating a value to be filled in later.

For condition column CX (and SX), template generation follows a slightly different path. For any condition $C = v$, C `IS NULL`, or C `IS NOT NULL` that

Table 2. Creating the mapping template for a type added using a TPH scheme, over a single horizontal partition where "Editor=Tom" and with a store-side constant "Source=A" — the final row shows the template filled in for a new type *Alumnus*

CE	**CP**	**CX**	**ST**	**SC**	**SX**	**K**	**D**
Person	ID	Editor=Tom	TPerson	PID	Type=Person AND Source=A	Yes	Guid
Student	ID	Editor=Tom	TPerson	PID	Type=Student AND Source=A	Yes	Guid
Staff	ID	Editor=Tom	TPerson	PID	Type=Staff AND Source=A	Yes	Guid
$\otimes$	**ID**	Editor=Tom	**TPerson**	**PID**	**Type=**$\otimes$ AND **Source=A**	Yes	Guid
Alumnus	**ID**	Editor=Tom	**TPerson**	**PID**	**Type=***Alumnus* AND **Source=A**	Yes	Guid

appear in *every* CX (or SX) field in $\boldsymbol{R}$ (treating a conjunction of conditions as a list that can be searched), and the value v is the same for each, add the condition to the template. If each row $r \in \boldsymbol{R}$ contains an equality condition $C = v$, but the value v is distinct for each row r, add condition $C = \otimes$ to the template. Ignore all other conditions.

Table 2 shows an example of generating a mapping template for a set of rows corresponding to a TPH relationship; the rows for the example are drawn from Table 1, with additional client and store conditions added to illustrate the effect of the algorithm acting on a single horizontal partition and a store constant. Note that the partition conditions and store conditions translate to the template; note also that the name of the store column remains consistent even though it is not named the same as the client property.

The function `NewMappingRow`(F, E) takes a template F and fills it in with details from E. Any $\otimes$ values in CE, CX, ST, and SX are filled with value E. Translating these new mapping table rows back to an EF mapping fragment is straightforward. For each horizontal partition, take all new rows collectively and run the algorithm from Section 2 backwards to form a single fragment.

Adding a new property to a type: When adding a new property to a type, one has a different pair of questions to answer: which descendant types must also remap the property, and to which tables must a property be added. The algorithm for adding property P to type E is similar to adding a new type:

- For each horizontal partition, determine the mapping scheme for $\Phi(E)$.
- If the local scope has a TPT or TPC scheme, add a new store column and a new row that maps to it. Also, for any child types whose local scope is mapped TPC, add a column and map to it as well.
- If the local scope has a TPH scheme, detect the column remap scheme. If remapping by name, see if there are other properties with the same name, and if so, map to the same column. If remapping by domain, see if there is an available column with the same domain and map to it. Otherwise, create a new property and map to it. Add a mapping row for all descendant types that are also mapped TPH.

Translating these new mapping rows backward to the existing EF mapping fragments is straightforward. Each new mapping row may be translated into a new item added to the projection list of a mapping fragment. For a new mapping row N, find the mapping fragment that maps $\sigma_{N.CX} N.CE = \sigma_{N.SX} N.ST$ and add $N.CP$ and $N.SC$ to the client and store projection lists respectively.

Changing or dropping a property: One can leverage the mapping relation to propagate schema changes and deletions through a mapping as well. Consider first a scenario where the user wants to increase the maximum length of `Student.Major` to be 50 characters from 20. We use the mapping relation to effect this change as follows. First, if $E.P$ is the property being changed, issue query $\pi_{ST,SC}\sigma_{CE=E \wedge CP=P}\mathcal{M}$ — finding all columns that property $E.P$ maps to (there may be more than one if there is horizontal partitioning). Then, for each result row t, issue query $Q = \pi_{CE,CP}\sigma_{ST=t.ST \wedge SC=t.SC}\mathcal{M}$ — finding all properties that map to the same column. Finally, for each query result, set the maximum length of the column $t.SC$ in table $t.SE$ to be the maximum length of all properties in the result of query Q.

For the `Student.Major` example, the property only maps to a single column `TPerson.String1`. All properties that map to `TPerson.String1` are shown in Table 3. If `Student.Major` changes to length 50, and `Staff.Office` has maximum length 40, then `TPerson.String1` must change to length 50 to accommodate. However, if `TPersonString1` has a length of 100, then it is already large enough to accommodate the wider `Major` property.

Dropping a property follows the same algorithm, except that the results of query Q are used differently. If query Q returns more than one row, that means multiple properties map to the same column, and dropping one property will not require the column to be dropped. However, if r is the row corresponding to the dropped property, then we issue a statement that sets $r.SC$ to `NULL` in table $r.ST$ for all rows that satisfy $r.SX$. So, dropping `Student.Major` will execute `UPDATE TPerson SET String1 = NULL WHERE Type='Student'`. If query Q returns only the row for the dropped property, then we delete the column.[3] In both cases, the row r is removed from $\mathcal{M}$. We refer to the process of removing the row r and either setting values to `NULL` or dropping a column as `DropMappingRow`(r).

Table 3. A listing of all properties that share the same mapping as `Student.Major`

CE	CP	ST	SC	SX	K	D
Student	Major	TPerson	String1	Type=Student	No	Text
Staff	Office	TPerson	String1	Type=Staff	No	Text

Moving a property from a type to a child type: If entity type E has a property P and a child type E', it is possible using a visual designer to specify that the property P should move to E'. In this case, all instances of E' should keep their values for property P, while any instance of E that is not an instance of E' should drop its P property. This action can be modeled using analysis of the mapping relation $\mathcal{M}$ as well. Assuming for brevity that there are no client-side conditions, the property movement algorithm is as follows:

[3] Whether to actually delete the data or drop the column from storage or just remove it from the storage model available to the ORM is a policy matter. Our current implementation issues `ALTER TABLE DROP COLUMN` statements.

```
1. MoveClientProperty($E, P, E'$):
2.   $r_0 \leftarrow \sigma_{CE=E \wedge CP=P}\mathcal{M}$ (without client conditions, this is a single row)
3.   If $|\sigma_{CE=E' \wedge CP=P}\mathcal{M}| = 0$: ($E'$ is mapped TPT relative to $E$)
4.       AddProperty($E', P$) (act as if we are adding property $P$ to $E'$)
5.       For each $r \in \sigma_{CE=E' \vee CE \in Descendants(E')}\sigma_{CP=P}\mathcal{M}$:
6.           UPDATE $r.ST$ SET $r.SC = (r.ST \bowtie r_0.ST).(r.SC)$ WHERE $r.SX$
7.   $\boldsymbol{E}^- \leftarrow$ all descendants of $E$, including $E$ but excluding $E'$ and descendants
8.   For each $r \in \sigma_{CE \in \boldsymbol{E}^- \wedge CP=P}\mathcal{M}$:
9.       DropMappingRow($r$) (drop the mapping row and effect changes to the
     physical database per the Drop Property logic in the previous case)
```

6 Related Work

A demonstration of our implementation is described in [10]. That work describes how to capture user intent to construct incremental changes to a conceptual model and how to generate SQL scripts. It gives intuition about how to select a mapping scheme given a change, and mentions the mapping relation. This paper extends that work by providing the algorithms, the formal underpinnings, a more complete example, and the exact conditions when the approach is applicable.

A wealth of research has been done on schema evolution [9], but very little on co-evolution of mapped schemas connected by a mapping. One example is MeDEA, which uses manual specifications of update policies [3]. One can specify policies for each class of incremental change to the client schema as to what the desired store effect should be on a per-mapping basis. The Both-As-View (BAV) federated database language can express non-trivial schema mappings, though it does not handle inheritance [5]. For some schema changes (to either schema in BAV), either the mapping or the other schema can be adjusted to maintain validity. Many cases require manual intervention for non-trivial mappings.

Two cases of schema evolution have been considered in data exchange, one on incremental client model changes [11], and one where evolution is represented as a mapping [12]. Both cases focus on "healing" the mapping between schemas, leaving the non-evolved schema invariant. New client constructs do not translate to new store constructs, but rather add quantifiers or Skolem functions to the mapping, which means new client constructs are not persisted. It is unclear whether a similar technique as ours can be applied to a data exchange setting. However, it would be an interesting exercise to see if it is possible to define both patterns and a mapping table representation for first-order predicate calculus, in which case similar techniques could be developed.

7 Conclusion and Future Work

We have presented a way to support model-driven application development by automatically translating incremental client model changes into changes to a store model, the mapping between the two, and any database instance conforming to the store model. Our technique relies on treating an O-R mapping as minable data, as well as a notion of pattern uniformity within a local scope.

A prominent feature of EF is that it compiles mapping fragments into views that describe how to translate data from a store model into a client model and vice versa. Mapping compilation provides several benefits, including precise mapping semantics and a method to validate that a mapping can round-trip client states. The computational cost for compiling and validating a mapping can become large for large models. An active area of our research is to translate incremental changes to a model into incremental changes to the relational algebra trees of the compiled query and update views, with results that are still valid and consistent with the corresponding mapping and store changes.

Finally, the mapping relation is a novel method of expressing an O-R mapping, and as such, it may have desirable properties that are yet unstudied. For instance, it may be possible to express constraints on a mapping relation instance that can validate a mapping's roundtripping properties.

References

1. Banerjee, J., Kim, W., Kim, H., Korth, H.F.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In: SIGMOD 1987 (1987)
2. Blakeley, J.A., Muralidhar, S., Nori, A.: The ADO .NET Entity Framework: Making the Conceptual Level Real. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, Springer, Heidelberg (2006)
3. Domíngueza, E., Lloret, J., Rubio, A.L., Zapata, M.A.: Evolving the Implementation of ISA Relationships in EER Schemas. In: Roddick, J., Benjamins, V.R., Si-said Cherfi, S., Chiang, R., Claramunt, C., Elmasri, R.A., Grandi, F., Han, H., Hepp, M., Lytras, M.D., Mišić, V.B., Poels, G., Song, I.-Y., Trujillo, J., Vangenot, C. (eds.) ER Workshops 2006. LNCS, vol. 4231, pp. 237–246. Springer, Heidelberg (2006)
4. Hibernate, `http://www.hibernate.org/`
5. McBrien, P., Poulovassilis, A.: Schema Evolution in Heterogeneous Database Architectures, a Schema Transformation Approach. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, p. 484. Springer, Heidelberg (2002)
6. Melnik, S., Adya, A., Bernstein, P.A.: Compiling Mappings to Bridge Applications and Databases. ACM TODS 33(4) (2008)
7. Oracle TopLink, `http://www.oracle.com/technology/products/ias/toplink/`
8. Ruby on Rails, `http://rubyonrails.org/`
9. Rahm, E., Bernstein, P.A.: An Online Bibliography on Schema Evolution. SIGMOD Record 35(4) (2006)
10. Terwilliger, J.F., Bernstein, P.A., Unnithan, A.: Worry-Free Database Upgrades: Automated Model-Driven Evolution of Schemas and Complex Mappings. In: SIGMOD 2010 (2010)
11. Velegrakis, Y., Miller, R.J., Popa, L.: Preserving Mapping Consistency Under Schema Changes. VLDB Journal 13(3) (2004)
12. Yu, C., Popa, L.: Semantic Adaptation of Schema Mappings When Schemas Evolve. In: VLDB 2005 (2005)