Incremental Mapping Compilation in an Object-to-Relational Mapping System*

Philip A. Bernstein
Microsoft Corporation
philbe@microsoft.com

Marie Jacob
Univ. of Pennsylvania
majacob@cis.upenn.edu

Jorge Pérez
DCC, Universidad de Chile
jperez@dcc.uchile.cl

Guillem Rull
Univ. Politècnica de Catalunya
grull@essi.upc.edu

James F. Terwilliger
Microsoft Corporation
jamest@microsoft.com

ABSTRACT

In an object-to-relational mapping system (ORM), mapping expressions explain how to expose relational data as objects and how to store objects in tables. If mappings are sufficiently expressive, then it is possible to define lossy mappings. If a user updates an object, stores it in the database based on a lossy mapping, and then retrieves the object from the database, the user might get a different result than the updated state of the object; that is, the mapping might not "roundtrip." To avoid this, the ORM should validate that user-defined mappings roundtrip the data. However, this problem is NP-hard, so mapping validation can be very slow for large or complex mappings.

We circumvent this problem by developing an incremental compiler for OR mappings. Given a validated mapping, a modification to the object schema is compiled into incremental modifications of the mapping. We define the problem formally, present algorithms to solve it for Microsoft's Entity Framework, and report on an implementation. For some mappings, incremental compilation is over 100 times faster than a full mapping compilation, in one case dropping from 8 hours to 50 seconds.

Categories and Subject Descriptors
H.0 [**Information Systems Applications**]: General; D.2.12 [**Software Engineering**]: Interoperability – *Data mapping*

Keywords
Object-to-relational mapping; incremental compilation

1. INTRODUCTION

An object-to-relational mapping (ORM) system enables developers to write object-oriented applications over object-oriented data stored in a relational database. An ORM supports two essential features: inheritance, which enables the database schema to match the structure of application classes; and updates over non-trivial mappings, which enables a flexible choice of relational schemas to store instances of classes. Generic ORMs in widespread use are Microsoft's ADO.NET Entity Framework [1], EclipseLink [18], Hibernate [19], Oracle's TopLink [20], and Ruby on Rails [21].

Using an ORM, a developer provides three definitions: an object-oriented schema that is the application program's view of the data; a relational schema that is the database system's view of the data; and a mapping between them. The ORM needs to compile the developer's schema and mapping definitions into an internal representation that supports the translation of queries and updates over the object-oriented view into queries and updates over the relations.

Applications often require hundreds of classes and tables—indeed, some require thousands. For such applications, mapping compilation can be slow, requiring tens of minutes or even hours. This compilation time is undesirable, but tolerable when compiling a large application for deployment. But during application development, as the mapping becomes large, long compilation time is a major impediment to programmer productivity. It is especially annoying when making a minor change to the object-oriented model, which has only a minor effect on the relational schema and mapping, yet still requires recompiling the entire mapping.

To solve this problem, we have developed an incremental compiler for object-to-relational mappings. The technology to do it turned out to be highly non-trivial, requiring a deep analysis of the effect of schema and mapping changes. The resulting performance gains were well worth the effort. Incremental compilation times were reduced by over 99%, in one case from 8 hours to 50 seconds.

1.1 Query and Update Mappings

A common way for an ORM to support query translation is to express the mapping as a view definition, where the object-oriented schema is a view over the relational schema. A query over the object-oriented schema can be implemented by view unfolding, which replaces view references in the query by the view definition.

Update translation is more challenging, because it requires a solution to the well-known hard problem of view updating [2]. An update U expressed on the object-oriented view of data must be translated into updates on the relational view that have exactly the effect of U and preserve database consistency. Since relational systems only allow updating of relatively trivial views, an ORM needs to provide its own logic for directing the translation of updates from objects to relations. This logic can be encapsulated in a fixed set of mapping types, such as templates in Hibernate or type annotations in Active Record definitions in Rails. It can be expressed procedurally, as custom mappings in an application suite or an INSTEAD-OF trigger in SQL; though flexible, this approach is brittle since

*This work was done while Marie Jacob, Jorge Pérez, and Guillem Rull were working at Microsoft Research.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

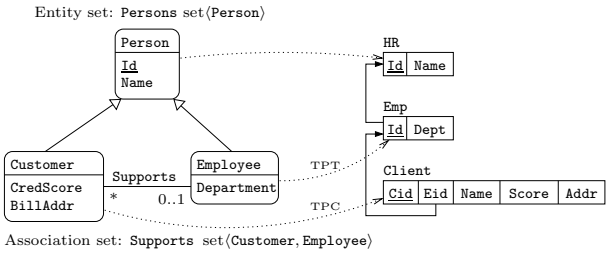

Figure 1: Client and store schema (from Fig. 6 in [13])

simple changes to the logical mapping can break the update logic. Or, it can be expressed as an update view that expresses the relational schema as a view over the object-oriented schema, as in Microsoft's Entity Framework.

Whatever language is chosen for expressing the mapping, an obvious correctness criterion for a mapping is that there is no data loss when translating updates on the object-oriented view into updates on the relational database. That is, updates should *roundtrip*, in the sense that after an update is propagated to the database, the object-oriented view of the database reflects the result of that update and no other changes.

In some ORMs, the mappings have sufficiently limited expressiveness that it is easy to see that updates are translated correctly. However, if more expressive mappings are permitted, then it can be quite challenging to determine whether all updates roundtrip. For example, the mapping in Figure 1 between three classes and three tables generates the 27-line query view in Figure 2, which has two nested subqueries, a case statement, a left outerjoin, and a union. (This example is taken from Figures 6 and 7 in [13] and will be discussed in detail in Section 3.) Even if the corresponding update views are trivial, manually checking that these views roundtrip is still quite difficult. Practical mappings involve dozens of classes and tables, which is surely out of reach for manual validation.

Nevertheless, most ORMs do not automatically check that mappings roundtrip. A developer has to take it on faith that they do. To the best of our knowledge, the only system that automates this task is Entity Framework. It explicitly checks that mappings roundtrip in a process called *mapping validation*.

In addition to addressing the roundtripping problem, mapping validation in Entity Framework is needed to facilitate program development: it supports very expressive query and update views, which are too complex for developers to code and maintain themselves (such as Figure 2). To avoid writing such views manually, the framework offers a higher-level declarative mapping language and a compiler that compiles declarative mappings into query and update views. A mapping consists of a set of *mapping fragments*, each of which is an equation $Q_{\mathcal{C}} = Q_{\mathcal{S}}$ where $Q_{\mathcal{C}}$ and $Q_{\mathcal{S}}$ are project-select queries over the client schema $\mathcal{C}$ and storage schema $\mathcal{S}$ respectively, and where the projected attributes in both queries include a key. Unfortunately, the rich nature of even this simple language can enable a developer to express mappings that are not valid. One solution would be to restrict the language to ensure that all expressible mappings roundtrip. However, we know of no syntactic restrictions that would accomplish this and still allow the expression of the desirable mappings that do roundtrip. Instead, the Entity Framework mapping compiler includes mapping validation to check whether the user-specified mappings roundtrip.

To compile and validate mappings, Entity Framework must generate query and update views from the mapping and then check roundtripping by checking that the composition of the views is the identity function. Given a query view Q and update view V compiled from the mapping, it needs to check that $V \circ Q \subseteq \mathcal{I}_{\mathcal{C}}$ and

```
People =
SELECT VALUE
  CASE
   WHEN (T5._from1 AND NOT T5._from2)
    THEN Person(T5.Id, T5.Name)
   WHEN T5._from2
    THEN Employee(T5.Id, T5.Name, T5.Dept)
    ELSE Customer(T5.Id, T5.Name, T5.BillAddr, T5.CredScore)
  END
FROM (
  (SELECT T1.Id, T1.Name, T2.Dept,
      CAST (NULL AS nvarchar) AS BillAddr,
      CAST (NULL AS int) AS CreditScore,
      False AS _from0, T1._from2,
      (T2._from2 AND T2._from2 IS NOT NULL) AS _from2
   FROM ( SELECT T.Id, T.Name, True AS _from1
          FROM HR AS T) AS T1
          LEFT OUTER JOIN (
          SELECT T.Id, T.Dept, True AS _from2
          FROM Empl AS T) AS T2
          ON T1.Id = T2.Id )
  UNION ALL (
  SELECT T.Id, T.Name, CAST (NULL AS nvarchar) AS Dept,
          T.Score AS CredScore, T.Addr AS BillAddr,
          True AS _from0, False AS _from1, False AS _from2
  FROM Client AS T)
  ) AS T5
```

Figure 2: Query view for mapping of Fig. 1 (from Fig. 7 in [13])

$V \circ Q \supseteq \mathcal{I}_{\mathcal{C}}$, where $\mathcal{I}_{\mathcal{C}}$ is the identity on client states. Unfortunately, query containment is NP-hard for conjunctive queries and hence has exponential worst-case running time for Entity Framework's mapping language, which is even more expressive. This validation step is one reason why mapping compilation can be slow.

A second reason is that Entity Framework's mapping compiler needs to check that the update view V preserves database integrity constraints. If V does not preserve constraints, then some updates to objects will not map to a valid database state and hence will not roundtrip. Integrity constraint maintenance is also encoded as query containment tests. For example, a foreign key constraint from table R_1 to R_2 is expressed as $\pi_k(R_1) \subseteq \pi_k(R_2)$, where k is the key of R_2.

A third reason why compilation can be slow is the reasoning required to construct entities of different types. For example, producing the CASE statement in Figure 2 requires reasoning over the client model and the view expressions associated with variables _from1 and _from2 to decide which types to instantiate. CASE statement are also used to instantiate attributes restricted by conditions in the mapping and thus typically occur in both query and update views. In general, the reasoning for constructing CASE statement can be quite complex.

The smallest mappings we know of where query compilation runs for hours have 32 to 35 entity types. Their object model consists of N entity types in a "hub and rim" arrangement (see Figure 3) where (i) entity type N inherits from entity type N-1, which inherits from entity type N-2, etc., (ii) each entity type has a foreign key to M other distinct entity types, and (iii) the entire hierarchy of $N + (N * M)$ entity types is mapped into one table with a discriminator column that identifies the entity type of each row. When $N + (N * M) > 32$, compilation is very slow. For example, with N=4 and M=8, compilation takes 5 hours on an HP xw6400 work-

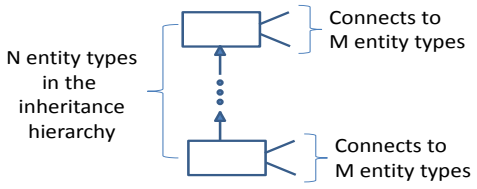

Figure 3: "Hub and rim" model

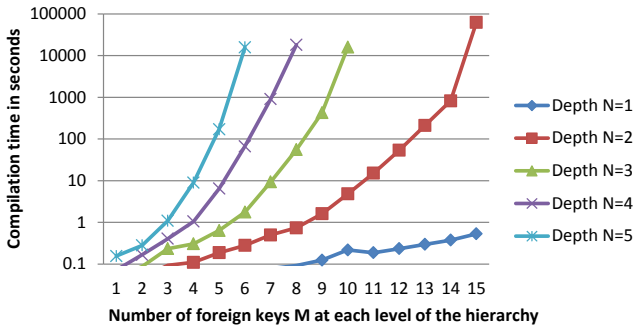

Figure 4: Compilation time of "hub and rim" model

station with an Intel Core Duo E6850 @ 3.00 GHz with 4GB of RAM running Windows 7 64-bit (see Figure 4). Compilation speed is sensitive to both schema size and mapping complexity. For example, for the same entity schema, if each entity type is mapped to a separate table, mapping compilation is under 0.2 seconds for all of the cases reported in Figure 4. Other examples of slow mappings are in Section 4.

1.2 Incremental Compilation to the Rescue

The goal of incremental mapping compilation is to reduce mapping compilation time by reducing the size of the schemas and mappings that need to be compiled and validated. It does this by leveraging the interactive nature of program development. While developing an application, a programmer typically makes small changes to a model that has already been validated and compiled into query views and update views. Since the given mapping and compiled views were correct before the change, it does not need to validate and recompile the entire mapping. Rather, it can limit its attention to the "neighborhood" of the modified part of the schema and mapping, which is the only part of the mapping that might affect the compiled views. Since the input includes generated query and update views for the pre-evolved model, the incremental compiler can reuse or modify these views in order to generate new views for the evolved model much faster than a full mapping recompilation.

Performing an incremental compilation requires a formal notion of a schema or model change. Our approach uses *schema modification operations* (SMOs)—operations that define a small change to the client schema, plus a directive on how the change maps to tables. Examples of SMOs are adding an entity type and mapping it to a new store table, or adding an association between two types and mapping the relationship to existing tables. While there are many ways to map a client schema change to tables, we focus on ones that are commonly used in ORMs, such as Table-per-type (TPT), Table-per-concrete class (TPC) and Table-per-hierarchy (TPH). These mapping types are used in the ORMs Entity Framework, Hibernate and EclipseLink, among others. We define the semantics of these mapping types in more detail in Section 3.

In general, the development environment can make SMOs directly available to developers. Or a developer can simply edit the model and then invoke a tool that generates a sequence of SMOs from a diff of the old and new models. For example, the tool can generate drop-operations of all model elements that were deleted, and then generate add-operations for elements that were added.

Our solution template for incremental compilation is comprised of four new algorithms for each type of SMO. For a particular SMO, the first algorithm generates query views for any newly added parts of the model and modified views for existing parts that are affected by the change. The second one does the same for update views. The third algorithm generates a modified mapping. And the fourth algorithm performs validity-checking.

Validity-checking of a complete mapping is defined by Algorithm 1 in [13], which has five steps. Step (1) of the algorithm checks that the left side of the mapping fragments is one-to-one. Step (5) checks that the composition of the mapping and update views is the identity. We define SMO's in such a way that if the mapping was valid before applying the SMO, then these two checks are guaranteed to hold and therefore do not have to be explicitly checked. Steps (2)-(4) of the algorithm check that update views preserve all integrity constraints. As we explained in the previous section, these constraints are not guaranteed to be preserved by the update views generated by the incremental compiler and therefore have to be checked to ensure that all updates roundtrip. These tests involve query containment checks, which are expensive in the worst-case. But since we need to focus only on the neighborhood of schema changes, the containment tests are smaller than those to validate the whole mapping. Hence, the incremental compiler usually runs much faster than a full mapping recompilation.

1.3 Contributions and Outline

In the rest of this paper, we describe algorithms that can incrementally compile mappings, and an implementation of those algorithms. The algorithms apply to schemas and mappings expressed in the language supported by Entity Framework, an ORM that has been shipping with Microsoft's .NET Framework since 2007.

Like any compiler, our incremental mapping compiler is necessarily specific to the language it compiles. However, we hope that the paper offers a template for how one would go about developing algorithms for other schema and mapping languages. Moreover, our implementation was done using public interfaces of the Entity Framework and without modifying Entity Framework source code, thereby showing that at least in this case, such a compiler can be developed as an add-on to an existing ORM.

We report on performance experiments of incremental compilation on both synthetic and customer mappings. In all cases, incremental compilation was at least 300 times faster than full compilation, reducing compilation time of hours or tens of minutes to at most one minute and often much less.

The main contributions of this paper are as follows:

- We introduce and formalize the problem of incremental compilation of object-to-relational mappings to circumvent the expensive validation of large and complex mappings.
- We present an algorithm to solve the problem for mappings expressed in the language of Microsoft's Entity Framework.
- We report on an implementation of the algorithm and show it has excellent performance on large test cases.

The paper is organized as follows. Section 2 formalizes the problem of incremental compilation and describes the schema and mapping languages we consider in the rest of the paper. Section 3 presents algorithms for incremental compilation. Section 4 describes our implementation of the algorithms and reports on its per-

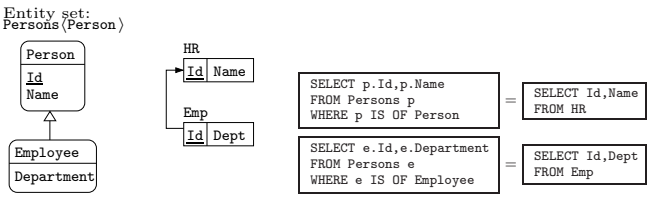

Figure 5: Example of client schema, store schema, and mapping fragments (example from [13])

formance. Section 5 summarizes related work. Section 6 presents the conclusion and future work.

2. PROBLEM FORMALIZATION

In Entity Framework (EF), the client schema $\mathcal{C}$ is expressed in EDM [4], an extended entity-relationship model comprised of entity types organized in an inheritance hierarchy. Instances of entity types, called *entities*, can be grouped into persistent collections called *entity sets*. An entity set of type E contains entities of type E or any entity type that inherits from E. Each entity type E has a set of attributes, denoted $\mathrm{att}(E)$, which includes a primary key. Two entity types E_1 and E_2 can be related by an association type. Each of its instances, called an *association*, connects entities of types E_1 and E_2. An association can have cardinality 1:1, 1:n, or m:n. For simplicity, we omit the use of EDM complex types; all of our results are easily extended to cover them.

We assume the store schema $\mathcal{S}$ is relational. It consists of a set of tables. Each table T has attributes, denoted $\mathrm{att}(T)$, which includes a primary key. It may also have foreign keys, each of which maps one or more attributes of T to the key of another table. A relational schema can be considered a restricted EDM schema, with no inheritance or associations.

Mappings and views are expressed in Entity SQL, a SQL dialect where queries can range over entity sets or associations. Relevant details of Entity SQL are described below.

2.1 Mapping fragments

EF's mapping language is composed of constraints of the form $Q_\mathcal{C} = Q_\mathcal{S}$, called *mapping fragments*, where $Q_\mathcal{C}$ and $Q_\mathcal{S}$ are project-select queries over the client and store schemas, respectively, with a relational output and a limited form of disjunction and negation. An example of a mapping given by mapping fragments in Entity SQL is shown in Figure 5. More precisely, a mapping fragment is an expression of the form

$$\pi_\alpha\left(\sigma_\psi\left(\mathcal{E}\right)\right) \;=\; \pi_\beta\left(\sigma_\chi\left(R\right)\right). \qquad (1)$$

In the above expression, $\mathcal{E}$ is the name of an entity set in the client schema, α is a sequence of attribute names containing key attributes of entities in $\mathcal{E}$, and ψ is an AND-OR combination of expressions of the form IS OF E, IS OF (ONLY E), A IS null, A IS NOT null, and $A\ \theta\ c$, where E is an entity type, A is an attribute name, c is a constant value, and θ is a comparison operator ($=$, $\leq$, etc.). An expression of the form IS OF E is satisfied by all entities of type E and derived types. For example, in Figure 5, the expression IS OF Person is satisfied by Person and Employee entities. An expression IS OF (ONLY E) is satisfied by E entities but not by entities of derived types. Moreover, in (1), R is the name of a relational table in the store schema, β is a sequence of attribute names containing the key of R, and χ is an AND-OR combination of expressions of the form A IS null, A IS NOT null, and $A\,\theta\,c$.

For example, in Figure 5, the first mapping fragment specifies that ids and names of entities of type Person (including derived types) are mapped to table HR(Id,Name). In our abstract notation, this mapping fragment is

$$\pi_{\texttt{Id,Name}}\big(\sigma_{\texttt{IS OF Person}}(\texttt{Persons})\big) = \pi_{\texttt{Id,Name}}(\texttt{HR}).$$

Mapping fragments can also map associations, in which case they are of the form $\pi_{\alpha_1\alpha_2}\left(\mathcal{A}\right) \;=\; \pi_\beta\left(\sigma_\chi\left(R\right)\right)$, where $\mathcal{A}$ is an association set, α_1 and α_2 are the key attributes of the entity types participating in the association, and β, χ and R are as in (1). EF assumes that every association set is mentioned in a single mapping fragment. We make the same assumption in this paper.

A set Σ of mapping fragments specifies a *mapping* $\mathcal{M} \subseteq \mathcal{C} \times \mathcal{S}$ given by the set of pairs (c,s) that satisfy the fragments. Here, we overload the symbols $\mathcal{C}$ or $\mathcal{S}$ to also represent the set of instances that the schemas $\mathcal{C}$ or $\mathcal{S}$ allow. Formally, a set Σ of mapping fragments defines the mapping

$$\mathcal{M} = \{(\mathrm{c},\mathrm{s}) \mid Q_\mathcal{C}(\mathrm{c}) = Q_\mathcal{S}(\mathrm{s}) \text{ for every mapping fragment } Q_\mathcal{C} = Q_\mathcal{S} \in \Sigma\}.$$

2.2 Mapping compilation

The notion of roundtripping was introduced in [13] to ensure that client data can be losslessly stored in the store schema. A mapping $\mathcal{M} \subseteq \mathcal{C} \times \mathcal{S}$ *roundtrips* if $\mathcal{M} \circ \mathcal{M}^{-1} = \mathcal{I}_\mathcal{C}$, where $\mathcal{I}_\mathcal{C}$ is the identity relation over the client schema. Whenever a mapping $\mathcal{M}$ roundtrips we say that $\mathcal{M}$ is *valid*. For every valid mapping $\mathcal{M}$ one can construct *query view* $Q : \mathcal{S} \to \mathcal{C}$ and *update view* $V : \mathcal{C} \to \mathcal{S}$ that together roundtrip the data. These views are constructed such that $V \subseteq \mathcal{M} \subseteq Q^{-1}$, which ensures that $V \circ Q = \mathcal{I}_\mathcal{C}$ [13]. Q is expressed as a set of views for each entity and association set in the client schema, while V is a set of views for every store table mentioned in a mapping fragment. The process of validating $\mathcal{M}$ and constructing Q and V is called *mapping compilation*.

In this paper we assume that a query view for an entity type E is an expression of the form $(Q_E \mid \tau_E)$ where Q_E is a relational query and τ_E is an expression that states how to construct entities from the relational output of Q_E. For instance, for the mapping in Figure 5, a query view for entity type Person can be specified as:

$$Q_{\texttt{Person}}:\ \pi_{\texttt{Id,Name}}(\texttt{HR}) \bowtie \pi_{\texttt{Id,Dept AS Department,true AS from\_Emp}}(\texttt{Emp})$$

with $\tau_{\texttt{Person}}$ the expression

```
if (from_Emp = true) then Employee(Id,Name,Department)
else Person(Id,Name)
```

Notice that the query view constructs entities of type Person and of derived type Employee. Since this information is obtained from two tables, the query view uses a special attribute (from_Emp) to track from which table the data came. Even in this simple scenario the query view needs outer join and if-else statements to construct entities. Similarly, we assume that an update view for a table T is an expression $(Q_T \mid \tau_T)$. For example, for table Emp, $Q_{\texttt{Emp}}$ is

$$\pi_{\texttt{Id,Department AS Dept}}\big(\sigma_{\texttt{IS OF Employee}}(\texttt{Persons})\big)$$

and $\tau_{\texttt{Emp}}$ is simply Emp(Id,Dept).

2.3 Problem Statement

We are now ready to formalize the incremental mapping compilation problem.

PROBLEM (INCREMENTAL COMPILATION). *Assume that we have a client schema $\mathcal{C}$, a store schema $\mathcal{S}$, and a mapping $\mathcal{M} \subseteq \mathcal{C} \times \mathcal{S}$ specified by a set Σ of mapping fragments, such that*

- *mapping $\mathcal{M}$ roundtrips, and*

- *$\mathcal{M}$ has been compiled into a set of query and update views.*

Given an SMO describing an incremental change to $\mathcal{C}$, adapt mapping $\mathcal{M}$ into a new valid mapping $\mathcal{M}'$, and compute query and update views for $\mathcal{M}'$.

The new mapping $\mathcal{M}'$ in the Incremental Compilation problem should also satisfy a soundness restriction. To formalize it we need some terminology. Let $\mathcal{C}'$ be the new client obtained from $\mathcal{C}$ after the incremental change. We can associate to every client state $c \in \mathcal{C}$ a corresponding client state $f(c) \in \mathcal{C}'$ such that c and $f(c)$ coincide in all the components that $\mathcal{C}$ and $\mathcal{C}'$ share, and any new components introduced in $\mathcal{C}'$ are empty in $f(c)$. For instance, if the incremental change adds a new entity type E, then for every $c \in \mathcal{C}$, $f(c)$ is a state in $\mathcal{C}'$ that is equal to c in all components of $\mathcal{C}$ and has no entities of type E. Thus, the soundness restriction is formalized as follows: for every client state $c \in \mathcal{C}$, mappings $\mathcal{M}$ and $\mathcal{M}'$ should be equivalent, that is, $(c, d) \in \mathcal{M}$ if and only if $(f(c), d) \in \mathcal{M}'$. This ensures that the mapping adaptation phase is not arbitrary, preventing the incremental compiler from dropping the mapping $\mathcal{M}$ and constructing a new mapping $\mathcal{M}'$ from scratch.

3. INCREMENTAL COMPILATION

In this section, we present our solution for the incremental compilation problem. We consider in detail two classes of SMOs:

- AddEntity: create a new entity type as a leaf in a type hierarchy.
- AddAssociation: create a new association between two existing entity types.

Each of the above classes describes a change to the client schema. Naturally, there are many other evolution classes that one can consider such as dropping entity types, adding properties to existing types, and transforming an association into an inheritance relationship. Our work includes algorithms for these SMOs, but due to lack of space, we describe them only briefly in Section 3.4.

3.1 Adding entity types: the TPT/TPC case

We start by describing an SMO that adds an entity type using the most common strategies for mapping entity types to tables, namely, *Table per Type* (TPT) and *Table per Concrete Type* (TPC) [8]. Let E be an entity type to be added to the client schema. In the TPT strategy, the primary key and the non-inherited attributes of E are mapped into a single table, say T. To construct entities of type E, data from T are joined with the tables that store the ancestors of E in the type hierarchy. The mappings shown in Figure 5 illustrate a TPT mapping. On the other hand, when one follows TPC, all the attributes of E are mapped into a table, say R. Thus, in TPC, to construct entities of type E we only need to use data from table R.

The language of mapping fragments allows more general variations of TPT and TPC. We capture all of them in the following carefully-crafted SMO that maps an arbitrary subset of the attributes of E (along with its primary key) to some table.

$\mathrm{AddEntity}(E, E', \alpha, P, T, f)$, where:

- E is the new entity type to be added and E' is the parent of E in the type hierarchy,
- α is a subset of the attributes $\mathrm{att}(E)$ that contains the primary key of E (PK_E),
- P is an ancestor of E in the hierarchy such that $\alpha \cup \mathrm{att}(P) = \mathrm{att}(E)$,
- T is a table in the store schema that is not mentioned in any mapping fragment,
- $f : \alpha \to \mathrm{att}(T)$ is a 1-1 function that maps the primary key of E to the primary key of T.

We require that for every $A \in \alpha$, $\mathrm{dom}(A) \subseteq \mathrm{dom}(f(A))$, to ensure that all attribute values of the new entity can be stored in table T. Moreover, all attributes in the set difference $\mathrm{att}(T) \setminus f(\alpha)$ must be nullable.

The semantics of $\mathrm{AddEntity}(E, E', \alpha, P, T, f)$ is given by the following mapping fragment:

$$\varphi_E \ : \ \pi_\alpha(\sigma_{\texttt{IS OF } E}(\mathcal{E})) = \pi_{f(\alpha)}(T). \qquad (2)$$

where $\mathcal{E}$ is the entity set to which E was added.

Example 1 Consider the scenario in Figure 1 and suppose the client schema is initially composed only of the entity type `Person` with mapping fragments $\Sigma_1 = \{\varphi_1\}$ where

$$\varphi_1 \ : \ \pi_{\texttt{Id,Name}}(\sigma_{\texttt{IS OF Person}}(\texttt{Persons})) = \pi_{\texttt{Id,Name}}(\texttt{HR}).$$

(Types `Customer`, `Employee` and association `Supports` are not initially in the client schema.) Suppose that we have computed query view $(Q^1_{\texttt{Person}} \mid \tau^1_{\texttt{Person}}) = (\pi_{\texttt{Id,Name}}(\texttt{HR}) \mid \texttt{Person(Id,Name)})$ and update view $(Q^1_{\texttt{HR}} \mid \tau^1_{\texttt{HR}}) = (\pi_{\texttt{Id,Name}}(\sigma_{\texttt{IS OF Person}}(\texttt{Persons})) \mid \texttt{HR(Id,Name)})$ for Σ_1. Σ_1 is valid and the views roundtrip.

We now want to add an entity type `Employee(Id, Name, Department)` that derives from `Person`, and map it as TPT to table `Emp`, as in Figure 1. For this we use the SMO

$$\mathrm{AddEntity}(\texttt{Employee,Person,(Id,Department),Person,Emp,}f_{\mathrm{E}}),$$

where $f_{\mathrm{E}}(\texttt{Id}) = \texttt{Id}$ and $f_{\mathrm{E}}(\texttt{Department}) = \texttt{Dept}$. Following Eq. 2, its semantics is

$$\varphi_2 \ : \ \pi_{\texttt{Id,Department}}(\sigma_{\texttt{IS OF Employee}}(\texttt{Persons})) = \pi_{\texttt{Id,Dept}}(\texttt{Emp}).\square$$

The mapping fragment φ_E in Eq. 2 only specifies how attributes α are mapped into table T. The reference to ancestor P of E in $\mathrm{AddEntity}$ states that all the attributes of E that are not mapped to T should be mapped as attributes of P. This distinction between attributes mapped to T and those mapped like attributes of P is what gives the SMO the power to express a range of mapping strategies.

TPT and TPC can be obtained as special cases of AddEntity. To map a new entity E via the TPC strategy one should use $\mathrm{AddEntity}(E, E', \mathrm{att}(E), \texttt{NIL}, T, f)$, which maps all E attributes (inherited and non-inherited) to table T. To map the new entity E via TPT, one should use $\mathrm{AddEntity}(E, E', (\mathrm{att}(E) \setminus \mathrm{att}(E')) \cup PK_E, E', T, f)$, which maps only the non-inherited attributes of E plus its primary key to table T. The reference to entity type E' states that the remaining attributes of entities of type E are mapped in the same way as the parent E' of E in the hierarchy.

As we explained in Section 1.2, we need four new algorithms to implement an SMO: adapt and create query views, adapt and create update views, adapt the previous mapping fragments, and validate the new mapping. We describe these in the next four sections.

3.1.1 Incrementally computing query views

Algorithm 1 incrementally computes query views when adding a new entity type by using $\mathrm{AddEntity}$. In the algorithm, we assume that $(Q^-_F \mid \tau^-_F)$ is the query view for entity type F before the incremental compilation. When constructing queries we use expression $(f(\alpha)\ \texttt{AS}\ \alpha)$ to denote a renaming of a sequence of attributes.

The algorithm is based on a somewhat involved case analysis of entity types E and P and those in between them in the hierarchy. In lines (3)-(10) the algorithm constructs the query view Q_E depending on whether a reference to an ancestor P has been specified. If

Algorithm 1 Query Views for $\mathrm{AddEntity}(E, E', \alpha, P, T, f)$

1: For every entity type F with $F \neq E$, let $(Q_F^- \mid \tau_F^-)$ be the query view for F before the addition of E
2: let t_E be a *fresh* attribute name
3: $\tau_E := E(\mathrm{att}(E))$
4: **if** $P = \texttt{NIL}$ **then**
5: $\quad Q_E := \pi_{(f(\alpha)\ \texttt{AS}\ \alpha)}(T)$
6: $\quad Q_{\text{aux}} := \pi_{(f(\alpha)\ \texttt{AS}\ \alpha,\ \text{true}\ \texttt{AS}\ t_E)}(T)$
7: **else**
8: $\quad Q_E := Q_P^- \bowtie \pi_{(f(\alpha)\ \texttt{AS}\ \alpha)}(T)$
9: $\quad Q_{\text{aux}} := Q_P^- \bowtie \pi_{(f(\alpha)\ \texttt{AS}\ \alpha,\ \text{true}\ \texttt{AS}\ t_E)}(T)$
10: **end if**
11: $\texttt{anc} := \{F \mid F$ is an entity type which is an ancestor of P in the type hierarchy$\}$
12: **for all** entity type F in $\texttt{anc}$ **do**
13: $\quad Q_F := Q_F^- \bowtie \pi_{(f(\alpha)\ \texttt{AS}\ \alpha,\ \text{true}\ \texttt{AS}\ t_E)}(T)$
14: $\quad \tau_F := \texttt{if}\ (t_E = \text{true})\ \texttt{then}\ \tau_E\ \texttt{else}\ \tau_F^-$
15: **end for**
16: $\texttt{p} := \{F \mid F$ is a proper ancestor of E and a proper descendant of P in the type hierarchy$\}$
17: **for all** entity type F in $\texttt{p}$ **do**
18: $\quad Q_F := Q_F^- \cup Q_{\text{aux}}$
19: $\quad \tau_F := \texttt{if}\ (t_E = \text{true})\ \texttt{then}\ \tau_E\ \texttt{else}\ \tau_F^-$
20: **end for**
21: **for all** entity type F not in $\texttt{p} \cup \texttt{anc} \cup \{E\}$ **do**
22: $\quad (Q_F \mid \tau_F) := (Q_F^- \mid \tau_F^-)$
23: **end for**
24: **return** query views $(Q_F \mid \tau_F)$ for all entity types F

$P \neq \texttt{NIL}$, the query view for E is constructed by joining the previous query view for P and table T (line (8)); otherwise, if $P = \texttt{NIL}$ only table T is used in the query view (line (5)).

Lines (12) to (20) incrementally compute the new query views for all ancestors of E in the type hierarchy, which are the only types affected by the addition, since query views for these ancestors now need to consider entities of type E too[1]. The algorithm constructs these new query views using left-outer-joins for ancestors of P (lines (12)-(15)) and unions for entity types in between E and P (lines (16)-(20)). In both cases, a new attribute t_E is used to *test the provenance* of tuples: whenever t_E is true, the entity created by the query view should be of type E (expression τ_E). Otherwise, the entity should be created as before (using expression τ_F^- for entity type F). Note how the algorithm leverages previously computed views, thus avoiding full mapping recompilation.

Example 2 Continuing with Example 1, let us walk through Algorithm 1 to compute the new query views generated by AddEntity. Notice that `Person` is playing the role of P. Thus, line (8) constructs the query view for `Employee` as follows:

$$Q^2_{\texttt{Employee}} \quad : \quad Q^1_{\texttt{Person}} \bowtie \pi_{\texttt{Id,Dept AS Department}}(\texttt{Emp}) = \pi_{\texttt{Id,Name}}(\texttt{HR}) \bowtie \pi_{\texttt{Id,Dept AS Department}}(\texttt{Emp})$$

and line (3) constructs $\tau^2_{\texttt{Employee}}$ as `Employee(Id,Name,Department)`.

To reconstruct the query view for `Person` we follow lines (12)-(15) of the algorithm. Thus, the new query view for `Person` is obtained by using $(Q^1_{\texttt{Person}} \mid \tau^1_{\texttt{Person}})$ as follows:

$$Q^2_{\texttt{Person}} \quad : \quad Q^1_{\texttt{Person}} \bowtie \pi_{\texttt{Id,Dept AS Department,true AS}\ t_{\texttt{E}}}(\texttt{Emp}) = \pi_{\texttt{Id,Name}}(\texttt{HR}) \bowtie \pi_{\texttt{Id,Dept AS Department,true AS}\ t_{\texttt{E}}}(\texttt{Emp})$$

and $\tau^2_{\texttt{Person}}$ is $(\texttt{if}\ (t_{\texttt{E}} = \text{true})\ \texttt{then}\ \tau^2_{\texttt{Employee}}\ \texttt{else}\ \tau^1_{\texttt{Person}})$. □

3.1.2 *Incrementally computing update views*

The process to construct update views is shown in Algorithm 2. The SMO $\mathrm{AddEntity}(E, E', \alpha, P, T, f)$ states that attributes α of entity type E should be mapped to table T; thus, the update view for T is very simple and is constructed in lines (2)-(3). The expression $f(\alpha)\ \mathrm{pad}\ \mathrm{att}(T)$ states that all attributes from T that are missing in $f(\alpha)$ are padded with null values.

The use of inheritance in update views introduces a subtle issue. The semantics of AddEntity states that the attributes of E not in α should be mapped like entities of type P. Thus, every update view that considered data from P entities, should now also include data from E entities. An expression $\texttt{IS OF}\ P$ in an update view refers to entities of the (new) derived type E, which is what we want. The issue is that expressions of the form $\texttt{IS OF}\ (\texttt{ONLY}\ P)$ incorrectly exclude E entities. Thus, line (7) of the algorithm replaces every expression $\texttt{IS OF}\ (\texttt{ONLY}\ P)$ by $\texttt{IS OF}\ (\texttt{ONLY}\ P) \vee \texttt{IS OF}\ E$.

Example 3 Continuing with Example 2, we construct update views. Lines (2) and (3) of Algorithm 2 generate the update view $(Q^2_{\texttt{Emp}} \mid \tau^2_{\texttt{Emp}})$ for table `Emp` where $\tau^2_{\texttt{Emp}}$ is just `Emp(Id,Dept)` and

$$Q^2_{\texttt{Emp}} \quad : \quad \pi_{\texttt{Id,Department AS Dept}}(\sigma_{\texttt{IS OF Employee}}(\texttt{Persons})).$$

The only previously computed update view is the view for `HR`. But since it does not mention an expression $\texttt{IS OF}\ (\texttt{ONLY Person})$, it is unchanged by the algorithm. That is, $(Q^2_{\texttt{HR}} \mid \tau^2_{\texttt{HR}}) = (Q^1_{\texttt{HR}} \mid \tau^1_{\texttt{HR}})$. □

Line (13) of the algorithm makes a complementary adaptation for entity types *in between* E and P. Consider an update view that includes a condition of the form $\texttt{IS OF}\ F$ where F is a proper ancestor of E and a proper descendant of P in the hierarchy. Entities of type E satisfy the condition $\texttt{IS OF}\ F$. However, all attributes of E should be mapped either to table T or to the tables to which attributes of P are mapped. Thus, line (14) replaces the expression $\texttt{IS OF}\ F$ by an expression that rules out entities of type E.

Example 4 To illustrate the issues with inheritance clauses, we now add an entity type `Customer(Id,Name,CredScore,BillAddr)` that derives from `Person`, and map the new entity type as TPC to table `Client` as shown in Figure 1. For this we use the SMO

$$\mathrm{AddEntity}(\texttt{Customer,Person,}\ \texttt{(Id,Name,CredScore,BillAddr),NIL,Client,}\ f_{\texttt{C}}),$$

where $f_{\texttt{C}}(\texttt{Id}) = \texttt{Cid}$, $f_{\texttt{C}}(\texttt{Name}) = \texttt{Name}$, $f_{\texttt{C}}(\texttt{CredScore}) = \texttt{Score}$, and $f_{\texttt{C}}(\texttt{BillAddr}) = \texttt{Addr}$. Notice that `NIL` plays the role of P.

Now we compute the query and update views for the new entity, and recompute the previous queries. Since $P = \texttt{NIL}$, line (5) of Algorithm 1 constructs the query view for entity `Customer` by considering only `Client`. Thus $Q^3_{\texttt{Customer}}$ is:

$$\pi_{\texttt{Cid AS Id,Name,Score AS CredScore,Addr AS BillAddr}}(\texttt{Client}),$$

and $\tau^3_{\texttt{Customer}}$ is `Customer(Id,Name,CredScore,BillAddr)`. `Person` is a proper ancestor of `Customer` and a proper descendant of $P = \texttt{NIL}$. Thus, lines (17)-(19) of Algorithm 1 incrementally compute the query view for `Person` as $Q^2_{\texttt{Person}} \cup Q_{\text{aux}}$, where Q_{aux} is $Q^3_{\texttt{Customer}}$ but with the attribute cast $(\text{true}\ \texttt{AS}\ t_{\texttt{C}})$; that is, $Q^3_{\texttt{Person}}$ is

$$\left(\pi_{\texttt{Id,Name}}(\texttt{HR}) \bowtie \pi_{\texttt{Id,Dept AS Department,true AS}\ t_{\texttt{E}}}(\texttt{Emp})\right) \cup \pi_{\texttt{Cid AS Id,Name,Score AS CredScore,Addr AS BillAddr,true AS}\ t_{\texttt{C}}}(\texttt{Client}),$$

with $\tau^3_{\texttt{Person}}$ the expression

```
if (t_C = true) then Customer(Id,Name,CredScore,BillAddr)
else if (t_E = true) then Employee(Id,Name,Department)
else Person(Id,Name)
```

[1]For simplicity, in this and subsequent algorithms we assume that every entity type in a hierarchy is a descendant of `NIL`.

Algorithm 2 Update Views for $\mathrm{AddEntity}(E, E', \alpha, P, T, f)$

1: For every table R, with $R \neq T$, let $(Q_R^- \mid \tau_R^-)$ be the update view for R before the addition of E
2: $Q_T := \pi_{(\alpha \texttt{ AS } f(\alpha)) \text{ pad } \mathrm{att}(T)}\big(\sigma_{\texttt{IS OF } E}(\mathcal{E})\big)$
3: $\tau_T := T(\mathrm{att}(T))$
4: **for all** update view $(Q_R^- \mid \tau_R^-)$ with $R \neq T$ **do**
5: $\quad Q_R := Q_R^-$
6: $\quad \tau_R := \tau_R^-$
7: $\quad$ **for all** expression $\gamma = \texttt{IS OF (ONLY } P\texttt{)}$ in Q_R **do**
8: $\quad\quad$ replace γ by $\big(\texttt{IS OF (ONLY } P\texttt{)} \vee \texttt{IS OF } E\big)$ in Q_R
9: $\quad$ **end for**
10: $\quad \mathtt{p} := \{F \mid F$ is a proper ancestor of E and a proper descendant of P in the type hierarchy$\}$
11: $\quad$ **for all** entity type F in $\mathtt{p}$ **do**
12: $\quad\quad d_{\mathtt{p}}(F) := \{F' \mid F'$ is a descendant of F and $F' \in \mathtt{p}\}$
13: $\quad\quad$ **for all** expression $\gamma = \texttt{IS OF } F$ in Q_R **do**
14: $\quad\quad\quad$ replace γ in Q_R by

$$\bigvee_{F' \in d_{\mathtt{p}}(F)} \Big(\texttt{IS OF (ONLY } F'\texttt{)} \vee \bigvee_{F'' \in ch_{\mathtt{p}}(F')} \texttt{IS OF } F''\Big)$$

$\quad\quad\quad$ where $ch_{\mathtt{p}}(F')$ are the children of F' not occurring in $\mathtt{p} \cup \{E\}$
15: $\quad\quad$ **end for**
16: $\quad$ **end for**
17: **end for**
18: **return** update views $(Q_R \mid \tau_R)$ for every relation R

This view is equivalent to the one in Figure 2 (in Entity SQL notation) for the same mapping, but we have compiled it incrementally. The query view for `Employee` does not change.

Line (2) of Algorithm 2 computes the update view of $Q^3_{\texttt{Client}}$ as

$$\pi_{\texttt{Id AS Cid,null AS Eid,Name,CredScore AS Score,BillAddr AS Addr}}\big(\sigma_{\texttt{IS OF Customer}}(\texttt{Persons})\big)$$

Lines (7)-(9) obtain $Q^3_{\texttt{HR}}$ from $Q^2_{\texttt{HR}}$ by replacing `IS OF Person` by `IS OF (ONLY Person)` $\vee$ `IS OF Employee`. □

3.1.3 Incrementally computing mapping fragments

Now consider the generation of mapping fragments. It is tempting to think we can just add φ_E in Eq. (2) to the set of mapping fragments. However, it is not that simple since it might produce an invalid mapping.

Let Σ^- be the set of mapping fragments before executing $\mathrm{AddEntity}$. Before adding φ_E to Σ^- we need to adapt the previous fragments to make them consistent with the semantics of $\mathrm{AddEntity}$, in particular with the fact that only α attributes are mapped to T and the rest are mapped as entities of type P. The adaptation process is essentially the same as for update views. We generate a set Σ^* from Σ^- replacing every occurrence of $\texttt{IS OF (ONLY } P\texttt{)}$ by $\texttt{IS OF (ONLY } P\texttt{)} \vee \texttt{IS OF } E$, and every occurrence of $\texttt{IS OF } F$ with F an entity type in between E and P, by an expression that rules out entities of type E. The new set is $\Sigma = \Sigma^* \cup \{\varphi_E\}$. It is not difficult to see that Σ^- and Σ are equivalent under the constraint that entity type E is empty, thus satisfying our soundness restriction on the adaptation of mapping fragments.

Example 5 In Example 1, the SMO AddEntity of `Employee` is applied to a model whose mapping has one fragment, φ_1. Since φ_1 has no expression `IS OF (ONLY Person)`, it is not changed and the new set of mapping fragments is $\Sigma_2 = \{\varphi_1, \varphi_2\}$ with φ_2 as in Example 1:

$$\varphi_2 \;:\; \pi_{\texttt{Id,Department}}\big(\sigma_{\texttt{IS OF Employee}}(\texttt{Persons})\big) = \pi_{\texttt{Id,Dept}}(\texttt{Emp}).$$

In Example 4, the SMO AddEntity of `Customer` is applied to a model whose mapping is Σ_2. In this SMO, `Person` is a proper descendant of P (`NIL` in this case) and a proper ancestor of `Customer`. Since mapping fragment φ_1 mentions the expression `IS OF Person`, we replace it by `IS OF (ONLY Person)` $\vee$ `IS OF Employee`. Thus, the new set of mapping fragments is $\Sigma_3 = \{\varphi_1', \varphi_2, \varphi_3\}$, where φ_1' is the mapping fragment

$$\pi_{\texttt{Id,Name}}\big(\sigma_{\texttt{IS OF (ONLY Person)} \vee \texttt{IS OF Employee}}(\texttt{Persons})\big) = \pi_{\texttt{Id,Name}}(\texttt{HR})$$

φ_2 is as before, and φ_3 is the mapping fragment

$$\pi_{\texttt{Id,Name,CredScore,BillAddr}}\big(\sigma_{\texttt{IS OF Customer}}(\texttt{Persons})\big) = \pi_{\texttt{Cid,Name,Score,Addr}}(\texttt{Client}).\square$$

3.1.4 Incrementally validating the mapping

Validating an incrementally modified mapping requires a modified version of Algorithm 1 in [13] that checks that (1) all possible new entity values are covered by the mapping, and (2) these new values can be mapped to a valid store state. Part (1) is ensured by the condition $\mathrm{att}(E) = \mathrm{att}(P) \cup \alpha$ and the fact that the mapping was valid before the addition, thus ensuring that all possible values of P entities are covered by the mapping. For part (2), we must ensure that no integrity constraint in the store is violated when mapping data of the new entity type. This requires some analysis.

Besides domain constraints, which are validated in the application of $\mathrm{AddEntity}$, our store schema considers only key and foreign-key constraints. It is easy to see that adding entities of type E cannot violate key constraints. There are two cases, E's target table T and all other tables. Table T is not previously used so only primary keys of E entities are stored in the key of T. All other tables that store data of E entities already store data of entity types from which E inherits, and these mappings did not violate any key constraint before adding E. Formally, suppose that data of some entity `e` of type E violates a key constraint when stored in a table R. Data from `e` is stored in R because of an update view that considers entity types of some ancestor F of E. Consider a state in which `e` has been replaced by an entity `f` of type F, which is identical to `e` in all the attributes that both share. Storing `f` in R would also violate the key, which contradicts the assumption that the mapping was valid before adding E.

By contrast, adding new entities can violate foreign key constraints. Such a scenario is shown in Figure 6. Assume first that only entity type E' and association $\mathcal{A}$ are mapped such that the key attributes of E' are mapped to attributes γ in S, attributes of $\mathcal{A}$ corresponding to keys of E' are mapped to attributes β in R, and table R has a foreign key constraint $\beta \rightarrow \gamma$ to table S. In the absence of entity type E, $\beta \rightarrow \gamma$ is always satisfied, because the set of possible values of attributes β in R comes from $\mathcal{A}$, and thus are key values of entities of type E'. Now consider entity type E which inherits from E' and is mapped as TPC to table T. Association $\mathcal{A}$ may relate entities of the new type E and store the corresponding key values in table R. Notice that all attributes (including keys) of entities of type E are stored in table T. Thus, keys of these entities are not stored in table S. This implies that the foreign key constraint $\beta \rightarrow \gamma$ is violated whenever an entity e of type E participates in association $\mathcal{A}$, since e's key is stored in R but not in S. Another type of violation can occur when table T has a foreign key to a table that was previously mentioned in mapping fragments.

We now explain how we check the validity of foreign key constraints after $\mathrm{AddEntity}(E, E', \alpha, P, T, f)$. If there is an entity

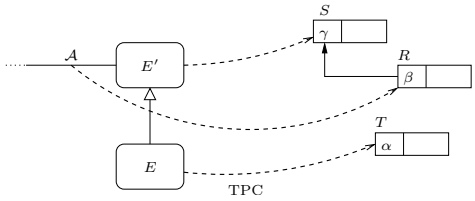

Figure 6: Violation of foreign key constraints

type F that is a proper ancestor of E and a proper descendant of P, and an association $\mathcal{A}$ that has F as endpoint, we do the following. Assume that $\mathcal{A}$ is mapped to table R and the key attributes PK_F of F are mapped to attributes β in R. We need to check two cases.

1. First check the query containment

$$\pi_{PK_F \text{ AS } \beta}(\mathcal{A}) \subseteq \pi_{\beta}(Q_R),$$

using the update view Q_R generated in the previous section. This tests whether we can store associations of entities of the new entity type E (that derives from F) in the same table R.

2. If there is a foreign key $\beta' \rightarrow \gamma$ where $\beta \cap \beta' \neq \emptyset$ from R to some table S, then we check that the foreign key is not violated by testing the query containment

$$\pi_{\beta' \text{ AS } \gamma}(Q_R) \subseteq \pi_{\gamma}(Q_S),$$

where Q_R and Q_S are the update views for R and S, respectively, generated in the previous section.

We also have to check that existing foreign keys in T are not violated as follows.

3. If T has a foreign key constraint $\beta \rightarrow \beta'$ to a table T' with $\beta \cap f(\alpha) \neq \emptyset$, then check the query containment

$$\pi_{\beta \text{ AS } \beta'}(Q_T) \subseteq \pi_{\beta'}(Q_{T'}),$$

where Q_T and $Q_{T'}$ are the update views for T and T'.

If any of the containments above fails, the incremental compilation is aborted since the mapping is not valid after the addition of E.

Example 6 To validate the mappings in Example 5, first consider the mapping for AddEntity of `Employee`. There are no associations (it has not been added yet), so we only need to check that the foreign key constraint `Emp.Id` $\rightarrow$ `HR.Id` is not violated (check 3 in the previous section). Thus, we have to test $\pi_{\texttt{Id}}(Q^2_{\texttt{Emp}}) \subseteq \pi_{\texttt{Id}}(Q^2_{\texttt{HR}})$. By unfolding the update views in the previous expression we obtain

$$\pi_{\texttt{Id}}(\sigma_{\texttt{IS OF Employee}}(\texttt{Persons})) \subseteq \pi_{\texttt{Id}}(\sigma_{\texttt{IS OF Person}}(\texttt{Persons}))$$

which holds since `Employee` inherits from `Person`. Thus, the mapping is valid.

Next consider the mapping for AddEntity of `Customer`. Here, we do not need to check validity of the foreign key constraint `Client.Eid` $\rightarrow$ `Emp.Id` since none of the attributes of the new entity type is mapped to attribute `Eid`. □

3.2 Adding associations

We now consider incremental compilation when adding an association set between existing components. This is considerably simpler than the addition of entity types. Recall that association sets are sets of tuples (α_1, α_2) corresponding to key attributes of the entities participating in the association. We describe in this section the addition of associations mapped to a key/foreign-key table in the store schema. For this case we consider the SMO

$$\text{AddAssocFK}(\mathcal{A}, E_1, E_2, \text{mult}, T, f), \text{ where:}$$

- $\mathcal{A}$ is the name of the new association set,
- E_1 and E_2 are the endpoints of the association,
- mult is an expression that denotes the multiplicity of the association, such that the endpoint corresponding to E_2 is not $*$ (many),
- T is a table in the store schema previously mentioned in mapping fragments,
- f is a 1-1 function $f : PK_1 \cup PK_2 \rightarrow \text{att}(T)$, where PK_1 and PK_2 are the key attributes of E_1 and E_2, respectively, and such that $f(PK_1)$ is the key of T.

For simplicity and without loss of generality, we assume that names of key attributes of E_1 and E_2 are disjoint (otherwise they can be renamed or aliases can be used). The semantics of the addition is given by the following mapping fragment

$$\varphi_{\mathcal{A}} : \pi_{PK_1, PK_2}(\mathcal{A}) = \pi_{f(PK_1), f(PK_2)}(\sigma_{f(PK_2) \text{ IS NOT null}}(T)).$$

Let Σ^- be the set of mapping fragments before adding the new association. In this case the adaptation is simple: we just add $\varphi_{\mathcal{A}}$ to Σ^-, that is, the new set of mapping fragments is $\Sigma = \Sigma^- \cup \{\varphi_{\mathcal{A}}\}$.

To ensure that the new set Σ is valid we need to verify that the addition does not violate the key and foreign key constraints in which table T participates. We need to check three scenarios:

1. Check that attributes $f(PK_2)$ in table T have not been previously used to map data from the client schema by inspecting the mapping fragments.
2. Check the containment

$$\pi_{PK_1}(\sigma_{\text{IS OF } E_1}(\mathcal{E})) \subseteq \pi_{f(PK_1) \text{ AS } PK_1}(Q^-_T),$$

with Q^-_T the update view for table T before the addition, and $\mathcal{E}$ the entity set to which entity type E_1 belongs. This ensures that the endpoint E_1 of the association can be entirely mapped to the primary key attributes of T.

3. If T has a foreign key of the form $f(PK_2) \rightarrow \beta$ to a table T' with primary key β, then check

$$\pi_{PK_2 \text{ AS } \beta}(\sigma_{\text{IS OF } E_2}(\mathcal{E})) \subseteq \pi_{\beta}(Q^-_{T'}),$$

with $Q^-_{T'}$ the update view for table T' before the addition, and $\mathcal{E}$ the entity set containing E_2 entities.

If any of the checks above fails, the incremental compilation is aborted since the mapping is not valid after the addition of $\mathcal{A}$. Notice that case (1) is necessary since the semantics of φ_A states that whenever $f(PK_2)$ is not null, then it is considered to be part of association $\mathcal{A}$.

3.2.1 Constructing views when adding associations

The construction of query views is also simple. All existing query views remain unaltered, and the query view $(Q_{\mathcal{A}} \mid \tau_{\mathcal{A}})$ for the new association set is defined as

$$\begin{aligned} Q_{\mathcal{A}} &: \pi_{f(PK_1) \text{ AS } PK_1, f(PK_2) \text{ AS } PK_2}(\sigma_{f(PK_2) \text{ IS NOT null}}(T)), \\ \tau_{\mathcal{A}} &: \mathcal{A}(PK_1, PK_2). \end{aligned}$$

That is, to construct the associations we just consider the attributes $f(PK_1), f(PK_2)$ of T. Similarly, for update views we only have to incrementally recompute the update view for table T. Let $(Q^-_T \mid \tau^-_T)$ be the update view for T before the addition. The new update view is given by

$$Q_T : \pi_{\text{att}(T) \setminus f(PK_2)}(Q^-_T) \rtimes \pi_{PK_1 \text{ AS } f(PK_1), PK_2 \text{ AS } f(PK_2)}(\mathcal{A})$$

with $\tau_T = \tau_T^-$. Notice that data stored in T now comes from the previous view Q_T^- without considering attributes $f(PK_2)$ (all the data stored in attributes $\mathrm{att}(T) \setminus f(PK_2)$ is still stored in T) and using an outer join to store values for $f(PK_2)$ whenever an association between PK_1 and PK_2 exists in the client schema.

Example 7 Continuing with our previous example, suppose we want to add association `Supports` between `Customer` and `Employee`, as shown in Figure 1. Thus we use the SMO

$$\mathrm{AddAssocFK}(\texttt{Supports},\texttt{Customer},\texttt{Employee},[*{-}0..1],\texttt{Client},f_S),$$

where $f_S(\texttt{Customer.Id}) = \texttt{Cid}$, and $f_S(\texttt{Employee.Id}) = \texttt{Eid}$. We obtain then the mapping specified by $\Sigma_4 = \{\varphi_1', \varphi_2, \varphi_3, \varphi_4\}$, where φ_4 is the fragment $\pi_{\texttt{Customer.Id,Employee.Id}}(\texttt{Supports}) = \pi_{\texttt{Cid,Eid}}(\sigma_{\texttt{Eid IS NOT null}}(\texttt{Client}))$. We now check that key and foreign key constraints in `Client` are not violated. First, it is clear that attribute `Eid` of table `Client` is not previously mentioned in the mapping fragment (check 1). Now, in order to check that the identifiers of entities of type `Customer` can be stored in table `Client` we have to check the containment $\pi_{\texttt{Id}}(\sigma_{\texttt{IS OF Customer}}(\texttt{Persons})) \subseteq \pi_{\texttt{Cid AS Id}}(Q^3_{\texttt{Client}})$ (check 2). By unfolding the definition of $Q^3_{\texttt{Client}}$ (and simplifying the expression) we obtain

$$\pi_{\texttt{Id}}(\sigma_{\texttt{IS OF Customer}}(\texttt{Persons})) \subseteq \pi_{\texttt{Cid AS Id}}(\sigma_{\texttt{IS OF Customer}}(\texttt{Persons})),$$

which clearly holds. Finally, we have to check the constraint $\texttt{Client.Eid} \rightarrow \texttt{Emp.Id}$ (check 3). So we have to test the following containment:

$$\pi_{\texttt{Id}}(\sigma_{\texttt{IS OF Employee}}(\texttt{Persons})) \subseteq \pi_{\texttt{Id}}(Q^3_{\texttt{Emp}}).$$

By unfolding the view $Q^3_{\texttt{Emp}}$ it is easy to see that the containment holds. Thus we have a valid mapping fragment. As an example of query and update views that are computed in this case, the update view for table `Client` is incrementally computed from $Q^3_{\texttt{Client}}$. In this case we have that $Q^4_{\texttt{Client}}$ is

$$\pi_{\texttt{Cid,Name,CredScore,BillAddr}}(Q^3_{\texttt{Client}}) \ \mathbb{\bowtie} \ \pi_{\texttt{Customer.Id AS Cid,Employee.Id AS Eid}}(\texttt{Supports}).$$

3.3 Entity types partitioned across tables

The SMO AddEntity assumes that the new data is stored in a single table. We developed a relaxed version of AddEntity in which one can use client-side conditions to specify that different tables are used to store data of different entities of the same type. For the sake of space, we do not describe all the details of the algorithms in this case, but just the main additional issues that need to be considered.

The SMO that we consider for adding entities in this case is $\mathrm{AddEntityPart}(E, E', P, \Gamma)$, similar to AddEntity, but in which Γ is a set $\{(\alpha_1, \psi_1, T_1, f_1), \ldots, (\alpha_n, \psi_n, T_n, f_n)\}$, where for every $i \in \{1, \ldots, n\}$ we have that α_i is a subset of the attributes of $\mathrm{att}(E)$ (containing the primary key of E), ψ_i is a satisfiable conjunction of conditions over $\mathrm{att}(E)$ and constant values, T_i is a table in the store schema, and $f_i : \alpha_i \rightarrow \mathrm{att}(T_i)$ is a 1-1 function that maps entity properties to table attributes. Since we do not consider disjunctions in ψ_i, checking satisfiability can be done efficiently if the conditions in ψ_i are equalities and inequalities. The semantics of this addition is given by the set of mapping fragments

$$\begin{aligned} \pi_{\alpha_1}\big(\sigma_{(\texttt{IS OF } E)\wedge\psi_1}(\mathcal{E})\big) &= \pi_{f(\alpha_1)}(T_1), \qquad (3)\\ &\ldots \\ \pi_{\alpha_n}\big(\sigma_{(\texttt{IS OF } E)\wedge\psi_n}(\mathcal{E})\big) &= \pi_{f(\alpha_n)}(T_n). \end{aligned}$$

As an application scenario consider adding an entity type `Person(name, age)` to entity set `Persons`, mapping entities to a table `Adult(name, age)` or `Young(name, age)` depending on the value of attribute `age`. Thus, we would use $\mathrm{AddEntityPart}$ with parameter $\Gamma = \{(\alpha_1, \psi_1, \texttt{Adult}, f_1), (\alpha_2, \psi_2, \texttt{Young}, f_2)\}$ in which $\alpha_1 = \{\texttt{name}, \texttt{age}\}$, $\psi_1 = (\texttt{age} \geq 18)$, $\alpha_2 = \{\texttt{name}, \texttt{age}\}$, $\psi_2 = (\texttt{age} < 18)$ (and f_1, f_2 identity functions). In this case the mapping fragments are

$$\begin{aligned} \pi_{\texttt{name,age}}\big(\sigma_{(\texttt{IS OF Person})\wedge\texttt{age}\geq 18}(\texttt{Persons})\big) &= \pi_{\texttt{name,age}}(\texttt{Adult}),\\ \pi_{\texttt{name,age}}\big(\sigma_{(\texttt{IS OF Person})\wedge\texttt{age}< 18}(\texttt{Persons})\big) &= \pi_{\texttt{name,age}}(\texttt{Young}). \end{aligned}$$

As for the case of the addition of entity types that we introduced in Section 3.1, the reference to the ancestor P is used to cover attributes of E that are not mapped by the above mapping fragments, by mapping them as the attributes in P. In fact if we consider a set Γ with a single tuple $(\alpha, \text{true}, T, f)$ we obtain exactly the SMO of Section 3.1. The adaptation process for the mapping fragments is essentially the same as in Section 3.1.

The verification process in this case is a bit more complex. In Section 3.1, to ensure that the data corresponding to E entities can be losslessly stored in the database, it was enough to require that $\mathrm{att}(P) \cup \alpha = \mathrm{att}(E)$ plus that constraints in the store are not violated when storing the new data. In this case, where entities are partitioned across several tables, we have to check a more complex condition to test whether the pairs (ψ_i, α_i) *cover* all the possibilities for E entities and thus no data is lost. We do this as follows. For every attribute A in $\mathrm{att}(E)$ we consider all formulas ψ_i such that either $A \in \alpha_i$ or the equality $A = c$ (with c a constant value) is a logical consequence of ψ_i. Then we check that the disjunction of all the selected formulas is a tautology.

For instance, in the `Person/Adult-Young` example, attribute `name` appears in α_1 and α_2, so we have to check that formula $\psi_1 \vee \psi_2$ is a tautology. In this case, the formula that we have to check is $(\texttt{age} \geq 18) \vee (\texttt{age} < 18)$, which is clearly a tautology. To show the necessity of considering equalities $A = c$, consider a scenario in which one adds an entity type `Person(id, name, gender)`, where `gender` is either `M` or `F`, mapped as follows. Ids are mapped to table `Men(id)` if $\texttt{gender} = \texttt{M}$ or to table `Women(id)` if $\texttt{gender} = \texttt{F}$, and all names are mapped to table `Name(id, name)`. Even though `gender` is not mapped to the store, we still have to check that its associated formula is a tautology. In this case the formula associated to attribute `gender` is $(\texttt{gender} = \texttt{M}) \vee (\texttt{gender} = \texttt{F})$ which is actually a tautology since the only possible values for `gender` are `M` and `F`.

After checking coverage of attributes, we also have to check that foreign key constraints in the store are not violated. The strategy for checking that, as well as the construction of update views, is essentially the same as in Section 3.1. In contrast, to construct the query view for entity type E we have to consider data stored in all tables T_i. The query view is the full outer join of all T_i's thus putting together all the pieces to create E entities. We also have to consider attribute values that are fixed by the client-side constraints. For instance, for the `Person/Men-Women-Name` example above, the new query view for `Person` would be

$$\pi_{\texttt{id,name}}(\texttt{Name}) \ \mathbb{\bowtie} \ \pi_{\texttt{id,M AS gender}}(\texttt{Men}) \ \mathbb{\bowtie} \ \pi_{\texttt{id,F AS gender}}(\texttt{Women})$$

Notice that the view has to set attribute `gender` to `M` when the data comes from `Men`, and to `F` when it comes from `Women`. In general, we also need to use the reference to entity type P to construct E entities as in Algorithm 1. Query views for other entity types can also be reconstructed as in Algorithm 1, using the full outer join of the T_i's instead of the single reference to table T.

3.4 Other SMOs

We have developed algorithms for some other natural SMOs. In this section we briefly describe some of them, focusing on ones

that have "interesting" compilation algorithms. Details on the algorithms can be found in [3].

We do not claim this set is able to generate all possible schemas and mappings. At least, one would like SMOs that can add and drop every feature of the client model (e.g., entity, association, and property) and modify some facets (e.g., data type and cardinality). Covering all possible object-to-relational mappings is more challenging, and we have deferred it to future work.

Section 3.1 described incremental compilation algorithms for adding an entity type via TPT and TPC. We also have algorithms for AddEntityTPH (for *Table per Hierarchy*), where data of all entities of a type hierarchy are stored in one table. A special *discriminator* column in the table is used to identify which type of entity is stored in each row. To add entity type E to table T, the query view of each ancestor F of E has to be unioned with a select-project query that retrieves rows containing entities of E; query views of other entity types are unaffected. The update view of T needs to be modified by changing expressions of the form `IS OF` E' by `IS OF (ONLY` E'`)`, where E' is the parent of E, and unioning it with a select-project query that selects entities of type E from the hierarchy's entity set. To validate the mapping fragments, besides checking that foreign key constraints are not violated, we need containment tests that check that the discriminator value that identifies E can be used as a valid discriminator value. Notice the symmetry of AddEntityTPH, which does a selection on the discriminator on the right side of mapping fragments, and AddEntityPart, which does the selection on the left side of mapping fragments.

In Section 3.2 we described how incremental compilation works when adding an association mapped to a key/foreign-key constraint in the store. Another way that associations can be mapped is to a new *join table*. This SMO is able to cover many-to-many associations which are not covered by the SMO in Section 3.2. We implemented associations mapped to join tables and we include them in our experiments in Section 4.

We also considered an SMO to add properties to an existing entity type E. In this case, the SMO states that the new property can be mapped to a table where E attributes were already mapped, or to a completely new table. In any case, we need to reconstruct query views not only for E but also for descendants of E in the type hierarchy.

Another SMO that we consider is *refactoring*. Let $\mathcal{A}$ be an association with cardinality 1 to 0..1 between entity types E_1 and E_2, respectively. In the refactoring process we delete $\mathcal{A}$ and make E_2 a derived type of E_1 (adding all attributes of entities of E_1 to E_2). The process implies that whenever an entity $\mathtt{e}_2$ of type E_2 was associated with $\mathtt{e}_1$ of type E_1 in the original schema, in the new schema we have a single entity of type E_2 containing attribute values from $\mathtt{e}_1$ and $\mathtt{e}_2$. The incremental compilation in the case of refactoring is a bit more complicated than for adding entities, as in this case not only query views for the ancestor of E_1 are affected but also query views for descendants of E_2 need to be transformed.

Finally, we also consider an SMO to drop an entity type E. In this case, we need to eliminate all references to E from mapping fragments and views. Notice that if E is not a leaf in the hierarchy, then references to E cannot just be deleted but have to be replaced by expressions that now consider all descendants of E.

4. EVALUATION

4.1 Implementation

We implemented a standalone incremental mapping compiler in C# using libraries from Microsoft's Entity Framework (EF) V4.5. Its high-level architecture is shown in Figure 7. The incremental

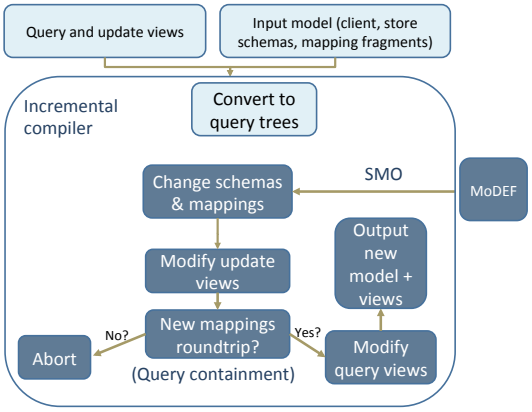

Figure 7: Architecture of the incremental mapping compiler. Light-colored components initialize the compiler. Dark-colored components perform the compilation.

compiler requires two inputs: the pre-evolved model containing the client schema, store schema and mapping fragments, and the query and update views given as Entity SQL queries.

The Entity SQL queries are embedded in a C# file that is generated by EF's mapping compiler. Our incremental compiler extracts the query and update views from that C# file and uses .NET functions provided by EF to translate the views into an internal object representation called *canonical query trees*—a published format that Microsoft supports with various library functions. It is similar to an abstract query tree or logical query plan, but is adapted for the additional constructs of Entity SQL. Our incremental compiler maintains these trees in memory for the duration of its operation.

After our compiler has taken the schemas, mapping, and views as input, the user can issue a command to make an incremental change in the client model, such as adding a new entity or association type. To determine appropriate changes to the store model and mapping fragments, we use the MoDEF system [16]. It examines existing mapping fragments in the neighborhood of the changes to determine its mapping style: TPC, TPT, or TPH. It then generates an SMO that is consistent with that mapping style. Once the SMO has been determined, the compiler proceeds to change the client and store schemas and then validate the new mappings.

To perform validation, the compiler first evolves the canonical query tree representation of the update views using one of the algorithms described earlier in Section 3, such as Algorithm 2 for AddEntityTPC/TPT. It then invokes a query containment checker, built for Entity SQL queries, to perform validation and ensure that the new mappings roundtrip. The query containment checker uses the algorithm described in [9]. If validation fails, the compiler undoes its changes to the schemas and update views and returns an exception. If it succeeds, then it evolves the canonical-query-tree representation of the query views, generates the new evolved model, and generates the Entity SQL query and update views from the evolved query trees, which it stores back in the C# file.

We implemented six SMOs in the current version of the compiler: three for AddEntity (TPT-TPC, TPH, Partitioned), two for AddAssociation (JT, FK) and one for AddProperty. In the next section, we show that these primitives offer substantial performance benefits compared to the baseline of full recompilation.

4.2 Experimental evaluation

Our experiments seek to address two main questions: (1) Does incremental compilation provide any benefits compared to full re-

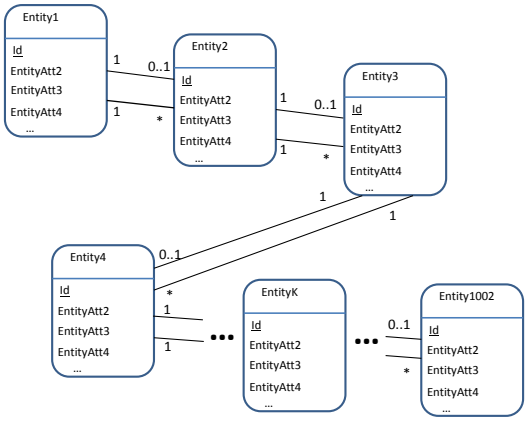

Figure 8: Chain model

compilation in EF? (2) How does the execution time of incremental compilation change with respect to different SMOs and different types of client models and mappings?

Methodology: All times quoted in this section are on a workstation running Windows 7 with a 3.0 gigahertz 64-bit Intel Core Duo processor and 8 gigabytes of memory.

We evaluated our incremental compiler over two large and diverse client models. The first is a synthetically generated model that has 1002 entity types with no inheritance relationships arranged in a chain, where each entity type is related by two associations with the next entity type in the chain. Mapping fragments consist of simple one-to-one mappings, where each entity type is mapped to its own table on the store side and each association is mapped to a key-foreign key relationship. Figure 8 shows a partial representation of this model. A full compilation of this model using the EF compiler takes 15 minutes.

In contrast to this simple, large model, the second model is a real customer model that consists of 230 different entity types over 18 non-trivial hierarchies. The deepest has four levels, and the largest has 95 entity types. They are mapped either TPT or TPH, with associations mapped to non-junction tables. A full compilation of the customer model takes 8 hours.

For each model, we chose entity types randomly as inputs to our SMOs. For SMOs that performed a TPT mapping, we added foreign key constraints for newly added tables on the store side. For each SMO, we measured the wall clock time to perform the evolution operation and averaged results across three runs. Figures 9 and 10 show the results of various SMOs. We use AE-x and AA-x to denote the AddEntity and AddAssociation primitives using mapping type x, and AEP-np-x to denote AddEntityPart with the entity set horizontally mapped across 2^n tables and vertically through mapping type x.

During testing, there were some SMOs where validation failed. Most were cases of AddEntityTPC similar to the one in Figure 6.

Results: Figure 9 shows runtimes of the SMOs on the synthetic model. Note that the y-axis is log-scale. The figure shows that incremental compilation is more than two orders-of-magnitude faster than full recompilation. We also see that all SMOs have roughly the same performance, except for AEP-np-TPT. As expected, AEP-np-TPT tends to scale exponentially with n (linearly with the number of tables) since the compiler has to validate 2^n new foreign key constraints, one for each new table. While the total runtime for each operation was 5-10 seconds, validation took only 1-2 seconds; the remaining time was taken to reload the evolved schemas using EF's object model; we found this to be a necessary step in making

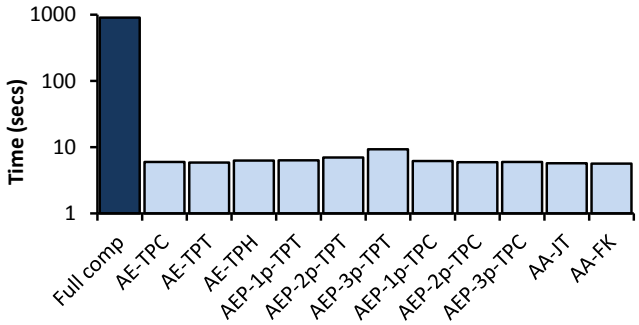

Figure 9: Results on synthetic model

a standalone compiler since the object representations of mappings and schemas in EF are only internally mutable. If integrated within EF, this step could easily be skipped, thus decreasing the overall running time significantly.

Figure 10 shows the running times for SMOs on the customer model. Most operations completed within 50 seconds, with the majority of time spent on query containment checks for mapping validation. We noticed high variability between different runs, as the update views of some entity types turned out to be more complex than others. Such cases occurred when association sets and entity sets were mapped to the same table, thus generating joins in the update views, which increase the time for containment checking. For the chosen entity types, AE-TPH overall took less time, as in some cases, the entire hierarchy was not wholly mapped to a single table (i.e., the original mapping was not TPH, and failed validation could easily be reported).

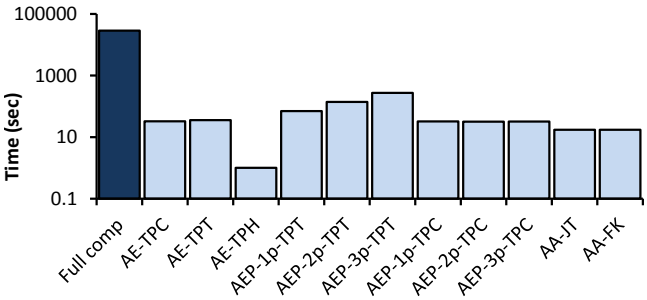

Figure 10: Results on customer model

5. RELATED WORK

The programming language community has worked on incremental compilation of program code for many years [15, 17]. It embodies a tradeoff between speed (by recompiling only a portion of a potentially much larger code base) and the efficiency of the product (because optimization decisions are frequently better when one considers larger windows of the code base).

A related technology from the programming language community is Lenses [10], which is a tool to establish updatable mappings between potentially different models A lens is an atomic transformation comprised of two functions, each describing one direction of data transformation; one constructs a mapping through composition of primitive lenses. In the classical definition of a lens, one must often have the entire state of both the source and target of the mapping to successfully propagate updates. Recent work has examined ways to instead propagate an incremental update specification only through a lens, namely Delta Lenses [5, 6] and Edit Lenses [11]. Edit Lenses in particular have some similarity with the

incremental Entity Framework in that the language of incremental updates is tailored to the target domain; lenses over lists, for example, operate on primitives that look like "insert element at index" and "reorder elements".

Despite copious amounts of research on schema evolution (enough that it merits its own bibliography [14]), there is relatively little work done on client-driven evolution scenarios in an ORM setting. One notable exception is MeDEA, which consumes schema evolution primitives against an Extended Entity Relationship model and propagates them to structural changes against a database [7]. These primitives are paired with directives on how those primitives should affect the mapping to relational storage, similar to some of the parameters to our SMOs (for example, the α parameter to AddEntity). In MeDEA, one primarily has the choice of directive from the three major mapping patterns (TPC, TPT, and TPH), and so does not necessarily have the same level of flexibility as our SMOs. In addition, MeDEA primarily concerns itself with structural changes, and thus does not need to worry about the data-level mappings and query translation for which Entity Framework compiled views are needed.

Client-driven schema evolution through a mapping is similar in spirit to the Demeter method and adaptive programming, in particular in the presence of a database [12]. Like in adaptive programming, our work allows the developer greater flexibility in design and to facilitate changes. With Demeter, one wites programs at a higher level of abstraction and employs an encapsulation of operations called *propagation patterns*. One can evolve the structure or behavior of a program by authoring refinements to those patterns, and the mapping to code objects automatically adjusts itself.

6. CONCLUSION AND FUTURE WORK

This paper introduced the problem of incremental compilation of object-to-relational mappings, to circumvent the NP-hard problem of mapping validation. We presented algorithms to solve the problem for mappings expressed in the language of Microsoft's Entity Framework. And we reported on their implementation, showing a huge speedup over full mapping re-compilation.

One area of future work is optimization of generated views. The full Entity Framework compilation algorithm has several optimization steps that it can consider [13]; e.g., it can leverage schema constraints to reduce costly operations like full outer joins into cheaper operations, such as UNION ALL and left outer joins. Incremental compilation already includes several of these optimizations inline; for instance, it directly produces left outer joins and UNION ALL operations without going through an intermediate step. It would be worth exploring whether other optimizations are possible. In our experiments, views generated by incremental compilation are identical or very similar to those generated by a full compilation. A carefully-designed comparative study of the differences between these views for different types of mappings and SMOs, and their effect on query and update performance, would be beneficial.

Another open problem is the expressiveness of the SMOs that we are considering. As we mentioned in Section 3.4, it would be interesting to develop a provably complete set of SMOs, one that can generate all schemas and mappings supported by Entity Framework. Ideally, it should be accompanied by an algorithm that, given a schema and mapping, generates a sequence of SMOs that produces the same result. Usually, many such sequences are possible. Does it matter which sequence it chooses? In particular, do some sequences complete successfully while others do not, because an SMO fails to validate? Is the efficiency of the resulting views affected by the order of SMOs? Answers to these questions affect the design of a solution and its implementation.

Acknowledgements

We are grateful for all the help we received from many members of the EF team, too numerous to mention, but especially Alexander Mineev, David Obando Chacon, Asad Khan, Tim Mallalieu, and Adi Unnithan. Jorge Pérez is funded by Fondecyt grant 11110404 and VID grant U-Inicia 11/04 Universidad de Chile.

7. REFERENCES

[1] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *SIGMOD Conference*, pages 877–888, 2007.

[2] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM TODS*, 6(4):557–575, 1981.

[3] P. A. Bernstein, M. Jacob, J. Pérez, G. Rull, and J. F. Terwilliger. Incremental mapping compilation in an object-to-relational mapping system (extended version). Technical Report MSR-TR-2013-45, Microsoft Research, June 2013.

[4] J. A. Blakeley, S. Muralidhar, and A. Nori. The ADO.NET Entity Framework: making the conceptual level real. In *ER Conference*, pages 552–565, 2006.

[5] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta- based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10:6: 1–25, 2011.

[6] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *MoDELS*, pages 304–318, 2011.

[7] E. Domínguez, J. Lloret, A. L. Rubio, and M. A. Zapata. Evolving the implementation of isa relationships in eer schemas. In *ER (Workshops)*, pages 237–246, 2006.

[8] D. W. Embley and W. Y. Mok. Mapping conceptual models to database schemas. In *Handbook of Conceptual Modeling*, pages 123–163. Springer, 2011.

[9] C. Farré, E. Teniente, and T. Urpí. Checking query containment with the cqc method. *Data Knowl. Eng.*, 53(2):163–223, 2005.

[10] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, pages 233–246, 2005.

[11] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *POPL*, pages 495–508, 2012.

[12] L. Liu, R. Zicari, W. L. Hürsch, and K. J. Lieberherr. The role of polymorphic reuse mechanisms in schema evolution in an object-oriented database. *IEEE Trans. Knowl. Data Eng.*, 9(1):50–67, 1997.

[13] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. *ACM TODS*, 33(4), 2008.

[14] E. Rahm and P. A. Bernstein. An online bibliography on schema evolution. *SIGMOD Record*, 35(4):30–31, 2006.

[15] S. P. Reiss. An approach to incremental compilation. In *SIGPLAN Conference*, pages 144–156, 1984.

[16] J. F. Terwilliger, P. A. Bernstein, and A. Unnithan. Automated co-evolution of conceptual models, physical databases, and mappings. In *ER*, pages 146–159, 2010.

[17] D. M. Ungar. The design and evaluation of a high performance smalltalk system. Technical report, Berkeley, CA, 1986.

[18] EclipseLink. http://www.eclipse.org/eclipselink/.

[19] Hibernate. http://www.hibernate.org/.

[20] Oracle TopLink. http://www.oracle.com/technology/products/ias/toplink/index.html.

[21] Ruby on rails. http://rubyonrails.org/.