Compiling Mappings to Bridge Applications and Databases

Sergey Melnik
Microsoft Research
One Microsoft Way
Redmond, WA 98052, U.S.A.
melnik@microsoft.com

Atul Adya
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, U.S.A.
adya@microsoft.com

Philip A. Bernstein
Microsoft Research
One Microsoft Way
Redmond, WA 98052, U.S.A.
philbe@microsoft.com

ABSTRACT

Translating data and data access operations between applications and databases is a longstanding data management problem. We present a novel approach to this problem, in which the relationship between the application data and the persistent storage is specified using a declarative mapping, which is compiled into bidirectional views that drive the data transformation engine. Expressing the application model as a view on the database is used to answer queries, while viewing the database in terms of the application model allows us to leverage view maintenance algorithms for update translation. This approach has been implemented in a commercial product. It enables developers to interact with a relational database via a conceptual schema and an object-oriented programming surface. We outline the implemented system and focus on the challenges of mapping compilation, which include rewriting queries under constraints and supporting non-relational constructs.

Categories and Subject Descriptors:
H.2 [Database Management], D.3 [Programming Languages]

General Terms: Algorithms, Performance, Design, Languages

Keywords: mapping, updateable views, query rewriting

1. INTRODUCTION

Developers of data-centric solutions routinely face situations in which the data representation used by an application differs substantially from the one used by the database. The traditional reason is the impedance mismatch between programming language abstractions and persistent storage [12]: developers want to encapsulate business logic into objects, yet most enterprise data is stored in relational database systems. A second reason is to enable data independence. Even if applications and databases start with the same data representation, they can evolve, leading to differing data representations that must be bridged. A third reason is independence from DBMS vendors: many enterprise applications run in the middle tier and need to support backend database systems of varying capabilities, which require different data representations. Thus, in many enterprise systems the separation between application models and database models has become a design choice rather than a technical impediment.

The data transformations required to bridge applications and databases can be very complex. Even relatively simple object-to-relational (O/R) mapping scenarios where a set of objects is partitioned across several relational tables may require transformations that contain outer joins, nested queries, and case statements in order to reassemble objects from tables (we will see an example shortly). Implementing such transformations is difficult, especially since the data usually needs to be updatable, a common requirement for many enterprise applications. A study referred to in [22] found that coding and configuring O/R data access accounts for up to 40% of total project effort.

Since the mid 1990's, client-side data mapping layers have become a popular alternative to handcoding the data access logic, fueled by the growth of Internet applications. A core function of such a layer is to provide an updatable view that exposes a data model closely aligned with the application's data model, driven by an explicit mapping. Many commercial products (e.g., Oracle TopLink) and open source projects (e.g., Hibernate) have emerged to offer these capabilities. Virtually every enterprise framework provides a client-side persistence layer (e.g., EJB in J2EE). Most packaged business applications, such as ERP and CRM applications, incorporate proprietary data access interfaces (e.g., BAPI in SAP R/3).

Today's client-side mapping layers offer widely varying degrees of capability, robustness, and total cost of ownership. Typically, the mapping between the application and database artifacts is represented as a custom structure or schema annotations that have vague semantics and drive case-by-case reasoning. A scenario-driven implementation limits the range of supported mappings and often yields a fragile runtime that is difficult to extend. Few data access solutions leverage data transformation techniques developed by the database community. Furthermore, building such solutions using views, triggers, and stored procedures is problematic for a number of reasons. First, views containing joins or unions are usually not updatable. Second, defining custom database views and triggers for every application accessing mission-critical enterprise data is rarely acceptable due to security and manageability risks. Moreover, SQL dialects, object-relational features, and procedural extensions vary significantly from one DBMS to the next.

In this paper we describe an approach to building a mapping-driven data access layer that addresses some of these challenges.

As a first contribution, we present a novel mapping architecture that provides a general-purpose mechanism for supporting updatable views. It enables building client-side data access layers in a principled way but could also be exploited inside a database engine. The architecture is based on three main steps:

- Specification: Mappings are specified using a *declarative language* that has well-defined semantics and puts a wide range of mapping scenarios within reach of non-expert users.
- Compilation: Mappings are compiled into *bidirectional views*, called query and update views, that drive query and update processing in the runtime engine.

SIGMOD'07, June 11–14, 2007, Beijing, China.

1. Data manipulation using Entity SQL

```
"SELECT p
 FROM SalesPeople AS p
 WHERE p.Hired.Year > 2000"
```

2. CRUD on objects (create/read/ update/delete)

```
foreach(Person p in aw.SalesPeople)
  if(p.Hired.Year > 2000) {...}
aw.SaveChanges();
```

3. Language-integrated queries (LINQ)

```
var people = from p in aw.SalesPeople
             where p.Hired.Year > 2000
             select p;
```

Figure 1: Interacting with data in the Entity Framework

- Execution: Update translation is done using algorithms for *materialized view maintenance*. Query translation uses view unfolding.

To the best of our knowledge, this mapping approach has not been exploited previously in the research literature or in commercial products. It raises interesting research challenges.

Our second contribution addresses one of these challenges—formulation of the mapping compilation problem and algorithms that solve it. The algorithms that we developed are based on techniques for answering queries using views and incorporate a number of novel aspects. One is the concept of bipartite mappings, which are mappings that can be expressed as a composition of a view and an inverse view. Using bipartite mappings enables us to focus on either left or right 'part' of mapping constraints at a time when generating query and update views. Another novel aspect is a schema partitioning approach that allows us to rewrite the mapping constraints using union queries and facilitates view generation. Furthermore, we discuss support for object-oriented constructs, case statement generation, bookkeeping of tuple provenance, and simplification of views under constraints.

The presented approach has been implemented in a commercial product, the Microsoft ADO.NET Entity Framework [1, 11]. A pre-release of the product is available for download.

This paper is structured as follows. In Section 2 we give an overview of the Entity Framework. In Section 3 we outline our mapping approach and illustrate it using a motivating example. The mapping compilation problem is stated formally in Section 4. The implemented algorithms are presented in Section 5. Experimental results are in Section 6. Related work is discussed in Section 7. Section 8 is the conclusion.

2. SYSTEM OVERVIEW

The ADO.NET Entity Framework provides a mapping-driven data access layer for developers of data-intensive applications. It comprises an extended entity-relationship data model, called EDM, and a set of design-time and run-time services. The services allow developers to describe the application data using an entity schema and interact with it at a high level of abstraction that is appropriate for business applications.

The system offers three major data programming facilities (see Figure 1). First, developers can manipulate the data represented in the entity schema using Entity SQL, an extension of SQL that can deal with inheritance, associations, etc. This capability enables general-purpose database development against the conceptual schema and is important for applications that do not need an object layer, such as business reporting. Second, the entity schema can be used to generate object-oriented interfaces in several major programming languages. In this way, persistent data can be accessed using create/read/update/delete operations on objects. Third, queries against the generated object model can be posed using a language-integrated query mechanism (LINQ [26]), which enables compile-time checking of queries.

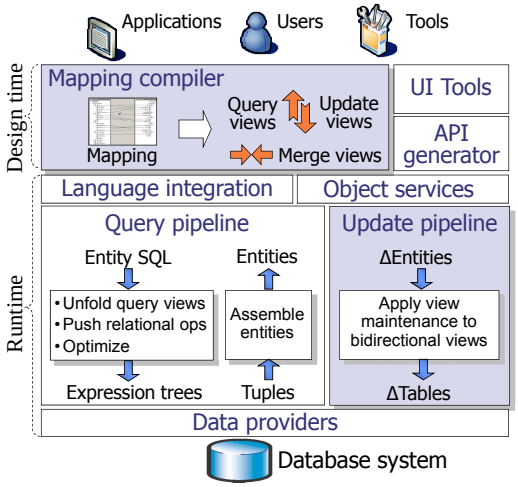

Figure 2: System architecture

EDM is an extended entity-relationship model. It distinguishes entity types, complex types, and primitive types. Instances of entity types, called entities, can be organized into persistent collections called entity sets. An entity set of type T holds entities of type T or any type that derives from T. Each entity type has a key, which uniquely identifies an entity in the entity set. Entities and complex values may have properties holding other complex values or primitive values. Like entity types, complex types can be specialized through inheritance. However, complex values can exist only as part of some entity. Entities can participate in 1:1, 1:n, or m:n associations, which essentially relate the keys of the respective entities. Notice that a relational schema with key and foreign key constraints can be viewed as a simple EDM schema that contains a distinct entity set and entity type for each relational table. We exploit this property to uniformly specify and manipulate mappings. Henceforth we use the general term *extent* to refer to entity sets, associations, and tables.

Entity SQL is a data manipulation language based on SQL. Entity SQL allows retrieving entities from entity sets and navigating from an entity to a collection of entities reachable via a given association. Path expressions can be used to 'dot' into complex values. Type interrogation can be done using ⟨value⟩ IS OF ⟨type⟩ or IS OF ONLY predicates. Entity SQL allows instantiating new entities or complex values similarly to the 'new' construct in programming languages. It supports a tuple constructor that produces row types and uses reference types similarly to SQL99. The semantics of Entity SQL statements used in the paper is explained where necessary.

The system architecture is depicted (abridged) in Figure 2. It comprises several major components: query and update pipelines, object services, mapping compiler, metadata services (not shown), data providers, and others. The shaded components are discussed in more detail in subsequent sections. All data access requests go through a data transformation runtime, which translates data and data access operations using a mapping between the entity schema and the relational schema. The translated data access operations are fed into a data provider, one for each supported database system, which turns them into expressions in a specific SQL dialect.

3. MAPPING APPROACH

In this section we present the three main steps of our approach: specifying a mapping as a set of constraints, compiling a mapping into bidirectional views, and using view maintenance algorithms to perform update translation.

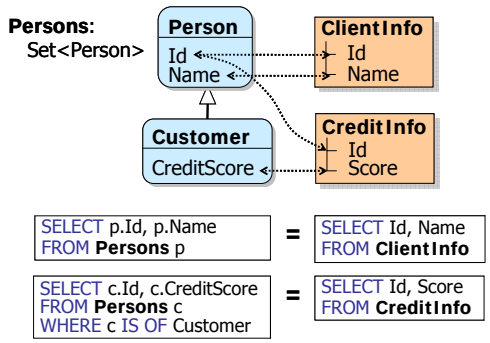

Figure 3: Mapping between entities (left) and tables (right)

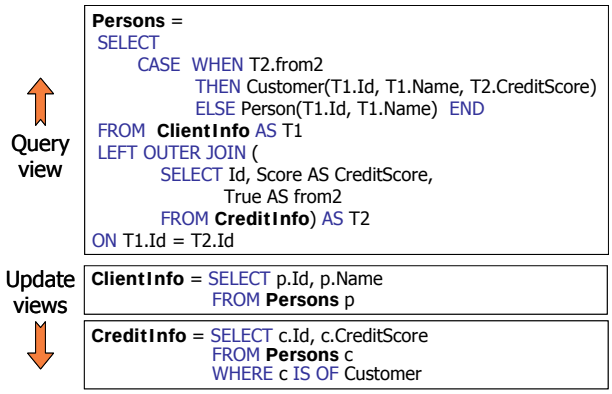

Figure 4: Bidirectional views compiled from mapping in Figure 3

Mappings. A mapping is specified using a set of mapping fragments. Each mapping fragment is a constraint of the form $Q_{\text{Entities}} = Q_{\text{Tables}}$ where Q_{Entities} is a query over the entity schema (on the client side) and Q_{Tables} is a query over the database schema (on the relational store side). A mapping fragment describes how a portion of entity data corresponds to a portion of relational data. In contrast to a view, a mapping fragment does not need to specify a complete transformation that assembles an entity set from tables or vice versa. A mapping can be defined using an XML file or a graphical tool.

To illustrate, consider the sample mapping scenario in Figure 3. It depicts an entity schema with entity types Person and Customer whose instances are accessed via the extent Persons. On the store side there are two tables, ClientInfo and CreditInfo, which represent a vertical partitioning of the entity data. The mapping is given by two fragments shown in Figure 3, and is visualized using lines between schema elements. The first fragment specifies that the set of (Id, Name) values for all entities in Persons is identical to the set of (Id, Name) values retrieved from the ClientInfo table. Similarly, the second fragment tells us that (Id, CreditScore) values for all Customer entities can be obtained from the CreditInfo table.

Bidirectional views. The mapping compiler (see Figure 2) takes a mapping as input and produces bidirectional views that drive the data transformation runtime. These views are specified in Entity SQL. *Query views* express entities in terms of tables, while *update views* express tables in terms of entities. Update views may be somewhat counterintuitive because they specify persistent data in terms of virtual constructs, but as we show later, they can be leveraged for supporting updates in an elegant way. The generated views respect the mapping in a sense that will be defined shortly and have the following properties (simplified here and elaborated in subsequent sections):

- Entities = QueryViews(Tables)
- Tables = UpdateViews(Entities)
- Entities = QueryViews(UpdateViews(Entities))

The last condition, called the *roundtripping criterion*, ensures that all entity data can be persisted and reassembled from the database in a lossless fashion. The mapping compiler included in the Entity Framework guarantees that the generated views satisfy the roundtripping criterion. It raises an error if no such views can be produced from the input mapping.

Figure 4 shows the query and update views generated by the mapping compiler for the mapping in Figure 3. In general, the views are significantly more complex than the input mapping, as they explicitly specify the required data transformations. For example, to reassemble the Persons extent from the relational tables, one needs to perform a left outer join between ClientInfo and CreditInfo tables, and instantiate either Customer or Person entities depending on whether or not the respective tuples from CreditInfo participate in the join. In the query pipeline (Figure 2), queries against the entity schema can now be answered by unfolding the query views in the queries, pulling up all non-relational constructs, and sending the relational-only portion of the query to the database server.

Update translation. A key insight exploited in our mapping architecture is that view maintenance algorithms can be leveraged to propagate updates through bidirectional views. This process is illustrated in Figure 5. Tables hold persistent data. Entities represent a virtual state, only a tiny fraction of which is materialized on the client. The goal is to translate an update ΔEntities on the virtual state of Entities into an update ΔTables on the persistent state of Tables. This can be done using the following two steps (we postpone the discussion of merge views shown in the figure):

1. View maintenance:
   ΔTables = ΔUpdateViews (Entities, ΔEntities)
2. View unfolding:
   ΔTables = ΔUpdateViews (QueryViews(Tables), ΔEntities)

Step 1 applies view maintenance algorithms to update views. This produces a set of *delta expressions*, ΔUpdateViews, which tell us how to obtain ΔTables from ΔEntities and a snapshot of Entities. Since the latter is not fully materialized on the client, Step 2 uses view unfolding to substitute Entities by query views in the computed delta expressions. Together, these steps give us an expression that takes as input the initial persistent database state and the update to entities, and computes the update to the database.

This approach yields a clean, uniform algorithm that works for both object-at-a-time and set-based updates (i.e., those expressed using data manipulation statements). In practice, Step 1 is often sufficient for update translation since many updates do not directly depend on the current database state; an example is given below. In those situations we have ΔTables = ΔUpdateViews(ΔEntities).

To illustrate, consider the update views in Figure 4. These views are very simple, so applying view maintenance rules is straightforward. For insertions we obtain the ΔUpdateViews expressions as:

ΔClientInfo = SELECT p.Id, p.Name FROM ΔPersons p
ΔCreditInfo = SELECT c.Id, c.CreditScore FROM ΔPersons c
WHERE c IS OF Customer

So, for inserted entity ΔPersons = $\{\langle$Customer(1, 'Alice', 700)$\rangle\}$, we compute row insertions ΔClientInfo = $\{\langle$1, 'Alice'$\rangle\}$, ΔCreditInfo = $\{\langle$1, 700$\rangle\}$. In general, update views can get much more complex. Using view maintenance techniques allows supporting a very large class of updates. It accounts in a uniform way

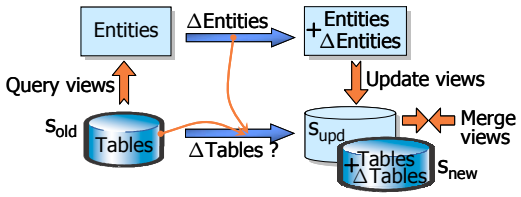

Figure 5: Update translation

for tricky update translations where multiple extents are updated, a deletion from an extent becomes an update in the store, etc.

4. MAPPING COMPILATION PROBLEM

In this section we state the mapping compilation problem formally. We start with a basic problem statement and refine it after introducing merge views. After that we describe the mapping language supported by the mapping compiler.

Since the data transformation runtime is driven by query and update views, one might wonder why a mapping is needed. We considered several alternatives before settling on the mapping compilation approach. One alternative is to let the developers supply both query and update views. This is problematic because checking the roundtripping criterion for Entity SQL views is undecidable. To see this, consider a trivial query view $E = R$ and an update view $R = E - (Q_1 - Q_2)$ where E is an entity set, R is a relational table, and Q_1 and Q_2 are queries on the entity schema that do not mention E. Unfolding the update view yields the roundtripping condition $E = E - (Q_1 - Q_2)$. It holds if and only if $Q_1 \subseteq Q_2$, which is undecidable for any relationally complete language [14]. Although it is possible to restrict the query and update views to a subset of Entity SQL for which containment is decidable, writing them by hand is hard and requires significant database expertise.

Another alternative is to obtain query views from update views. This requires testing the injectivity of update views and inverting them, which is also undecidable for Entity SQL, as can be seen using the construction in the paragraph above. A third alternative is to obtain update views from query views. This requires solving the view update problem. As was shown in [13], finding a unique update translation for even quite simple (query) views is rarely possible.

Therefore, we follow the mapping compilation approach, where query and update views are generated from mappings and are guaranteed to roundtrip. Currently, the Entity Framework only accepts the views produced by the built-in mapping compiler.

4.1 Mappings and data roundtripping

We start with a general problem statement that makes no assumptions about the languages used for specifying schemas, mappings, and views. To emphasize that, we refer to the entity schema as a 'client' schema and to the relational schema as a 'store' schema.

A *schema* defines a set of states (also called instances). Let $\mathcal{C}$ be the set of valid client states, i.e., all instances satisfying the client schema and all its schema constraints. Similarly, let $\mathcal{S}$ be the set of valid store states, i.e., those conforming to the store schema. Occasionally we use the same symbol ($\mathcal{C}$, $\mathcal{S}$, etc.) to denote the schema itself, when its role is clear from the context.

A *mapping* between the client and store schema specifies a binary relation between $\mathcal{C}$ states and $\mathcal{S}$ states. Set Σ_{map} of mapping constraints expressed in some formal language defines the mapping $map = \{(c, s) \mid c \in \mathcal{C}, s \in \mathcal{S}, (c, s) \models \Sigma_{map}\}$ consisting of all states (c, s) that satisfy every constraint in Σ_{map}. We say that map is given by Σ_{map}. A *view* is a total functional mapping that maps each instance of a given schema to an instance of a result schema.

The first question that we address is under what conditions a mapping $map \subseteq \mathcal{C} \times \mathcal{S}$ describes a valid data access scenario. The job of the mapping layer is to enable the developer to run queries and updates on the client schema as if it were a regular database schema. That is, the mapping must ensure that each database state of $\mathcal{C}$ can be losslessly encoded in $\mathcal{S}$. Hence, each state of $\mathcal{C}$ must be mapped to a distinct database state of $\mathcal{S}$, or to a set of database states of $\mathcal{S}$ that is disjoint from any other such set. If this condition is satisfied we say that the mapping *roundtrips* data, denoted as

$$map \circ map^{-1} = Id(\mathcal{C})$$

i.e., the composition[1] of the mapping with its inverse yields the identity mapping on $\mathcal{C}$. Notice that map may be non-functional. As an example, consider a mapping just like the one in Figure 3 except that CreditInfo table has an extra column Date. This column is not referenced in the mapping constraints. Hence, the client schema provides access to a proper subset of the data in the store, i.e., there exist multiple corresponding store states for each client state.

The next question we consider is what it means to obtain query and update views that roundtrip data and respect the mapping. Our statement of this problem is based on the following theorem:

THEOREM 1 (DATA ROUNDTRIPPING) Let $map \subseteq \mathcal{C} \times \mathcal{S}$. Then, $map \circ map^{-1} = Id(\mathcal{C})$ if and only if there exist two (total) views, $q : \mathcal{S} \to \mathcal{C}$ and $u : \mathcal{C} \to \mathcal{S}$, such that $u \subseteq map \subseteq q^{-1}$. □

If the views u and q above are given as sets of constraints (or view definitions) Σ_u and Σ_q, then $u \subseteq map \subseteq q^{-1}$ means that Σ_u implies Σ_{map}, which in turn implies Σ_q for all instances in $\mathcal{C} \times \mathcal{S}$. It is easy to show that for each q and u satisfying the above theorem $u \circ q = Id(\mathcal{C})$, called the roundtripping criterion in Section 3. Hence, we formulate the following *data roundtripping problem*:

> For a given $map \subseteq \mathcal{C} \times \mathcal{S}$, construct views q and u expressed in some language L, such that $u \subseteq map \subseteq q^{-1}$, or show that such views do not exist.

We refer to q and u as the query view and update view, respectively. Sometimes, we use the plural form 'views' to emphasize that q and u are specified as sets of view definitions (e.g., as shown in Figure 4).

4.2 Merge views

In typical mapping scenarios, only part of the store data is accessible through the client schema. Some tables or columns, such as CreditInfo.Date mentioned above, may not be relevant to the application and not exposed through the mapping. Usually, such unexposed information needs to remain intact as updates are performed against the store. To address this requirement we use the concept of *merge views*. A merge view $m : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ combines the updated store state s_{upd} computed by the client and the old store state s_{old} into the new store state s_{new} (see Figure 5).

To illustrate, consider the merge view for the extended table CreditInfo(Id, Score, Date):

```
CreditInfo_new = SELECT upd.Id, upd.Score, old.Date
                 FROM CreditInfo AS upd
                 LEFT OUTER JOIN CreditInfo_old AS old
                 ON upd.Id = old.Id
```

The above view together with the query and update views shown in Figure 4 determine completely the update behavior for the extended CreditInfo table. The left outer join ensures that the new customer entities added on the client appear in the CreditInfo_{new} table, and

[1]Composition, inverse, and range of mappings are defined in a standard way as the respective algebraic operations on binary relations.

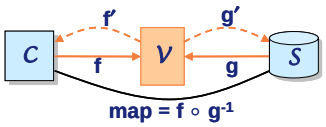

Figure 6: Bipartite mapping between $\mathcal{C}$ and $\mathcal{S}$

the deleted ones get removed. If a customer's credit score gets updated, the Date value remains unchanged.

In contrast to query and update views, whose purpose is to reshape data between the client and the store, merge views capture the store-side state transition behavior. This behavior may go beyond preserving unexposed data. For example, if Date denotes the last date on which the credit score was modified, it may be necessary to reset it to the current date upon each update. More generally, merge views may implement various *update policies* such as updating timestamps, logging updates, resetting certain columns, or rejecting deletions (using full outer join in the merge view).

Currently, we use merge views exclusively for preserving unexposed data. The formal criterion can be stated as

$$\forall s \in Range(map) : m(u(q(s)), s) = s$$

It requires that a client that retrieves all store data and writes it back unmodified leaves the store in the unchanged state. This property needs to hold for all mapped store states, i.e., the ones that are consistent with the specified mapping.

The extended roundtripping criterion that takes merge views into account can be stated as

$$\forall c \in \mathcal{C}, \forall s \in Range(map) : q(m(u(c), s)) = c$$

It requires that applying the update view u to an arbitrary client state c, followed by merging the computed store state with the existing store state, must allow reassembling c using the query view q from the merged store state.

4.3 Bipartite mappings

The mappings in the Entity Framework are specified using a set of mapping fragments $\Sigma_{map} = \{Q_{C1}{=}Q_{S1}, \ldots, Q_{Cn}{=}Q_{Sn}\}$. A mapping fragment is a constraint of the form $Q_C = Q_S$ where Q_C is a query over the client schema and Q_S is a query over the store schema. We call such mappings bipartite mappings.

A *bipartite mapping* is one that can be expressed as a composition mapping of a view with an inverse of a view with the same view schema. Thus, the mapping given by Σ_{map} above can be expressed as a composition mapping $f \circ g^{-1}$ where the view $f : \mathcal{C} \rightarrow \mathcal{V}$ is given by queries $Q_{C1}, \ldots, Q_{Cn}$, the view $g : \mathcal{S} \rightarrow \mathcal{V}$ is given by queries $Q_{S1}, \ldots, Q_{Sn}$, and $\mathcal{V}$ corresponds to the view schema $V_1, \ldots, V_n$ induced by these queries (see Figure 6). We refer to V_i as a *fragment view (symbol)*.

A key property of bipartite mappings is using equality in mapping constraints instead of inclusion. Inclusion constraints of the form $Q_{src} \subseteq Q_{tgt}$ used in source-to-target data exchange and query answering settings are inadequate for data roundtripping because they specify a 'one-way' relationship between the schemas. For example, the above query inclusion does not constrain the source database; it may remain empty for each target database.

As we demonstrate shortly, bipartite mappings enable us to compile mappings by applying answering-queries-using-views techniques to the views f and g, one after another. This reduces the complexity of the solution and enables developing largely symmetric algorithms for generating query and update views.

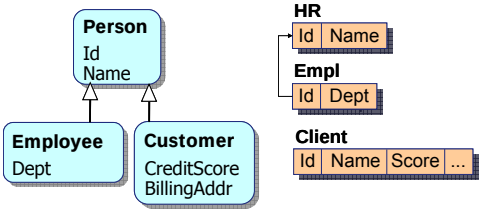

Figure 7: Mapping scenario illustrating a combination of vertical and horizontal partitioning

4.4 Mapping language

The bipartite mappings taken as input by the mapping compiler are specified using constraints of the form $Q_C = Q_S$ where Q_C and Q_S are Entity SQL queries. In the current version of the system, these queries, which we call *fragment queries*, are essentially project-select queries with a relational-only output and a limited form of disjunction and negation. This class of mapping constraints allows expressing a large number of mapping scenarios yet is sufficiently simple that mapping compilation can be performed effectively.

The specification of a fragment query Q is given below:

Q ::= SELECT P [, P]* FROM E AS e WHERE C

C ::= C AND C | C OR C | P IS NULL | P IS NOT NULL |
$P = c$ | P IS OF T | P IS OF (ONLY T)

P ::= e | $P.A$

where E is an extent with alias e, A is a property of an entity type or complex type, c is a scalar constant, and T is an entity type or a complex type. The return type of Q is required to be a row of scalars and needs to contain the key properties of E.

To illustrate why our mapping language supports disjunction consider the scenario in Figure 7. A combination of horizontal and vertical partitioning is used to store the inheritance hierarchy shown on the left. Table HR holds (portions of) instances of Person and Employee, but no Customer instances. The mapping fragment for HR can be specified using OR as shown below. Any other way of expressing the mapping would require using negation or joins:

```
SELECT p.Id, p.Name FROM Persons AS p
WHERE p IS OF (ONLY Person) OR p IS OF (ONLY Employee)
=
SELECT t.Id, t.Name FROM HR AS t
```

The mapping language specified above allows describing most inheritance mapping scenarios proposed in the literature and implemented in commercial products. In particular, it supports table-per-hierarchy, table-per-type (vertical partitioning), table-per-concrete-type (horizontal partioning) strategies, and all their combinations. Furthermore, it supports entity splitting scenarios where entities are stored in different tables based on certain property values.

5. VIEW GENERATION ALGORITHM

We start by explaining the intuition behind the algorithm. The key principle that we use is reducing the mapping compilation problem to that of finding exact rewritings of queries using views [21].

5.1 Intuition

Consider the problem of obtaining query views from mapping $map = f \circ g^{-1}$. The view f on $\mathcal{C}$ and the view g on $\mathcal{S}$ are specified as $\{V_1{=}Q_{C1}, \ldots, V_n{=}Q_{Cn}\}$ and $\{V_1{=}Q_{S1}, \ldots, V_n{=}Q_{Sn}\}$, respectively. Suppose that the view f is materialized. To preserve all information from $\mathcal{C}$ in the materialized view, f must be lossless,

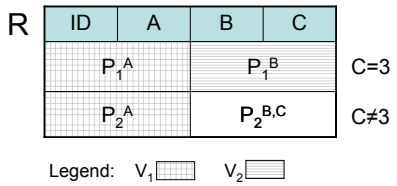

Figure 8: Partitioning relation R based on V_1 and V_2

i.e., an injective[2] function. In this case, we can reconstruct the state of $\mathcal{C}$ from the materialized view by finding an exact rewriting of the identity query on $\mathcal{C}$ using the view f. Suppose $f' : \mathcal{V} \rightarrow \mathcal{C}$ is such a rewriting (depicted as a dashed arrow in Figure 6), given by a set of queries on $V_1, \ldots, V_n$. Hence, we can unfold g in f' to express $\mathcal{C}$ in terms of $\mathcal{S}$. That gives us the desired query view $q = f' \circ g$.

Update views and merge views can be obtained in a similar fashion by answering the identity query on $\mathcal{S}$. The extra subtlety is that g need not be injective and may expose only a portion of the store data through the mapping. This raises two issues. First, the exact rewriting of the identity query on $\mathcal{S}$ may not exist. That is, to leverage answering-queries-using-views techniques we need to extract an injective view from g. Second, we need to ensure that the store information not exposed in the mapping can flow back into the update processing in the form of a merge view. Before we present a general solution, consider the following example:

EXAMPLE 1 Suppose that the store schema contains a single relation R(ID, A, B, C). Let $map = f \circ g^{-1}$ where g is defined as

$$V_1 = \pi_{\mathrm{ID,A}}(R)$$
$$V_2 = \pi_{\mathrm{ID,B}}(\sigma_{\mathrm{C=3}}(R))$$

The identity query on R cannot be answered using fragment views V_1 and V_2 since g is non-injective and loses some information in R. So, we translate the store schema into an equivalent partitioned schema containing relations P_1^A(ID, A), P_1^B(ID, B), P_2^A(ID, A), $P_2^{B,C}$(ID, B, C) with the following schema constraints: for all $i \neq j$, $\pi_{\mathrm{ID}}(P_i^X) \cap \pi_{\mathrm{ID}}(P_j^Y) = \emptyset$ and $\pi_{\mathrm{ID}}(P_i^X) = \pi_{\mathrm{ID}}(P_i^Y)$. The partitioning of R is illustrated in Figure 8. The equivalence between the partitioned schema and the store schema $\{R\}$ is witnessed by two bijective views p and r, where p is defined as

$$P_1^A = \pi_{\mathrm{ID,A}}(\sigma_{\mathrm{C=3}}(R))$$
$$P_1^B = \pi_{\mathrm{ID,B}}(\sigma_{\mathrm{C=3}}(R))$$
$$P_2^A = \pi_{\mathrm{ID,A}}(\sigma_{\mathrm{C\neq 3}}(R))$$
$$P_2^{B,C} = \pi_{\mathrm{ID,B,C}}(\sigma_{\mathrm{C\neq 3}}(R))$$

and r is defined as

$$R = \pi_{\mathrm{ID,A,B,3}}(P_1^A \bowtie P_1^B) \cup (P_2^A \bowtie P_2^{B,C})$$

The above partitioning scheme is chosen in such a way that the view g can be rewritten in terms of union queries on the partitions. Thus, g can be restated in terms of P_1^A, P_1^B, and P_2^A as follows:

$$V_1 = P_1^A \cup P_2^A$$
$$V_2 = P_1^B$$

We call partitions P_1^A, P_1^B, P_2^A *exposed* because they appear in the above rewriting. They are depicted as a shaded region in Figure 8. Partition $P_2^{B,C}$ is *unexposed* (white region). Notice that the above rewriting is injective, i.e., information-preserving, on the schema formed by the exposed partitions. Due to the constraint $\pi_{\mathrm{ID}}(P_1^A) = \pi_{\mathrm{ID}}(P_1^B)$ on the partitioned schema, we can reconstruct the exposed partitions from V_1 and V_2 as follows:[3]

$$P_1^A = V_1 \bowtie V_2$$
$$P_1^B = V_2$$
$$P_2^A = V_1 \,\overline{\ltimes}\, V_2$$

Now we have all the building blocks to construct both g' in Figure 6 and the merge view. Let R_{old}, R_{new}, and R_{upd} denote respectively the old state of R in the store, the new state obtained by merging R_{upd} and R_{old}, and the updated state of R computed using g'. R_{upd} is populated from the exposed partitions, ignoring the information in the unexposed partitions[4]:

$$\begin{aligned} R_{upd} =& \pi_{\mathrm{ID,A,B,3}}(P_1^A \bowtie P_1^B) \cup \pi_{\mathrm{ID,A,NULL,NULL}}(P_2^A) \\ =& \pi_{\mathrm{ID,A,B,3}}(V_1 \bowtie V_2) \cup \pi_{\mathrm{ID,A,NULL,NULL}}(V_1 \,\overline{\ltimes}\, V_2) \end{aligned}$$

Since $\pi_{\mathrm{ID}}(P_1^A) = \pi_{\mathrm{ID}}(P_1^B)$, so $\pi_{\mathrm{ID}}(V_2) \subseteq \pi_{\mathrm{ID}}(V_1)$ and R_{upd} can be simplified as follows:

$$R_{upd} = \pi_{\mathrm{ID,A,B,3}}(V_1 \rightarrowtriangle V_2)$$

The merge view for R_{new} assembles the exposed partitions that carry the updated client state (R_{upd}) and the unexposed partitions holding the old store state (R_{old}). The goal is to keep as much of the old store information in the unexposed partitions as possible. (Notice that keeping the complete information is undesirable. For example, preserving all of $P_2^{B,C}$ data from the old store state would prevent the client from deleting any tuples from R with $C \neq 3$, which are partially visible through V_1.) More precisely, the constraint that we exploit is that partition $P_2^{B,C}$ of R_{old} is contained in partition $P_2^{B,C}$ of R_{new} (but is not equivalent to it). Unfolding the definitions of partitions, while replacing R by R_{upd} for all exposed partitions and replacing R by R_{old} for all unexposed partitions, yields:

$$\begin{aligned} R_{new} =& \sigma_{\mathrm{C=3}}(R_{upd}) \cup \\ & (\pi_{\mathrm{ID,A}}(\sigma_{\mathrm{C\neq 3}}(R_{upd})) \rightarrowtriangle \pi_{\mathrm{ID,B,C}}(\sigma_{\mathrm{C\neq 3}}(R_{old}))) \\ =& \pi_{\mathrm{ID,A,B,CASE\ldots AS\ C}}(R_{upd} \rightarrowtriangle R_{old}) \end{aligned}$$

where the abbreviated case statement is:

CASE WHEN R_{old}.ID IS NOT NULL THEN R_{old}.C ELSE 3 END

Composing the merge view with g' produces

$$R_{new} = \pi_{\mathrm{ID,A,B,CASE\ldots AS\ C}}(V_1 \rightarrowtriangle V_2 \rightarrowtriangle R_{old})$$

Unfolding f (from $map = f \circ g^{-1}$) in the above expression states V_1 and V_2 in terms of the client schema and produces the final transformation that drives the update pipeline. □

The example motivates several issues. First, we need to justify that the above approach produces query, update, and merge views that satisfy the conditions from Section 4, i.e., solve the data roundtripping problem. Second, we need to develop a partitioning scheme to express the client and store schema using an equivalent schema that allows rewriting the mapping constraints using union queries. This rewriting simplifies the algorithm for reconstructing partitions and testing injectivity of f when constructing query views. Notice that the choice of the partitioning scheme is sensitive to the mapping language. Third, as we show below, the generated case statements can become quite complex and require

[2] f is injective if $f(x) = z$ and $f(y) = z$ implies $x = y$

[3] $\overline{\ltimes}$ is the left anti-semijoin, $\rightarrowtriangle$ is the left outer join

[4] We use extended π operator which may contain constants and computed expressions in the projection list

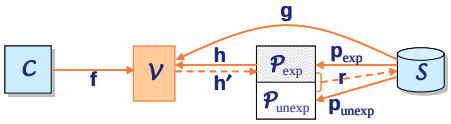

Figure 9: Obtaining u and m using partitioning scheme

careful reasoning. Finally, unfolding f in the expression $V_1 \supset\!\bowtie V_2$ may require further simplification to avoid gratuitous self-joins and self-unions if for example both terms are queries over the same entity set in the client schema. We discuss each of these issues in the subsequent sections.

5.2 Mapping validation

The approach exemplified in the previous section works only if we start with a mapping that roundtrips. To illustrate what can go wrong, suppose that f in Example 1 is given as $\{E_1 = V_1, E_2 = V_2\}$ for entity sets E_1, E_2. So, the mapping implies $\pi_{\text{ID}}(E_2) \subseteq \pi_{\text{ID}}(E_1)$. If this condition is violated for some given E_1 and E_2, there exists no R that satisfies the mapping. Hence, the mapping does not roundtrip and it is impossible to generate query and update views that store and retrieve data losslessly. To validate that a mapping roundtrips we exploit the following theorem:

THEOREM 2 Let $map = f \circ g^{-1}$ be a bipartite mapping. Then the following conditions are equivalent:

1. $map \circ map^{-1} = Id(\mathcal{C})$
2. f is injective and $Range(f) \subseteq Range(g)$ □

That is, a mapping roundtrips if and only if f is injective and the range constraints of f (i.e., those inferred from f and the schema constraints in $\mathcal{C}$) imply the range constraints of g (i.e., those inferred from g and the schema constraints in $\mathcal{S}$). As shown in [28], the problem of computing the range of a mapping (or a view, as a special case) can be reduced to that of mapping composition. Composing mappings is very challenging [15, 4]. Due to space constraints, in this paper we only consider the first part of the mapping validation task, checking the injectivity condition. As we show next, this check can be done by exploiting the partitioning scheme.

5.3 General solution

In this section we describe our view generation approach in general terms and show that it solves the data roundtripping problem.

Let $\mathcal{P}$ be a schema containing a set of relations (partitions) and a set of constraints $\Sigma_{\mathcal{P}}$. $\mathcal{P}$ is a *partitioned schema* for $\mathcal{S}$ relative to a query language L if there exists a procedure that (i) allows rewriting each query $g \in L$ on $\mathcal{S}$ using a unique set of partitions in $\mathcal{P}$, (ii) each such rewriting is injective on the subschema $\mathcal{P}_{\text{exp}}$ formed by the partitions used in the rewriting and the respective constraints from $\Sigma_{\mathcal{P}}$, and (iii) the rewriting of the identity query on S uses all partitions in $\mathcal{P}$. (In Example 1, $\mathcal{P}_{\text{exp}}$ contains P_1^A, P_1^B, P_2^A and constraints $\pi_{\text{ID}}(P_1^A) = \pi_{\text{ID}}(P_1^B)$, $\pi_{\text{ID}}(P_1^A) \cap \pi_{\text{ID}}(P_2^A) = \emptyset$.)

By (ii) and (iii), there exist bijective views $r : \mathcal{P} \to \mathcal{S}$, $r \in L$ and $p : \mathcal{S} \to \mathcal{P}$ witnessing the equivalence of $\mathcal{P}$ and $\mathcal{S}$.

As shown in Figure 9, query g partitions $\mathcal{S}$ into $\mathcal{P} \subseteq \mathcal{P}_{\text{exp}} \times \mathcal{P}_{\text{unexp}}$ such that $p_{\text{exp}} : \mathcal{S} \to \mathcal{P}_{\text{exp}}$ and $p_{\text{unexp}} : \mathcal{S} \to \mathcal{P}_{\text{unexp}}$ are view complements [2] that together yield p. By condition (ii), g can be rewritten as $h \circ p_{\text{exp}}$ where h is injective. Let h' be an exact rewriting of the identity query on $\mathcal{P}_{\text{exp}}$ using h, i.e., h' reconstructs $\mathcal{P}_{\text{exp}}$ from $\mathcal{V}$. Then, the update view u and merge view m can be constructed as follows

$$u := f \circ h' \circ r[., \emptyset]$$
$$m(s_1, s_2) := r(p_{\text{exp}}(s_1), p_{\text{unexp}}(s_2))$$

where $\emptyset$ is the $\mathcal{P}_{\text{unexp}}$-state where all relations are empty, the view $r[., \emptyset]$ is such that $r[., \emptyset](x) = y$ iff $r(x, \emptyset) = y$, and $s_1, s_2 \in \mathcal{S}$.

Assuming that f is injective and $Range(f) \subseteq Range(g)$, it is easy to show that u and m satisfy the information-preservation condition of Section 4. The extended roundtripping criterion holds if we choose the view r in such a way that

$$\forall x \in \mathcal{P}_{\text{exp}}, \forall s \in \mathcal{S} : p_{\text{exp}}(r(x, p_{\text{unexp}}(s))) = x$$

To obtain such r, in Example 1 we left-outer-joined exposed partitions with unexposed partitions that agree on keys.

As we explain next, for our mapping and view language it is always possible to obtain the views h, h', p_{exp}, p_{unexp}, and r that satisfy the conditions stated above. Therefore, our solution is complete in that it allows constructing query, update, and merge views for each given valid mapping.

Due to conditions (i)–(iii), g is injective if and only if $\mathcal{P}_{\text{unexp}}$ is empty, i.e., has zero partitions. We exploit this property to check the injectivity of f required by Theorem 2. To do that, we swap the sides of the mapping such that f in the above construction takes the place of g, and apply the partitioning scheme to $\mathcal{C}$ in a symmetric fashion.

5.4 Compilation steps

In summary, mapping compilation comprises the following steps:

1. Subdivide the mapping into independent sets of fragments.
2. Perform partial mapping validation by checking the condition $Range(f) \subseteq Range(g)$ (using mapping composition techniques).
3. Apply the partitioning scheme to $\mathcal{C}$ and f.
4. If the $\mathcal{P}_{\text{unexp}}$ obtained above is non-empty, abort since f is not injective. Otherwise, produce f' as an exact rewriting of the identity query on $\mathcal{C}$.
5. Construct query view as $q = f' \circ g$.
6. Simplify q.
7. Apply the partitioning scheme to $\mathcal{S}$ and g ($\mathcal{P}_{\text{unexp}}$ produced here may be non-empty). Rewrite g as $h \circ p_{\text{exp}}$.
8. Produce h' as an exact rewriting of identity query on $\mathcal{P}_{\text{exp}}$.
9. Obtain u and m as shown in the previous section.
10. Simplify u and m.

Step 1 uses a divide-and-conquer method to scope the mapping compilation problem. Two mapping constraints are *dependent* if their fragment queries share a common extent symbol or some integrity constraint spans their extent symbols; in this case, they are placed into the same subset of constraints. All the remaining steps process one such subset of constraints at a time. Steps 3–6 produce query views, while Steps 7–10 produce update and merge views. Next we discuss the partitioning scheme, which is applied in Step 3 and Step 7 of mapping compilation.

5.5 Partitioning scheme

A partitioning scheme allows rewriting the mapping constraints using queries on partitions. Thus, its choice depends directly on the mapping language. Since the fragment queries used in our mapping language are join-free, the partitioning scheme can be applied to one extent at a time. Imagine that the data that belongs to a certain extent is laid out on a 2-dimensional grid along a 'horizontal' and a 'vertical' axis where each point on the vertical axis corresponds to a combination of (a) entity type, (b) complex types appearing in entities, (c) conditions on scalar properties, and (d) is null / is not null conditions on nullable properties; and each point on the hor-

izontal axis corresponds to a single direct or inherited attribute of an entity or complex type.

The partitioning scheme can produce a very large number of partitions if applied directly to the language L of fragment queries defined in Section 4.4. Therefore, we restrict L to the actual fragment queries appearing in a given mapping map. The partitioning along the vertical axis can be computed using the following recursive algorithm, which is invoked for each extent of type T_p. The elementary path to the extent is passed in p:

procedure PartitionVertically(p, T_p, map)
  $Part := \emptyset$ // start with an empty set of partitions
  for each type T that is derived from or equal to T_p **do**
    $P := \{\sigma_p \text{ IS OF (ONLY } T)\}$
    for each direct or inherited member A of T **do**
      if map contains a condition on $p.A$ **then**
        if $p.A$ is of primitive type **then**
          $P := P \times Dom(p.A, map)$
        else if $p.A$ is of complex type T_A **then**
          $P := P \times$ PartitionVertically($p.A, T_A, map$)
        end if
      end for
      $Part := Part \cup P$
    end for
  return $Part$

Each tuple of conditions added to $Part$ is interpreted as a conjunction. $Dom(p.A, map)$ denotes the domain of $p.A$ relative to the mapping map. That is, if $p.A$ is an enumerated or boolean domain, $Dom(p.A, map)$ contains conditions of the form $p.A = c$ for each value c from that domain, and the IS NULL condition if $p.A$ is nullable. Otherwise, $Dom(p.A, map)$ contains equality conditions on all constants appearing in conditions on $p.A$ in any mapping fragment in map, treating NOT NULL conditions as illustrated in the example below.

EXAMPLE 2 Suppose the mapping constraints contain conditions ($p = 1$) and (p IS NOT NULL) on path p of type integer. Then, $Dom(p, map) := \{\sigma_{cond_1}, \sigma_{cond_2}, \sigma_{cond_3}\}$ where

$$cond_1 := (p = 1)$$
$$cond_2 := (p \text{ IS NULL})$$
$$cond_3 := \text{NOT}(p = 1 \text{ OR } p \text{ IS NULL})$$

Every pair of conditions in $Dom(p, map)$ is mutually unsatisfiable. $\bigvee Dom(p, map)$ is a tautology. Notice that (p IS NOT NULL) is equivalent to ($cond_1$ OR $cond_3$). That is, selection $\sigma_{p \text{ IS NOT NULL}}(R)$ can be rewritten as a union query $\sigma_{cond_1}(R) \cup \sigma_{cond_3}(R)$. □

The following example illustrates how the partitioning algorithm works in presence of complex types:

EXAMPLE 3 Consider an entity schema shown on the left in Figure 7 where BillingAddr is a nullable property with complex type Address, and Address has a subtype USAddress. Then, the vertical partitioning algorithm produces the following partitions, which list all possible shapes of entities that can appear in the extent:

$P_1 : \sigma_{e \text{ IS OF (ONLY Person)}}$
$P_2 : \sigma_{e \text{ IS OF (ONLY Customer) AND } e.\text{BillingAddr IS NULL}}$
$P_3 : \sigma_{e \text{ IS OF (ONLY Customer) AND } e.\text{BillingAddr IS OF (ONLY Address)}}$
$P_4 : \sigma_{e \text{ IS OF (ONLY Customer) AND } e.\text{BillingAddr IS OF (ONLY USAddress)}}$
$P_5 : \sigma_{e \text{ IS OF (ONLY Employee)}}$ □

Horizontal partitioning is done by splitting each vertical partition in $Part$ according to the properties projected in mapping fragments in map. It is easy to show that the above partitioning scheme allows expressing each fragment query appearing in map as a union query over the produced partitions.

5.6 Reconstructing partitions from views

We move on to Step 4 and Step 8 of mapping compilation. Let $\mathbf{P}$ be the set of partitions constructed in the previous section for a given $\mathcal{C}$ or $\mathcal{S}$ extent. Let h be a view defined as $\mathbf{V} = (V_1, \ldots, V_n)$ where each fragment view V_i is expressed as a union query over partitions in $\mathbf{P}$. To simplify the notation, each view can be thought of as a set of partitions. Let $\mathbf{P}_{\text{exp}} = \bigcup \mathbf{V}$ be all partitions that are exposed in views in $\mathbf{V}$. Let the set of non-key attributes of each partition P and view V be denoted as $Attrs(P)$ and $Attrs(V)$, respectively.

If $\mathbf{P}_{\text{exp}} \neq \mathbf{P}$, then h is non-injective (as explained in Section 5.3). However, even if all partitions are used in $\mathbf{V}$, h may still be non-injective. Injectivity holds only if we can reconstruct each partition $P \in \mathbf{P}$ from the views $\mathbf{V}$. The algorithm that does that is presented below.

procedure RecoverPartitions($\mathbf{P}_{\text{exp}}, \mathbf{P}, \mathbf{V}$)
  Sort $\mathbf{V}$ by increasing number $|V|$ of partitions per view and
    by decreasing number $|Attrs(V)|$ of attributes per view
  for each partition $P \in \mathbf{P}_{\text{exp}}$ **do**
    $Pos := \emptyset$; $Neg := \emptyset$; // keeps intersected & subtracted views
    $Att := Attrs(P)$; // attributes still missing
    $PT := \mathbf{P}$; // keeps partitions disambiguated so far
    // Phase 1: intersect
    for ($i = 1$; $i \leq n$ **and** $|PT| > 1$ **and** $|Att| > 0$; i++) **do**
      if $P \in V_i$ **then**
        $Pos := Pos \cup V_i$; $PT := PT \cap V_i$
        $Att := Att - Attrs(V_i)$
      end if
    end for
    // Phase 2: subtract
    for ($i = n$; $i \geq 1$ **and** $|PT| > 1$; i--) **do**
      if $P \notin V_i$ **then**
        $Neg := Neg \cup V_i$; $PT := PT \cap V_i$
      end if
    end for
    if $|PT| = 1$ **and** $|Att| = 0$ **then**
      $Recovered[P] := (Pos, Neg)$
    end if
  end for
return

The algorithm takes as input the exposed partitions $\mathbf{P}_{\text{exp}}$ and the fragment views $\mathbf{V}$ and constructs an associative table $Recovered$ (at the bottom of the algorithm) that maps a partition P to a set of 'positive' views Pos and 'negative' views Neg. These views, if they exist, can be used to reconstruct P by joining all views in Pos and subtracting (using anti-semijoin) the views in Neg as $P = (\bowtie Pos) \,\overline{\ltimes}\, (\cup Neg)$.

The algorithm starts by ordering the views. This step is a heuristic for producing more compact expressions, which may use fewer views. Every set of rewritings produced by the algorithm is equivalent under each view ordering. The set of needed attributes Att represents a horizontal region, while the recovered partitions PT represents the vertical region of the view space. These regions need to be narrowed down to exactly P, for each $P \in \mathbf{P}_{\text{exp}}$. In Phase 1, we narrow the vertical region by intersecting the views that contain P, while keeping track of the attributes they provide. If joining the views is insufficient to disambiguate P tuples, then in Phase 2, we further narrow the vertical region using anti-semijoins with views that do not contain P. P is fully recovered if PT does not contain any tuples beyond those in P (i.e., $|PT| = 1$) and covers all the required attributes (i.e., $|Att| = 0$). Due to this condition, the algorithm is sound, i.e., each found rewriting is correct.

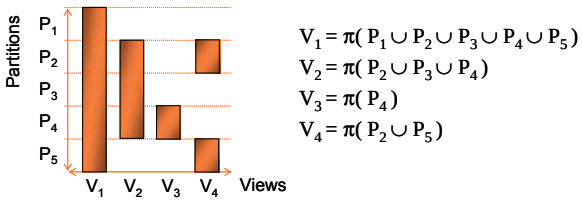

Figure 10: Partitions and views (based on Example 3)

EXAMPLE 4 Consider the partitioning scheme in Example 3. Let the rewritten fragment queries used in the mapping be expressed as shown in Figure 10 (for brevity, we finesse horizontal partitioning, represented by unqualified π). Sorting yields $\mathbf{V} = (V_3, V_4, V_2, V_1)$. Suppose that V_1 contains an attribute (e.g., Name) that is required by every partition. Then, the algorithm produces (up to projection – π is omitted for brevity):

$$P_1 = (V_1) \,\overline{\ltimes}\, (V_2 \cup V_4)$$
$$P_2 = (V_4 \bowtie V_2 \bowtie V_1)$$
$$P_3 = (V_2 \bowtie V_1) \,\overline{\ltimes}\, (V_4 \cup V_3)$$
$$P_4 = (V_3 \bowtie V_1)$$
$$P_5 = (V_4 \bowtie V_1) \,\overline{\ltimes}\, (V_2)$$

Expanding the view definitions shows that the above rewritings are implied by the expressions in Figure 10. To illustrate the effect of sorting, notice that view V_3 need not be used in the negative part of P_1, and V_2 does not appear in the positive part of P_4, in contrast to using the default sorting V_1, V_2, V_3, V_4. □

To establish the completeness of the algorithm, we need to show that it fails to recover P only if P cannot be reconstructed from $\mathbf{V}$. This result is based on the following theorem. For clarity of exposition, we speak of constants $\mathbf{P}$ and sets $\mathbf{V}$ while reusing the symbols for partitions and views.

THEOREM 3 Let p be a constant from $\mathbf{P}$, and $\mathbf{V} = \{V_1, \ldots, V_n\}$ be a set of subsets of $\mathbf{P}$. Then, there exists a relational expression $Expr(V_1, \ldots, V_n)$ such that $Expr = \{p\}$ if and only if

$$\bigcap\{V \mid p \in V, V \in \mathbf{V}\} - \bigcup\{V \mid p \notin V, V \in \mathbf{V}\} = \{p\} \quad \square$$

The above theorem proves the completeness of the algorithm, but does not guarantee that each found expression is minimal, i.e., uses the smallest possible number of views and/or operators. Finding a minimal solution is equivalent to solving the set cover problem, which is NP-complete. So, the algorithm implements a greedy approach which often produces near-optimal results. Due to sorting, the overall complexity of the algorithm is $O(n \log n)$, while Phases 1 and 2 are $O(n)$ in the number of fragment views.

5.7 Exploiting outer joins

The algorithm RecoverPartitions tells us how to reconstruct each partition P of a given extent. Ultimately, we need to produce a rewriting for the entire extent, which combines multiple partitions. Consider Examples 3 and 4. To obtain the query view for the Persons extent, we could union partitions $P_1, \ldots, P_5$ from Example 4. The resulting expression would use ten relational operators and be clearly suboptimal. Instead, we exploit the following observation. Since each fragment query produces a new view symbol V_i and is join-free, every V_i contains only partitions from a single extent. Therefore, each extent can be reconstructed by doing a full outer join of all the views that contain partitions from that extent. The full-outer-join expression can be further simplified using inclusion and disjointness constraints on the views.

EXAMPLE 5 Let Persons $= \tau(E)$ where τ is an expression that performs entity construction, fills in required constants, etc. (we talk about it shortly). Then, E can be obtained using the following expressions, all of which are equivalent under the view definitions of Figure 10:

$$E = P_1 \cup P_2 \cup P_3 \cup P_4 \cup P_5$$
$$= V_1 \,\overline{\bowtie}\, V_2 \,\overline{\bowtie}\, V_3 \,\overline{\bowtie}\, V_4$$
$$(*) \quad = (V_1 \,\overline{\bowtie}\, (V_2 \,\overline{\bowtie}\, V_3)) \,\overline{\bowtie}\, V_4$$
$$(**) \quad = ((V_1 \,\overline{\bowtie}\, V_2) \,\overline{\bowtie}\, (V_3 \cup^a V_4))$$

$\cup^a$ denotes union without duplicate elimination (UNION ALL). The equivalence can be shown by noticing that the following constraints hold on the views: $V_3 \cap V_4 = \emptyset$, $V_3 \subseteq V_2 \subseteq V_1$, $V_4 \subseteq V_1$. □

Using full outer joins allows us to construct an expression that is minimal with respect to the number of relational operators: n views are connected using the optimal number of $(n-1)$ binary operators. Still, as illustrated in Example 5, multiple minimal rewritings may exist. In fact, their number grows exponentially with n. As we illustrate in the experimental section, these rewritings may have quite different performance characteristics when they are used to execute queries on the database. One reason is the inability of the optimizer to push down selection predicates through outer joins.

To find a good rewriting that is likely to result in efficient execution plans, we use a heuristic that increases optimization opportunities by replacing outer joins by inner joins and unions. As a side effect, it helps avoid unnecessary joining of data that is known to be disjoint and minimize the size of intermediate query results.

The algorithm we use performs successive grouping of the initial full-outer-join expression using the inclusion and disjointness constraints between the views. The views are arranged into *groups*, which are relational expressions in the prefix notation. The constraints between groups are tested using set operations on the partitions they contain. Let $\rho(L)$ denote the set of partitions used in all groups and subgroups in L, where L is a sequence of groups. For example, consider a group $G = \overline{\bowtie}(V_2, V_3)$ containing two views from Figure 10. We have $\rho(G) = \{P_2, P_3, P_4\}$. The grouping algorithm is shown below.

procedure GroupViews($\mathbf{V}$)
  Arrange $\mathbf{V}$ into pairwise disjoint $\overline{\bowtie}$-groups $G_1, \ldots, G_m$
  such that each group contains views from a single source extent
  $G_\mathbf{V} := \overline{\bowtie}(G_1, \ldots, G_m)$
  for each group G in $G_\mathbf{V}$ **do recursively** depth-first
    if $G = (\overline{\bowtie} : \_, L_1, \_, L_2, \_)$ **and** $|L_1| > 0$ **and** $|L_2| > 0$ **then**
      $G_1 = (\overline{\bowtie} : L_1)$; $G_2 = (\overline{\bowtie} : L_2)$
      if $\rho(L_1) \cap \rho(L_2) = \emptyset$ **then** $G = \overline{\bowtie}(\cup^a(G_1, G_2), \_)$
      else if $\rho(L_1) = \rho(L_2)$ **then** $G = \overline{\bowtie}(\bowtie(G_1, G_2), \_)$
      else if $\rho(L_1) \supseteq \rho(L_2)$ **then** $G = \overline{\bowtie}(\overline{\bowtie}(G_1, G_2), \_)$
      else if $\rho(L_1) \subseteq \rho(L_2)$ **then** $G = \overline{\bowtie}(\overline{\bowtie}(G_2, G_1), \_)$
      end if
    end if
  end for
  Flatten singleton groups in $G_\mathbf{V}$
return $G_\mathbf{V}$

The grouping is performed by replacing $\overline{\bowtie}$ by $\cup^a$, $\bowtie$, and $\overline{\bowtie}$, in that order, where possible. Using $\cup^a$ (UNION ALL) is critical to avoid the sorting overhead upon query execution. The above algorithm does not prescribe the choice of L_1 and L_2 uniquely. In our implementation, a greedy approach is used to find 'large' L_1 and L_2 in linear time, yielding the overall $O(n)$ complexity.

We defer the explanation of the very first step of the algorithm, the initial grouping by source extents, to Section 5.9.

5.8 Producing CASE statements

In Example 5, we used the expression τ to encapsulate constant creation and non-relational operators that reside on top of a relational expression. In this section we explain how τ is obtained. Consider the partitions shown in Example 3. Upon assembling the Persons extent from $P_1, \ldots, P_5$ we need to instantiate Persons, Customers, or Employees depending on the partition from which the tuples originate. Thus, all tuples coming from P_1 yield Person instances, while all tuples coming from P_2 or P_3 or P_4 produce Customer instances. Furthermore, for all tuples in P_2, the BillingAddr property needs to be set to NULL.

Let the boolean variables $b_{P1}, \ldots, b_{P5}$ keep track of the tuple provenance. That is, b_{P1} is true if and only if the tuple comes from P_1, etc. Then, we can determine what instances and constants need to be plugged into τ using a disjunction of b_P variables. This fact is not incidental: it is due to using the partitioning scheme that allows us to rewrite all fragment queries as unions of partitions. Continuing with Example 3, we obtain Persons as

```
CASE WHEN b_P1 THEN Person(...)
     WHEN (b_P2 OR b_P3 OR b_P4) THEN Customer(...,
          CASE WHEN b_P2 THEN NULL
               WHEN b_P3 THEN Address(...)
               WHEN b_P4 THEN USAddress(...)
               END AS BillingAddr, ... )
     WHEN b_P5 THEN Employee(...)     (E)
```

where E is the relational expression whose construction we discussed in the previous section. By exploiting outer joins we obtained a compact $E = ((V_1 \sqsupset\!\bowtie V_2) \sqsupset\!\bowtie (V_3 \cup^a V_4))$. However, now we need to do some extra work to determine the boolean variables $b_{P1}, \ldots, b_{P5}$ from E. That is, the problem is to rewrite b_P variables in terms of b_V variables, where b_{V_i} is true if and only if the tuple originates from V_i.

The information needed for this rewriting is delivered by the RecoverPartitions algorithm of Section 5.6. The sets Pos and Neg of intersected and subtracted views directly translate into a boolean conjunction of positive and negated boolean literals. Hence, continuing Example 4 we obtain

$$
\begin{aligned}
b_{P_1} &= b_{V_1} \wedge \neg b_{V_2} \wedge \neg b_{V_4} \\
b_{P_2} &= b_{V_4} \wedge b_{V_2} \wedge b_{V_1} \\
b_{P_3} &= b_{V_2} \wedge b_{V_1} \wedge \neg b_{V_4} \wedge \neg b_{V_3} \\
b_{P_4} &= b_{V_3} \wedge b_{V_1} \\
b_{P_5} &= b_{V_4} \wedge b_{V_1} \wedge \neg b_{V_2}
\end{aligned}
$$

The b_P variables in the case statement can now be replaced by their b_V rewritings. The disjunction $(b_{P_2} \vee b_{P_3} \vee b_{P_4})$ can be further compacted using standard boolean optimization techniques. Notice that by exploiting the constraints in Figure 10 we could obtain an even more optimal rewriting $(b_{P_2} \vee b_{P_3} \vee b_{P_4}) = b_{V_2}$, by extending the RecoverPartitions algorithm to consider unions of partitions.

5.9 Eliminating self-joins and self-unions

We discuss the final simplification phase of mapping compilation, performed in Step 6 and Step 10.

EXAMPLE 6 Consider constructing query views in Example 5. In Step 5 we expand the view symbols V_i by fragment queries on relational tables. Suppose the fragment queries for V_2 and V_3 are on the same table, e.g., $V_2 = R$ and $V_3 = \sigma_{A=3}(R)$. If we choose the rewriting $(*)$, then $V_2 \sqsupset\!\bowtie V_3$ becomes a redundant self-join and can be replaced by R. We obtain:

$$E = (V_1 \sqsupset\!\bowtie R) \sqsupset\!\bowtie V_4$$

In contrast, choosing the rewriting $(**)$ gives us

$$E = ((V_1 \sqsupset\!\bowtie R) \sqsupset\!\bowtie (\sigma_{A=3}(R) \cup V_4)) \quad \square$$

To bias the algorithm GroupViews toward the top expression above, in its first step we group the fragment views according to their *source* extents. For query views, source extents are tables; for update views, source extents are those in the entity schema. The main loop of GroupViews preserves this initial grouping.

Eliminating the redundant operations poses an extra complication. As explained in the previous section, we need to keep track of tuple provenance. So, when self-joins and self-unions get collapsed, the boolean variables b_V need to be adjusted accordingly. These boolean variables are initialized in the leaves of the query tree by replacing each V_i by $V_i \times \{\text{True AS } b_{V_i}\}$. In Entity SQL, boolean expressions are first-class citizens and can be returned by queries (see Figure 4). To preserve their value upon collapsing of redundant operators, we use several rewriting rules, one for each operator. Let b_1 and b_2 be computed boolean terms, $\mathbf{A}$ and $\mathbf{B}$ be disjoint lists of attributes, c_1 and c_2 be boolean conditions, and let

$$
\begin{aligned}
E_1 &= \pi_{b_1,\mathbf{A},\mathbf{B}}(\sigma_{c_1}(E)) \\
E_2 &= \pi_{b_2,\mathbf{A},\mathbf{C}}(\sigma_{c_2}(E))
\end{aligned}
$$

Then, the following equivalences hold:

$$
\begin{aligned}
E_1 \bowtie_{\mathbf{A}} E_2 &= \pi_{b_1,b_2,\mathbf{A},\mathbf{B},\mathbf{C}}(\sigma_{c_1 \wedge c_2}(E)) \\
E_1 \sqsupset\!\bowtie_{\mathbf{A}} E_2 &= \pi_{b_1,(b_2 \wedge c_2),\mathbf{A},\mathbf{B},\mathbf{C}}(\sigma_{c_1}(E)) \\
E_1 \sqsupset\!\bowtie\!\sqsubset_{\mathbf{A}} E_2 &= \pi_{(b_1 \wedge c_1),(b_2 \wedge c_2),\mathbf{A},\mathbf{B},\mathbf{C}}(\sigma_{c_1 \vee c_2}(E))
\end{aligned}
$$

Observe that if $\pi_{\mathbf{A}}(E_1) \cap \pi_{\mathbf{A}}(E_2) = \emptyset$, then $E_1 \cup^a E_2 = E_1 \sqsupset\!\bowtie\!\sqsubset_{\mathbf{A}} E_2$. The disjointness holds for $\cup^a$ introduced by the GroupViews algorithm because it requires the unioned expressions to have non-overlapping keys. Therefore, for $\cup^a$ we can use the same rule as for $\sqsupset\!\bowtie\!\sqsubset$.

The presented rules assume two-valued boolean logic, where outer joins produce False values instead of NULLs for boolean terms. However, like standard SQL, Entity SQL uses three-valued boolean logic (True, False, NULL). To compensate for it, we translate nullable boolean expressions into non-nullable ones that replace NULLs by False values. This can be done by wrapping a nullable boolean expression b as (b AND (b IS NOT NULL)) or as (CASE WHEN b THEN True ELSE False END).

The final query view produced by the mapping compiler for the scenario of Figure 7 is shown in [5] (there BillingAddr is assumed to be atomic). It illustrates the compound effect of most of the presented techniques.

6. EVALUATION

The goal of a client-side mapping layer is to boost the developer's productivity while offering performance comparable to a lower-level solution that uses handcoded SQL. In this section we discuss the experimental evaluation of the Entity Framework, focusing on the mapping compiler.

Correctness. The top priority for the mapping compiler is correctness. It needs to produce bidirectional views that roundtrip data, otherwise data loss or corruption is possible. In practice, proving correctness of a complex system on paper is insufficient, since errors unavoidably creep into the implementation. We followed two paths. First, our product test team developed an automated suite that generates thousands of mappings by varying some core scenarios. The compiled views are verified by deploying the entire data access stack to query and update pre-generated sample databases. Although this does not guarantee complete coverage, it

exercises many other parts of the system (e.g., client-side query optimization) in addition to the mapping compiler. The second evaluation technique was developed by our colleagues in the software verification group at Microsoft Research: a tool that translates Entity SQL queries into first-order logic formulas and feeds them into a theorem prover. As we explained in Section 4, testing roundtripping of views expressed in Entity SQL is undecidable, so the prover may not terminate. For the positive cases, it has been able to verify all views tested so far.

Efficiency. Another aspect that we studied is the efficiency of mapping compilation. Step 2 of the main algorithm is the only compilation step that requires exponential time. Steps 3–10 are $O(n \log n)$. Subdividing the mapping constraints in Step 1 usually yields a manageable number of fragments n. So far, we have not seen cases where n exceeded a few dozen. Compiling mappings prior to query execution was a fundamental design choice, which alleviates performance issues; it eliminates the runtime penalty of teasing out the data transformation logic defined in a mapping. In our initial design, the compiler was invoked upon the first query or update issued by the application, yielding a one-time performance hit per lifetime of an application. This few-second delay was deemed unacceptable. One reason is that applications are rerun frequently during development. Another is that mid-tier applications often shut down and restart for load balancing. Therefore, we decided to factor out the compiler execution from the runtime and integrate it with the development environment (IDE). This approach may enable using exhaustive, exponential-time techniques in place of the currently deployed heuristics, where view generation runs as a compile-time background job and recompiles only those parts of the mapping that have been modified by the developer.

Performance. Mapping compilation plays a key role both in the client-side query rewriting and server-side execution. In the algorithms we presented, we made heavy use of implied constraints to simplify the generated views. To illustrate these benefits qualitatively, consider the following experiment run on the AdventureWorks sample database shipped with Microsoft SQL Server. We partitioned the Sales table horizontally into H1 and H2, and vertically into V1 and V2. This produces two mapping scenarios, where the sales data needs to be reassembled from the respective tables. Leveraging the mapping constraints allows us to rewrite $\sqsupset\bowtie\sqsubset$ as $\cup^a$ (exploiting the disjointness of H1 and H2) and as $\bowtie$ (knowing that V1 and V2 agree on keys). Suppose that the client issues selection queries as shown in Figure 11. If the query views contain outer joins, the SQL optimizer is not able to push the selections down, resulting in suboptimal query plans that take many times longer to execute. This effect multiplies in more complex scenarios.

In addition to the server load, another important performance metric is the total overhead of the client-side layer. The major factors contributing to this overhead are object instantiation, caching, query manipulation, and delta computation for updates. For small data sets and OLTP-style queries, these factors may dominate the overall execution time. Therefore judicious optimization of every code path is critical.

Performance analysts in our product group conducted an extensive study comparing the overall system performance with custom implementations and competing products. The study found that the views produced by the mapping compiler are close to those written by hand by experienced database developers. Details of this study are beyond the scope of this paper. The generated views enable the query and update pipelines to produce query and update statements whose server-side execution performance approaches that of a custom implementation. Some mapping scenarios supported by the compiler, in particular arbitrary combinations or vertical and horizontal partitioning in inheritance mappings, go beyond those supported in other commercial systems.

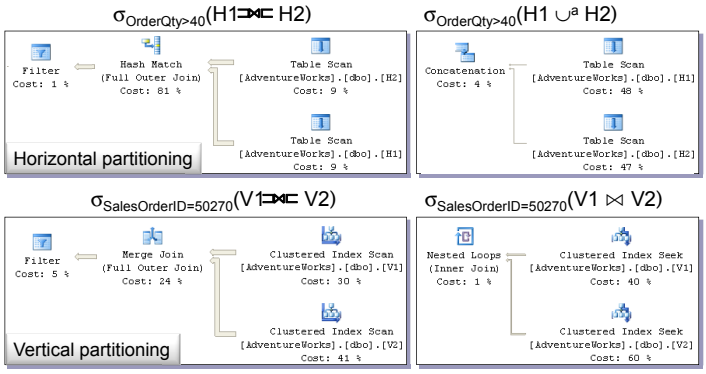

Figure 11: Exploiting mapping constraints to speed up queries

7. RELATED WORK

Transforming data between database and application representations remains a hard problem. Researchers and practitioners attacked it in a number of ways. A checkpoint of these efforts was presented by Carey and DeWitt in 1996 [10]. They outlined why many such attempts, including object-oriented databases and persistent programming languages, did not pan out. They speculated that object-relational databases would dominate in 2006. Indeed, many of today's database systems include a built-in object layer that uses a hardwired O/R mapping on top of a conventional relational engine [9, 25]. However, the O/R features offered by these systems appear to be rarely used for storing enterprise data, with the exception of multimedia and spatial data types [19]. Among the reasons are data and vendor independence, the cost of migrating legacy databases, scale-out difficulties when business logic runs inside the database instead of the middle tier, and insufficient integration with programming languages [30].

Database research has contributed many powerful techniques that can be leveraged for supporting mapping-driven data access. And yet, there are significant gaps. Among the most critical ones is supporting updates through mappings. Compared to queries, updates are far more difficult to deal with as they need to preserve data consistency across mappings, may trigger business rules, and so on. Updates through database views have a long history. In 1978, Dayal and Bernstein [13] put forth the view update problem. They observed that finding a unique update translation even for very simple views is rarely possible due to the intrinsic underspecification of the update behavior by a view. As a consequence, commercial database systems offer very limited support for updatable views.

Subsequent research followed two major directions. One line of work focused on determining under what circumstances view updates can be translated unambiguously, more recently for XML views [8]. Unfortunately, there are few such cases; besides, usually every update needs to have a well-defined translation, i.e., rejecting a valid update is unacceptable. Furthermore, in mapping-driven scenarios the updatability requirement goes beyond a single view. For example, an application that manipulates Customer and Order entities effectively performs operations against two views. Sometimes a consistent application state can only be achieved by updating several views simultaneously. Another line of research focused on closing the underspecification gap. Several mechanisms were suggested in the literature, such as constant complement [2] or dynamic views [17]. So far, this work has turned out to be mostly

of theoretical value, although techniques of [3, 23] appear to have had commercial impact. Our computation of merge views exploits view complements.

Recently, researchers have shown rekindled interest in the view update problem. Pierce et al. developed a bidirectional mechanism called 'lenses' [7, 16] which uses *get* and *putback* functions. Our query views correspond to *get*, while update and merge views combined define *putback*. One of our key distinguishing contributions is a technique for propagating updates *incrementally* using view maintenance [6, 20], which is critical for practical deployment. Another one is a fundamentally different mechanism for obtaining the views, by compiling them from mappings, which allows describing complex mapping scenarios in an elegant way. Also, in our approach data reshaping (specified by query and update views) is separated from the update policy (merge views). Further recent work on view updates is [24], where delta tuples are stored in auxiliary tables if the updates cannot be applied to the underlying database.

Mapping compilation was explored in IBM's Clio project, which introduced the problem of generating data transformations from correspondences between schema elements [27]. Our mapping compilation procedure draws on answering queries using views for exact rewritings (surveyed in [21] and examined recently in [18, 29]), and exploits several novel aspects such as the partitioning scheme, bipartite mappings, and rewriting queries under bidirectional mapping constraints. We are not aware of published work that exploited outer joins and case statements as we did to optimize the generated views. Other related mapping manipulation problems are surveyed in [5].

8. CONCLUSIONS

Bridging applications and databases is a fundamental data management problem. We presented a novel mapping approach that supports queries and updates through mappings in a principled way. We formulated the mapping compilation problem and described our solution. To the best of our knowledge, the ADO.NET Entity Framework is the first commercial data access product that is driven by declarative bidirectional mappings with well-defined semantics.

The new mapping architecture exposed many interesting research challenges. In addition to mapping compilation, these challenges include enforcing data consistency using a combination of client-side and server-side constraints, exploiting efficient view maintenance techniques for object-at-a-time and set-based updates, translating errors through mappings, optimistic concurrency, notifications, and many others. We plan to report on them in the future.

9. ACKNOWLEDGEMENTS

We are grateful to the members of the Microsoft SQL Server product team for their support and engagement in developing the ideas presented in this paper. Special thanks are due to José Blakeley, S. Muralidhar, Jason Wilcox, Sam Druker, Colin Meek, Srikanth Mandadi, Shyam Pather, Tim Mallalieu, Pablo Castro, Kawarjit Bedi, Mike Pizzo, Nick Kline, Ed Triou, Anil Nori, and Dave Campbell. We thank Paul Larson, Yannis Katsis, and Renée Miller for helpful discussions and comments on an earlier version of this paper.

10. REFERENCES

[1] A. Adya, J. A. Blakeley, S. Melnik, S. Muralidhar, The ADO.NET Team. Anatomy of the ADO.NET Entity Framework. In *SIGMOD*, 2007.

[2] F. Bancilhon, N. Spyratos. Update Semantics of Relational Views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[3] T. Barsalou, A. M. Keller, N. Siambela, G. Wiederhold. Updating Relational Databases through Object-Based Views. In *SIGMOD*, 1991.

[4] P. A. Bernstein, T. J. Green, S. Melnik, A. Nash. Implementing Mapping Composition. In *VLDB*, 2006.

[5] P. A. Bernstein, S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*, 2007.

[6] J. A. Blakeley, P.-Å. Larson, F. W. Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, 1986.

[7] A. Bohannon, B. C. Pierce, J. A. Vaughan. Relational Lenses: A Language for Updatable Views. In *PODS*, 2006.

[8] V. P. Braganholo, S. B. Davidson, C. A. Heuser. From XML View Updates to Relational View Updates: Old Solutions to a New Problem. In *VLDB*, 2004.

[9] M. J. Carey, D. D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, N. M. Mattos. O-O, What Have They Done to DB2? In *VLDB*, 1999.

[10] M. J. Carey, D. J. DeWitt. Of Objects and Databases: A Decade of Turmoil. In *VLDB*. Morgan Kaufmann, 1996.

[11] P. Castro, S. Melnik, A. Adya. ADO.NET Entity Framework: Raising the Level of Abstraction in Data Programming. In *SIGMOD (demo)*, 2007.

[12] W. R. Cook, A. H. Ibrahim. Integrating Programming Languages and Databases: What is the Problem? ODBMS.ORG, Expert Article, Sept. 2006.

[13] U. Dayal, P. A. Bernstein. On the Updatability of Relational Views. In *VLDB*, 1978.

[14] R. A. Di Paola. The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas. *J. ACM*, 16(2):324–327, 1969.

[15] R. Fagin, P. G. Kolaitis, L. Popa, W. C. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.

[16] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt. Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. In *POPL*, 2005.

[17] G. Gottlob, P. Paolini, R. Zicari. Properties and Update Semantics of Consistent Views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.

[18] G. Gou, M. Kormilitsin, R. Chirkova. Query Evaluation Using Overlapping Views: Completeness and Efficiency. In *SIGMOD*, 2006.

[19] S. Grimes. Object/Relational Reality Check. *Database Programming & Design (DBPD)*, 11(7), July 1998.

[20] A. Gupta, I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[21] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.

[22] C. Keene. Data Services for Next-Generation SOAs. *SOA WebServices Journal*, 4(12), 2004.

[23] A. M. Keller, R. Jensen, S. Agrawal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. In *SIGMOD*, 1993.

[24] Y. Kotidis, D. Srivastava, Y. Velegrakis. Updates Through Views: A New Hope. In *ICDE*, 2006.

[25] V. Krishnamurthy, S. Banerjee, A. Nori. Bringing Object-Relational Technology to Mainstream. In *SIGMOD*, 1999.

[26] E. Meijer, B. Beckman, G. M. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD*, 2006.

[27] R. J. Miller, L. M. Haas, M. A. Hernández. Schema Mapping as Query Discovery. In *VLDB*, 2000.

[28] A. Nash, P. A. Bernstein, S. Melnik. Composition of Mappings Given by Embedded Dependencies. In *PODS*, 2005.

[29] L. Segoufin, V. Vianu. Views and Queries: Determinacy and Rewriting. In *PODS*, 2005.

[30] W. Zhang, N. Ritter. The Real Benefits of Object-Relational DB-Technology for Object-Oriented Software Development. In *BNCOD*, 2001.