Bidirectional by Necessity: Data Persistence and Adaptability for Evolving Application Development

James F. Terwilliger

Microsoft Corporation

Abstract. Database-backed applications are ubiquitous. They have common requirements for data access, including a bidirectional requirement that the application and database must have schemas and instances that are synchronized with respect to the mapping between them. That synchronization must hold under both data updates (when an application is used) and schema evolution (when an application is versioned). The application developer treats the collection of structures and constraints on application data — collectively called a *virtual database* — as indistinguishable from a persistent database. To have such indistinguishability, that virtual database must be mapped to a persistent database by some means. Most application developers resort to constructing such a mapping from custom-built middleware because available solutions are unable to embody all of the necessary capabilities. This paper returns to first principles of database application development and virtual databases. It introduces a tool called a *channel*, comprised of incremental atomic transformations with known and provable bidirectional properties, that supports the implementation of virtual databases. It uses channels to illustrate how to provide a singular mapping solution that meets all of the outlined requirements for an example application.

1 Introduction

The persistent data application is a staple of software development. A client application written in some programming language (usually object-oriented in contemporary systems) presents data to a user, who may update that data as well. That data is stored in a relational database, whose schema may be designed entirely independently from the application. This paradigm is ubiquitous at this point, and as such, ample software tooling support has been developed to support it. There are graphical designers for the user interface, model builders to design workflow or class diagrams, integrated development environments for the design of the code, and data persistence tools to handle the retrieval and updating of data between the application and the database.

An application typically has some local understanding of its data that conforms to a schema. That schema may be explicit in the form of a local data cache, or it may be implicitly present in the form of a programming interface. That schema likely also has constraints over its data in terms of valid data states, as well as some form of referential integrity between entities. In short, an application's local data schema has many of the same conceptual trappings as the schema of a relational database. Thus, one can consider the local schema to be a *virtual database*.

The structure and semantics of that virtual database may differ significantly from the actual schema of the data where it is stored. Despite these differences, the application

R. Lämmel, J. Saraiva, and J. Visser (Eds.): GTTSE 2011, LNCS 7680, pp. 219–270, 2013.

developer has certain expectations of data as it moves to and from persistence. For instance, when the application constructs data, it assumes that it will be able to retrieve it again, and in the same form as it was created. In short, the designer of an application expects to be able to treat a virtual database as if it were indistinguishable from a real database with persistence. Tools such as query-defined views, object-relational mappers (e.g., [28,31,34]), extract-transform-load scripts (e.g., [42]), and research prototypes (e.g., [8,9]) fulfill some but not all of the requirements of a virtual database. For instance, query-defined views provide a simple way to transform data in a database to match an application's schema, but only provide limited support for update and no support for relationships expressed between views. As a result, the typical application developer combines multiple data access components bound together with additional program code, with no assurances for correctness and an unknown amount of maintenance cost. Most data access frameworks have minimal support for the evolution of an application's schema over multiple versions, resulting in additional manual maintenance and testing cost.

This paper views the problem of persistent data in applications and virtual databases from first principles: It first looks at the requirements of application development and execution, then develops one possible tool — called a *channel* — capable of fulfilling those requirements.

1.1 Scenario Requirements

The development and usage lifecycles of the database-backed application induce a number of requirements that any data access solution — or whatever collection of data access solutions and manual processes are employed by an application — must fulfill. The requirements come about from how an application is designed initially, how it is used in a production environment, and how it is maintained over time across versions.

One-Way Roundtripping. When the user inputs data into the application, or the user sees a complete entity in the view of the application, the user expects that data to be unchanged when persisted to the database and retrieved again. The same property can also be true in the opposite direction, where data in the database must be unchanged by a trip to and from the application, but there are a variety of reasons why that property need not be respected. A prime example of requiring that database-centered roundtripping need not be respected is the situation where accessing the database is recorded for security purposes. In this case, a set of operations that have the aggregate effect of leaving the application model unchanged will in fact change the database state.

Object-Relational Mapping. A data persistence tool must provide a solution to the impedance mismatch. Object-centered constructs like class hierarchies and collection-valued properties must have a mapping to relational storage.

Relational-Relational Mapping. The application and storage schemas may have been designed independently and thus have arbitrarily different structures. Once the impedance mismatch has been overcome, the persistence software must be able to accommodate the remaining differences between relational schemas. For instance, an

application's database tables may be represented as key-attribute-value triples because the number of attributes per table would be too large otherwise.

Business Logic. The relationship between schemas may exist to simply restructure data into a different form without altering information capacity. However, it may also include business-specific rules. For instance, an application may require that no data ever be deleted from the database but rather be "deprecated" to maintain audit trails.

So-called CRUD Operations. Applications require the ability to **C**reate a new entity, **R**etrieve an individual entity based on key value, **U**pdate an entity's properties, and **D**elete a given entity.

Bonus: Arbitrary Query and Set-Based Update. Many applications, though not all, require the ability to perform arbitrary queries against their application schema. Other applications may require the ability to update or delete entities based on arbitrary conditions instead of key values. These features are sometimes not required by the user interface of the application, but rather by some internal processing or workflow within the application.

Bonus: Evolution of the Application Schema. Different versions of an application will likely have different models of data to varying degrees. As that data schema evolves, the schema and instances of the persistent data store, as well as the mapping between the two schemas, must evolve with it. Most data access frameworks do not account for such evolution automatically and must be edited manually, but such evolution is the natural byproduct of application versioning.

1.2 The Status Quo

Database virtualization mechanisms present a perspective on persistent data that is different from the actual physical structures to match the model an application presents to a user. Virtualization can mask certain data for security, simplify structure for simpler querying, allow existing programs to operate over a revised physical structure, and so forth. Various virtualization mechanisms for databases have been proposed over the decades, and are in use in production environments; the most well-known is relational views, expressed as named relational queries.

Query-defined views, e.g., as specified in SQL, are highly expressive — especially for read-only views — and offer an elegant implementation that can leverage the optimizer in the DBMS. But query-defined views fall short of the above requirements for several reasons. First, while the view update problem has been well studied, there is no support for expressing schema modifications (Data Definition Language statements, or DDL), including key and foreign key constraints, against a view. If an application's demands on its view schema change, the developer has no recourse but to manually edit the physical schema and mapping.

Second, even if DBMSs in common use supported the full view update capability described in the research literature (e.g., [4,10,16,22]), database applications would still require more. The relationship between an application's view and physical schemas may require discriminated union, value transformation according to functions or lookup

tables, or translation between data and metadata, as in the case where persistence is in the form of key-attribute-value triples. Business logic like the "data deprecating" example above are not handled either. None of these transformations are supported by updatable views as currently implemented by database systems; the final two are not considered by research literature; and deprecation is not expressible in SQL without stored procedures, triggers, or advanced features like temporal capabilities (e.g., [24]).

Because there are not yet tools that support true virtual databases, applications requiring anything more than a trivial mapping to storage often have custom-crafted solutions built from SQL, triggers, and program code. This approach is maximally expressive, using a general-purpose language, but presents an interface of one or more read/update routines with pre-defined queries — far from being indistinguishable from a real database. A programming language-based approach is not well-suited for declarative specification, analysis, simplification, and optimization of the virtualization mapping. Thus, there is essentially no opportunity to formally reason about the properties of a database virtualization expressed in middleware, in particular, to prove that information is preserved through the virtualization.

1.3 Related Tools with Different Requirements

There are other applications for relational-to-relational mappings that are not directly related to application development and have different requirements. For instance, a federated database system may use relational mappings to connect constituent data sources to a single integrated system (e.g., [26]). In such a system, if you consider the collective set of data sources as a single integrated source, the mapping still needs to support query operations, but not necessarily update operations. They need not handle business logic. Some federated languages like Both-As-View (BAV) can also support schema evolution of the federated schema in a semi-automated fashion.

Another related technology is data exchange, where a mapping is used to describe how to translate data from one schema to another. In this scenario, the mapping may be lossy in both directions, since the schemas on either end of the mapping need not have been developed with any common requirements [2]. In this scenario, most of the assumptions about requirements from the first section are inapplicable. For instance, queries against the target schema may be incomplete due to incomplete mapping information. Instance updates are not a requirement, though effort has been made to investigate scenarios where mappings may be invertible with varying degrees of loss. Schema evolution of either schema has been considered, but evolutions are allowed to be lossy, as they do not propagate to the partner schema.

1.4 Overview

The narrative thread of this paper follows a simple theme: devising a complete solution to a problem. Section 2 will introduce an example of an application model and a database, both derived from a real-world application. The majority of the paper addresses the technical challenges posed by that application. Finally, Section 8 describes how to implement the application using the created tools.

Section 3 explores further the notion of a virtual database and one possible tool to implement database virtualization called a *channel*. Section 4 continues the discussion of channels by defining channel transformations that operate over relational data only, and gives fleshed-out examples of two such transformations. Section 5 introduces the notion of a channel transformation whose behavior may be non-deterministic. Section 6 defines a channel transformation that maps object-oriented, hierarchical data to relational data. Finally, Section 9 gives some insights on further reading on channels, related work, and future directions for investigation.

Some of the work presented in this paper has been drawn from two conference papers: one specifically on relational mappings [41], and one on object-relational mappings [37]. Chapter 5 has not yet appeared in any publication (outside of dissertation work [36]). The relational work was done as part of a larger context of graphical application development, where the schema for an application is derived from its user interface so that alternative artifacts such as query interfaces may be automatically generated from the user interface as well. Additional work has been published on the general framework for user interface development [39] as well as giving the user the ability to create permanent, redistributable anchors to data as displayed in a user interface [40]. The ability to use an application interface and schema as a first-class entity is enabled by database virtualization, and channels are one way to effect that property.

2 Example Scenario

To motivate the discussion, consider an application whose object-oriented model of data is pictured in Figure 1. This sample application is a reduction from a real-world application written in the field of clinical endoscopy for the purpose of maintaining an electronic medical record. The primary data to be tracked in the application is clinical procedure data. In the full application, procedure data is tracked from pre-procedure through post-procedure, including the state of the patient prior to the procedure, pain management regimens, the length of the procedure, the treatments and therapies applied, and post-operative instructions, care, and outcomes. In Figure 1, a simplified form of the data to be tracked can be found in the `Procedure` class hierarchy. Note that in the real application, the number of attributes per procedure numbers in the range of several hundred.

There are also relationships established between the procedures and the people involved, both the patients and the clinical staff performing the procedure or doing pre- or post-procedure assessments. The people involved are represented in the `Person` hierarchy, and the relationships between procedures and people are represented by the foreign keys in the diagram, shown as dotted lines. For instance, since there is exactly one patient associated with a procedure, but a given patient may undergo many procedures, the relationship between patients and procedures is a one-to-many relationship. In the figure, “Navigation Properties” are merely shortcuts that allow object-oriented languages a means to traverse relationships, so for instance to find the patient associated with a procedure, one need only access the `Patient` property of a procedure.

Figure 2 shows the schema for the relational storage for the application schema in Figure 1. The relational schema in the figure is also a simplification of the actual relational schema used in the clinical application. The class hierarchy `Person` in Figure 1

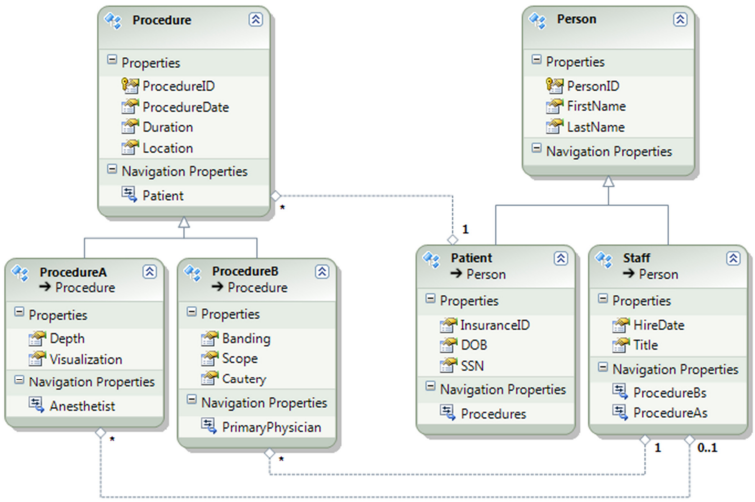


Fig. 1. The object-oriented schema of the example application

maps to the two tables `Patient` and `Staff` in Figure 2 (this mapping style is often called "table per concrete class"). The mapping between the `Procedure` hierarchy and its tables is much less straightforward, but encompasses the tables `Procedure`, `TextData`, and `NumericData`.

To illustrate how the mapping works for procedure data, consider an example instance as shown in Figure 3. The patient object and the staff member object in that example (Figure 3(a)) map cleanly to individual rows in relations (Figure 3(b)). The procedure, however, is split across three tables: one to hold the "basic" information about the procedure, including the kind of the procedure (also known as a *discriminator* column), one to hold the remaining text-valued attributes of the procedure, and one to hold the remaining number-valued procedure attributes. The text and number value tables have been "pivoted"; each row in the table corresponds to an attribute in the original procedure.

Also, for each text or number attribute, three additional columns have been added: a start time (`VTStart`, or Valid Time Start), an end time (`VTEnd`, or Valid Time End), and an indicator of what user last updated that row. These three columns are the result of business logic in the application. Every time one of these attributes is edited in the application, the old value of the attribute is still kept around, but the "end time" is set, in essence deprecating the value. Then, the new updated value is added as a new row. In Figure 3(b), one can see that user "travis" has updated two of the values in the original procedure to become the values as seen in the current state of the objects in Figure 3(a).

In the original application upon which this example is based, the mapping between the two schemas comprises several different technologies:

- Stored procedures to handle business logic for updating rows in the manner described in Figure 3, and to return only non-deprecated data

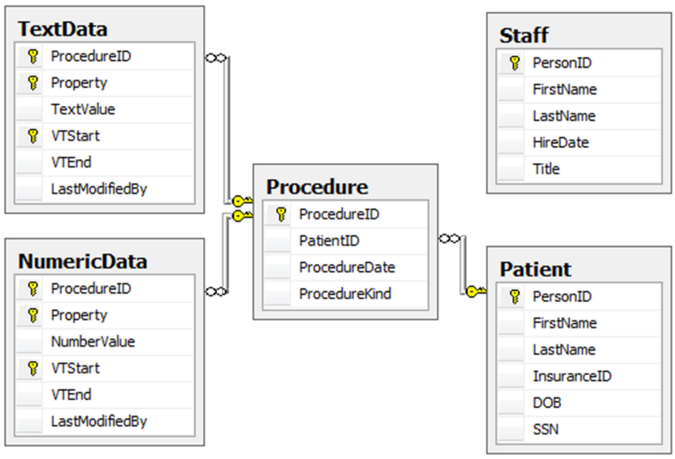
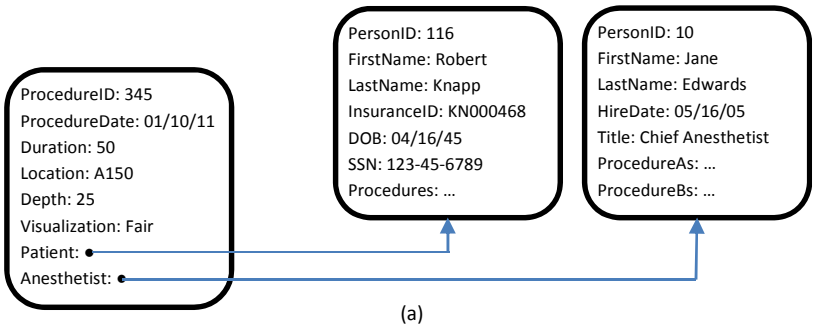


Fig. 2. The relational schema of the example application, serving as the persistent storage for the schema in Figure 1

- In-memory programming logic to handle the re-assembly of procedure data into objects, and also to break a procedure apart into the various tables when writing to the database
- Manual editing effort when evolving the application to add new properties to an existing procedure kind, or to add another procedure kind altogether
- Extract-Transform-Load (ETL) scripts for translating the data into a more human-readable form for reporting purposes (i.e., arbitrary queries)

Relative to the requirements laid out in Section 1.1, notice that this example has the following characteristics:

- **One-Way Roundtripping:** The user of the application will have the expectation that new patients, staff members, or procedures can be added in the tool, and subsequent retrievals will pull the correct data. The same can be said about updated patients, etc. The same is not true in reverse; someone could technically add a new procedure to the database manually, but if the user pulls that procedure into the application, changes an attribute, and then changes it back, the database state may not be in the same state, as the values in the "LastModifiedBy" fields may be different.
- **Object-Relational Mapping:** The procedure class hierarchy must be mapped down to tables in some manner.
- **Relation-Relational Mapping:** The way that the procedure hierarchy is mapped to tables is far more complex than is necessary. In particular, the data is partitioned and pivoted.
- **Business Logic:** The "VTStart", "VTEnd", and "LastModifiedBy" columns contain data not present in the application model, and are populated based on business rules about data retention.
- **CRUD Operations:** The application operating over this schema must be able to create, retrieve, update, and delete individual patients, staff, and procedures.



(a)

Procedure:

ProcedureID	PatientID	ProcedureDate	ProcedureKind
345	116	01/10/11	ProcedureA

TextData:

ProcedureID	Property	TextValue	VTStart	VTEnd	LastModifiedBy
345	Location	A150	01/10/11		davis
345	Visualization	Poor	01/10/11	01/12/11	davis
345	Visualization	Fair	01/12/11		travis

NumericData:

ProcedureID	Property	NumberValue	VTStart	VTEnd	LastModifiedBy
345	Duration	40	01/10/11	01/12/11	davis
345	Duration	50	01/12/11		travis
345	Depth	25	01/10/11		davis
345	Anesthetist	10	01/10/11		davis

Patient:

PersonID	FirstName	LastName	InsuranceID	DOB	SSN
116	Robert	Knapp	KN000468	04/16/45	123-45-6789

Staff:

PersonID	FirstName	LastName	HireDate	Title
10	Jane	Edwards	05/16/05	Chief Anesthetist

(b)

Fig. 3. Examples of instances in the example application, both in its object representation (a) and its relational storage (b)

- **Queries and Set-Based Updates:** The application includes the capability to search for patients, staff, or procedures based on complex search criteria built from a graphical interface. The application can also edit existing staff information based on employment data changes.
- **Application Schema Evolution:** The schema for clinical data changes over time — potentially frequently — as more or different data is collected during procedures. These changes directly impact the application schema, and when they happen, the database must adapt to compensate.

Section 8 revisits this example, demonstrating how to accomplish the same mapping but with additional capabilities and none of the tools in the list above.

3 An Introduction to Virtual Databases and Channels

This paper demonstrates how to support virtual databases that are *indistinguishable* from a "real" database in the same way that a virtual machine is indistinguishable from a hardware machine. This capability requires that the user (e.g., an application developer) be able to issue queries, DML operations (insert, update, and delete), as well as DDL operations (to define and modify both schema and constraints) against the application's virtual schema. The candidate tool described in this paper for supporting virtual databases is called a *channel*. One constructs a channel by composing atomic schema transformations called *channel transformations* (CT's), each of which is capable of transforming arbitrary queries, data manipulation statements, schema evolution primitives, and referential integrity constraints addressing the input schema (called the *virtual schema*) into equivalent constructs against its output schema (called the *physical schema*). Our approach is similar to Relational Lenses [8] in that one constructs a mapping out of atomic transformations. Lenses use a state-based approach that resolves an updated instance of a view schema with a physical schema instance, whereas a channel translates query, DML, and DDL update statements directly.

This section defines an initial set of CT's that cover a large number of database restructuring operations seen in practice. Later sections show how CT's can be formally defined by describing how they transform the full range of query, DML, and DDL statements. The framework includes a definition of correctness criteria for CT's that guarantees indistinguishability. All CT's must support one-way invertibility where operations issued against the input database, after being propagated to the output database, have the same effect (as observed from all operations issued against the input database) as if the operations had been issued against a materialized instance of the input database.

3.1 Channels and Channel Transformations

A *channel transformation* (CT) is a uni-directional mapping from an input (virtual) schema S to an physical schema $\overline{S}$ that encapsulates an instance transformation. A CT represents an atomic unit of transformation that is known to be updatable. A *channel* is built by composing CT's. A channel is defined by starting with the virtual schema and applying transformations one at a time until the desired physical schema is achieved, which explains the naming conventions of the transformations. Figure 4 shows a graphical representation of how applications interact with a virtual schema connected to a physical one through a channel.

Formally, a CT is a 4-tuple of functions $(\mathsf{S}, \mathsf{I}, \mathsf{Q}, \mathsf{U})$, each of which translates statements expressed against the CT's virtual schema into statements against its physical schema. Let $\mathcal{S}$ be the set of possible relational schemas and $\mathcal{D}$ be the set of possible database instances. Let $\mathcal{Q}$ be the set of possible relational algebra queries. Let $\mathcal{U}$ be the set of possible database update statements, both data (DML) and schema (DDL), as listed in Table 1. Let $[\mathcal{U}]$ be the set of finite lists of update statements — i.e., an element of $[\mathcal{U}]$ is a transaction of updates. Finally, let ϵ represent an error state.

- Function S is a *schema transformation* $\mathsf{S} : \mathcal{S} \rightarrow \mathcal{S} \cup \{\epsilon\}$. A channel transformation may have prerequisites on the virtual schema s, where $\mathsf{S}(s) = \epsilon$ if those prerequisites are not met. Function S must be injective (1-to-1) whenever $\mathsf{S}(s) \neq \epsilon$.

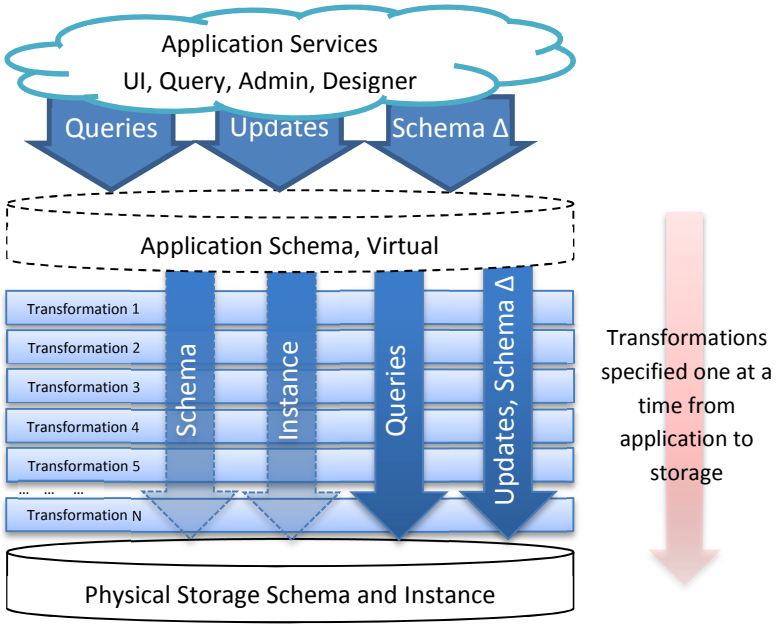


Fig. 4. A channel connecting a virtual schema (which fields all operations from application services) to a concrete, physical one

- Function I is an *instance transformation* $\mathsf{I} : \mathcal{S} \times \mathcal{D} \rightarrow \mathcal{D}$, defined on pairs of input (s, d) where $\mathsf{S}(s) \neq \epsilon$ and instance d conforms to schema s. Function I must be injective on its second argument, and output a valid instance of $\mathsf{S}(s)$.
- Function Q is a *query transformation* $\mathsf{Q} : \mathcal{S} \times \mathcal{Q} \rightarrow \mathcal{Q}$, de-fined on pairs of input (s, q) where $\mathsf{S}(s) \neq \epsilon$ and query q is valid over schema s, i.e., the query executed on an instance of the schema would not return errors. Function Q must be injective on its second argument, and output a valid query over $\mathsf{S}(s)$.
- Function U is an *update transformation* $\mathsf{U} : \mathcal{S} \times [\mathcal{U}] \rightarrow [\mathcal{U}] \cup \epsilon$, defined on pairs of input $(s, \boldsymbol{u})$ where $\mathsf{S}(s) \neq \epsilon$ and update transaction $\boldsymbol{u}$ is valid over schema s, where each update in the transaction references existing schema elements and when executed on schema s do not cause errors or schema conflicts (e.g., renaming a column of a table to a new name that conflicts with an existing column). Function U must be injective on its second argument when $\mathsf{U}(s, \boldsymbol{u}) \neq \epsilon$, and output a valid update transaction over $\mathsf{S}(s)$. Expression $\mathsf{U}(s, \boldsymbol{u})$ evaluates to the error state ϵ if $\boldsymbol{u}$ applied to s produces schema s' where $\mathsf{S}(s') = \epsilon$.

The function S (and function I) provides the semantics for a CT in terms of translating a virtual schema (and an instance of it) into a physical schema (and an instance of it). These functions are not used in any implementation, but allow one to reason about the correctness of functions Q and U. Neither query nor update functions require a database instance as input; a CT directly translates the statements themselves.

The associated functions of a channel transformation must satisfy the following commutativity properties, where $q(d)$ means executing query q over instance d, and $\boldsymbol{u}(s)$ means executing update transaction $\boldsymbol{u}$ on schema s:

- For a virtual schema s, a concrete instance d of s, and a query q, let $\overline{q} = \mathsf{Q}(s, q)$ (the translated query) and $\overline{d} = \mathsf{I}(s, d)$ (the translated instance). Then, $\overline{q}(\overline{d}) = q(d)$. In other words, translating a query and then executing the result on the translated instance will produce the same result as running the query on an instance of the virtual schema (Figure 5(a)).
- For a virtual schema s and a valid update transaction $\boldsymbol{u}$ against s, let $\overline{s} = \mathsf{S}(s)$ (the translated schema) and $\overline{u} = \mathsf{U}(s, \boldsymbol{u})$ (the translated update). Then, $\overline{u}(\overline{s}) = \mathsf{S}(\boldsymbol{u}(s))$. Running a translated update against a translated schema is equivalent to running the update first, then translating the result (Figure 5(b)).
- For a virtual schema s and a valid update transaction $\boldsymbol{u}$ against s, for each table $t \in s$, let q^t be the query `SELECT * FROM t`. Let $\overline{s} = \mathsf{S}(s)$ (the translated schema) and $\overline{u} = \mathsf{U}(s, \boldsymbol{u})$ (the translated update). Finally, let $\overline{q^t_u} = \mathsf{Q}(\boldsymbol{u}(s), q^t)$, the result of translating query q^t on schema s after it was updated by $\boldsymbol{u}$. Then, $\overline{q^t_u}(\overline{u}(\overline{s})) \equiv q^t(\boldsymbol{u}(s))$. Running a translated query against a translated schema that has been updated by a translated update is equivalent to running the query locally after a local update (Figure 5(c)).

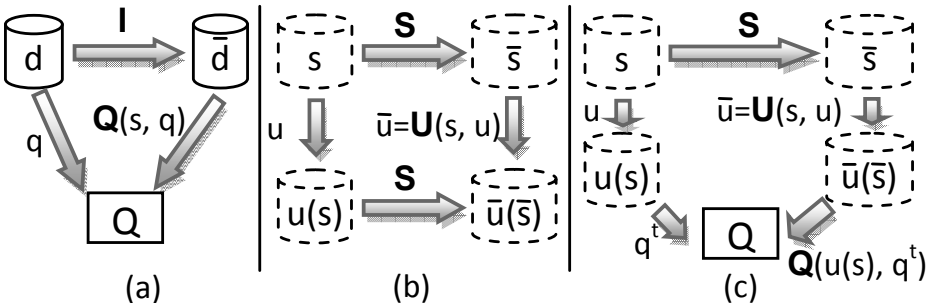


Fig. 5. Three commutativity diagrams that must be satisfied for CT's that have defined instance-at-a-time semantics

The last commutativity property abuses notation slightly by allowing queries to run on a schema instead of a database instance, but the semantics of such an action are straightforward. If s is a schema and q^t is the query `SELECT * FROM t` for $t \in s$, then $q^t(s) \equiv t$, and more complicated queries build on that notion recursively. The notation allows us to reason about queries and updates without referring to database instances by treating a single update statement as interchangeable with the effect it has on an instance:

- If $u = \mathbf{I}(t, \boldsymbol{C}, Q)$, then $q^t(u(s)) \equiv t \cup Q$ (all of the rows that were in t plus the new rows Q added)

- If $u = \mathbf{D}(t, \boldsymbol{F})$, then $q^t(u(s)) \equiv \sigma_{\neg \boldsymbol{F}} t$ (all of the rows that were in t that do not satisfy conditions $\boldsymbol{F}$)
- If $u = \mathbf{AC}(t, C, D)$, then $q^t(u(s)) \equiv t \times \rho_{1 \to C}\{null\}$ (a new column has been added with all null values)
- If $u = \mathbf{DE}(t, C, E)$ for a key column C, then $q^t(u(s)) \equiv \sigma_{C \neq E} t$ (delete rows that have the dropped element for column C)

In addition to the commutativity properties, function U must have the following properties:

- $\mathsf{U}(s, \boldsymbol{u}) = \epsilon \iff \mathsf{S}(\boldsymbol{u}(s)) = \epsilon$. Function U returns an error if and only if applying the update transaction to the virtual schema results in the schema no longer meeting the transformation's schema preconditions.
- If $\mathsf{U}(s, \boldsymbol{u}) \neq \epsilon$ and d is an arbitrary instance of schema s, $\mathsf{U}(s, \boldsymbol{u})(\mathsf{I}(s, d)) = \epsilon \iff \boldsymbol{u}(d) = \epsilon$. Applying a transaction to an instance returns an error in case of a primary or foreign key violation. This property ensures that a violation occurs on the output instance if and only if a violation would occur if a materialized instance of the virtual schema were updated. Note that such a violation occurs when the transaction is *executed* rather than when it is *translated*.

4 Transformations over Relational Schemas

This section and the next few sections continue the discussion of channels and channel transformations by considering channel transformations whose input and output schemas are both relational. A relational CT translates a relational schema into another relational schema, a relational algebra query into another relational algebra query, and relational update statements into relational update statements. A CT is named based on its effect on artifacts in the direction of its operation, even though CT's are applied from the application schema toward the physical schema. For instance, the "HMerge" CT describes a horizontal merging of tables from the virtual schema into a table in the physical schema. Examples of relational CT's include the following transformations, where all parameters with an overbar represent constructs in the CT's output and those with a vector notation (e.g., $\boldsymbol{T}$) are tuples:

- $VPartition(T, f, \overline{T_1}, \overline{T_2})$ distributes the columns of table T into two tables, $\overline{T_1}$ and $\overline{T_2}$. Key columns of T appear in both output tables, and a foreign key is established from $\overline{T_2}$ to $\overline{T_1}$. Non-key columns that satisfy predicate f are in $\overline{T_1}$, while the rest are in $\overline{T_2}$.
- $VMerge(T_1, T_2, \overline{T})$ vertically merges into table $\overline{T}$ two tables T_1 and T_2 that are related by a one-to-one foreign key.
- $HPartition(T, C)$ horizontally partitions the table T based on the values in column C. The output tables are named using the domain elements of column C.
- $HMerge(f, \overline{T}, \overline{C})$ horizontally merges all tables whose schema satisfies predicate f into a new table $\overline{T}$, adding a column $\overline{C}$ that holds the name of the table from which each row came.

- $Apply(T, \boldsymbol{C}, \overline{\boldsymbol{C}}, f, g)$ applies an invertible function f with inverse g to each row in the table T. The function input is taken from columns $\boldsymbol{C}$, and output is placed in columns $\overline{\boldsymbol{C}}$.
- $Unpivot(T, A, V, \overline{T})$ transforms a table T from a standard one-column-per-attribute form into key-attribute-value triples, effectively moving column names into data values in new column A (which is added to the key) with corresponding data values placed in column V. The resulting table is named $\overline{T}$.
- $Pivot(T, A, V, \overline{T})$ transforms a table T in generic key-attribute-value form into a form with one column per attribute. Column A must participate in the primary key of T and provides the names for the new columns in $\overline{T}$, populated with data from column V. The resulting table is named $\overline{T}$.

These informal definitions of CT's describe what each "does" to a fully materialized instance of a virtual schema, but a virtual schema is virtual and thus stateless. Thus, a CT maintains the operational relationship between input and output schemas by translating all operations expressed against the virtual schema into equivalent operations against the physical schema. Two of these CT's — Horizontal Merge and Pivot — will be used as running examples through the next few sections.

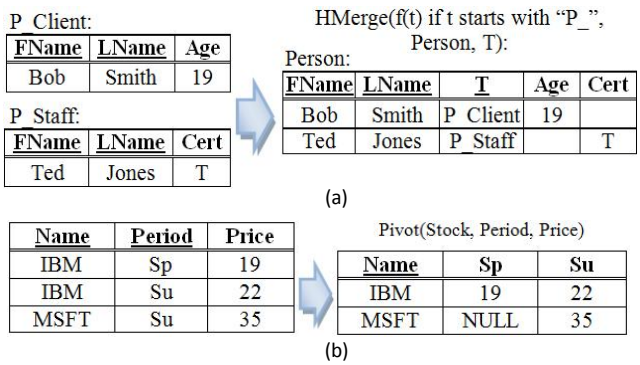


Fig. 6. Instances transformed by an HMerge CT (a) and a Pivot CT (b)

Example: HMerge. The HMerge transformation takes a collection of tables with identically named and union-compatible primary keys and produces their outer union, adding a discriminator column $\overline{C}$ to give each tuple provenance information. Any table in the virtual schema that does not satisfy predicate f is left unaltered[1]. Figure 6(a) shows an example of an HMerge CT.

Let the CT for $HMerge(f, \overline{T}, \overline{C})$ be the 4-tuple $\mathbf{HM} = (\mathsf{S}_{\mathbf{HM}}, \mathsf{I}_{\mathbf{HM}}, \mathsf{Q}_{\mathbf{HM}}, \mathsf{U}_{\mathbf{HM}})$. Let $\boldsymbol{T}^f$ be the set of all tables in virtual schema s that satisfy predicate f and $\mathbf{Cols}(t)$ be the set of columns for table t. Define $\mathsf{S}_{\mathbf{HM}}$ on schema s as follows: replace tables $\boldsymbol{T}^f$ with table $\overline{T}$ with columns $(\bigcup_{t \in \boldsymbol{T}^f} \mathbf{Cols}(t)) \cup \{\overline{C}\}$, the union of all columns from the source

[1] The predicate parameter for HMerge is described only informally in this paper. One such example would be "Table has prefix P_", which is the predicate in Figure 6(a).

Table 1. Relational DML and DDL statements supported by channel transformations

Statement	Formalism	Explanation of Variables
Insert	$\mathbf{I}(T, \boldsymbol{C}, Q)$	Insert rows into table T into columns $\boldsymbol{C}$, using the values of $\boldsymbol{C}$ from the rows in Q. The value of Q may be a table constant or a query result.
Update	$\mathbf{U}(T, \boldsymbol{F}, \boldsymbol{C}, Q)$	Update rows in table T that satisfy all equality conditions $\boldsymbol{F}$ specified on key columns. Non-key columns $\boldsymbol{C}$ hold the new values specified by query or constant Q. Query Q may refer to the pre-update row values as constants. Not all key columns need to have a condition.
Delete	$\mathbf{D}(T, \boldsymbol{F})$	Delete rows from table T that satisfy all equality conditions $\boldsymbol{F}$ specified on key columns. Not all key columns need to have a condition.
Add Table	$\mathbf{AT}(T, \boldsymbol{C}, \boldsymbol{D}, \boldsymbol{K})$	Add new table T, whose columns $\boldsymbol{C}$ have domains $\boldsymbol{D}$, with key columns $\boldsymbol{K} \subseteq \boldsymbol{C}$.
Rename Table	$\mathbf{RT}(T_o, T_n)$	Rename table T_o to be named T_n. Throw error if T_n already exists.
Drop Table	$\mathbf{DT}(T)$	Drop the table named T.
Add Column	$\mathbf{AC}(T, C, D)$	Add to table T a column named C with domain D.
Rename Column	$\mathbf{RC}(T, C_o, C_n)$	In table T, rename the column C_o to be named C_n. Throw error if C_n already exists.
Change Column Facet	$\mathbf{CP}(T, C, F, V)$	For the column C of the table T, change its domain facet F to have the value V. Common facets include whether a column is nullable, the column's maximum length if the column has a string-valued domain, or the precision and scale if the column is numeric-valued.
Drop Column	$\mathbf{DC}(T, C)$	In table T, drop the non-key column C.
Add Element	$\mathbf{AE}(T, C, E)$	In table T, in column C, add a new possible domain value E.
Rename Element	$\mathbf{RE}(T, C, E_o, E_n)$	In table T, in column C, rename domain element E_o to be named E_n. Throw error if E_n conflicts with an existing element.
Drop Element	$\mathbf{DE}(T, C, E)$	In table T, in column C, drop the element E from the domain of possible values.
Add Foreign Key	$\mathbf{FK}(\boldsymbol{F}\|T.\boldsymbol{X} \rightarrow \boldsymbol{G}\|T'.\boldsymbol{Y})$	Add foreign key constraint from columns $T.\boldsymbol{X}$ to columns $T'.\boldsymbol{Y}$, so that for each tuple $t \in T$, if t satisfies conditions $\boldsymbol{F}$ and $t[\boldsymbol{X}] \neq null$, there must be tuple $t' \in T'$ such that t' satisfies conditions $\boldsymbol{G}$ and $t[\boldsymbol{X}] = t'[\boldsymbol{Y}]$.
Drop Foreign Key	$\mathbf{DFK}(\boldsymbol{F}\|T.\boldsymbol{X} \rightarrow \boldsymbol{G}\|T'.\boldsymbol{Y})$	Drop the constraint imposed by the enclosed statement.
Add Constraint	$\mathbf{Check}(Q_1 \subseteq Q_2)$	Add a check constraint so that the result of query Q_1 must always be a subset of the results of query Q_2. This constraint is also called a Tier 3 FK.
Drop Constraint	$\mathbf{DCheck}(Q_1 \subseteq Q_2)$	Remove the check constraint between the results of queries Q_1 and Q_2.
Loop	$\mathbf{Loop}(t, Q, \boldsymbol{S})$	For each tuple t returned by query Q, execute transaction $\boldsymbol{S}$.
Error	$\mathbf{Error}(Q)$	Execute query Q, and raise an error if any rows are returned.

tables eliminating duplicates, plus the provenance column, whose domain is the names of the tables in $\boldsymbol{T}^f$. The key of $\overline{T}$ is the common key from tables $\boldsymbol{T}^f$ plus the column $\overline{C}$. $\mathsf{S}_{\mathbf{HM}}(s) = \epsilon$ if the keys are not union-compatible and identically named.

Define $\mathsf{I}_{\mathbf{HM}}$ on schema s and instance d by replacing the instances of $\boldsymbol{T}^f$ in d with $\biguplus_{t \in \boldsymbol{T}^f}(t \times \{(name(t))\})$, where $\biguplus$ is outer union with respect to column name (as opposed to column position) and $name(t)$ represents the name of the table t as a string value.

Example: Pivot. Recall that a Pivot CT takes four arguments: T (the table to be pivoted), A (a column in the table holding the data that will be pivoted to form column names in the result), V (the column in the table holding the data to populate the pivoted columns), and $\overline{T}$ (the name of the resulting table). Let the channel transformation for $Pivot(T, A, V, \overline{T})$ be the 4-tuple $\mathbf{PV} = (\mathsf{S}_{\mathbf{PV}}, \mathsf{I}_{\mathbf{PV}}, \mathsf{Q}_{\mathbf{PV}}, \mathsf{U}_{\mathbf{PV}})$. An example instance transformation appears in Figure 6(b).

Let $\mathsf{S}_{\mathbf{PV}}$ be defined on schema s by removing table T (which has key columns $\boldsymbol{K}$ and non-key columns $\boldsymbol{N}$, where $A \in \boldsymbol{K}$ and $V \in \boldsymbol{N}$), and replacing it with $\overline{T}$ with key columns $(\boldsymbol{K} - \{A\})$ and non-key columns $(\boldsymbol{N} - \{V\} \cup \mathbf{Dom}(A))$. $\mathbf{Dom}(A)$ represents the domain of possible values of column A (not the values present in any particular instance);

therefore, the output of $\mathsf{S_{PV}}(s)$ is based on the domain definition for A as it appears in schema. The new columns for each element in $\mathbf{Dom}(A)$ have domain $\mathbf{Dom}(V)$. If A is not present or not a key column, or if $\mathbf{Dom}(A)$ has any value in common with an input column of T (which would cause a name conflict in the output), then $\mathsf{S_{PV}}(s) = \epsilon$.

Let $\mathsf{I_{PV}}$ be defined on schema s and instance d by replacing the instance of T in d with $\not\boxtimes^{\rightarrow}_{\mathbf{Dom}(A);A;V} T$, where $\not\boxtimes^{\rightarrow}$ is an extended relational algebra operator that performs a pivot, detailed in the next section. Formally, $\mathbf{Dom}(A)$ could be any finite domain; practically speaking, **PV** would only be applied where $\mathbf{Dom}(A)$ is some small, meaningful set of values such as the months of the year or a set of possible stock ticker names.

4.1 Translating Queries

Each CT receives queries, expressed in extended relational algebra addressing the CT's virtual schema, and produces queries expressed in extended relational algebra addressing its physical schema. The query language accepted by a channel includes the eight standard relational algebra operators (σ, π, $\times$, $\bowtie$, $\cup$, $\cap$, $-$, and $\div$)[2], the rename operator (ρ), table and row constants, plus:

- Left outer join ($\rightarrowtriangle\!\bowtie$) and left antisemijoin ($\overline{\ltimes}$) [6]
- Pivot ($\not\boxtimes^{\rightarrow}_{\boldsymbol{C};A;V}$): For a set of values $\boldsymbol{C}$ on which to pivot, pivot column A, and pivot-value column V (translating a relation from key-attribute-value triples into a normalized, column-per-attribute form)
- Unpivot ($\not\boxtimes^{\leftarrow}_{\boldsymbol{C};A;V}$), the inverse operation to pivot
- Function application ($\alpha_{\boldsymbol{I},\boldsymbol{O},f}$): Apply function f iteratively on all rows, using columns $\boldsymbol{I}$ as input and placing the result in output columns $\boldsymbol{O}$

The pivot query operator is defined as:

$$\not\boxtimes^{\rightarrow}_{\boldsymbol{C};A;V} Q \equiv (\pi_{columns(Q)-\{A,V\}} Q) \rightarrowtriangle\!\bowtie (\rho_{V\rightarrow C1} \pi_{columns(Q)-\{A\}} \sigma_{A=C1} Q)$$

$$\rightarrowtriangle\!\bowtie \ldots \rightarrowtriangle\!\bowtie (\rho_{V\rightarrow Cn} \pi_{columns(Q)-\{A\}} \sigma_{A=Cn} Q) \text{ for } C1, \ldots, Cn = \boldsymbol{C}$$

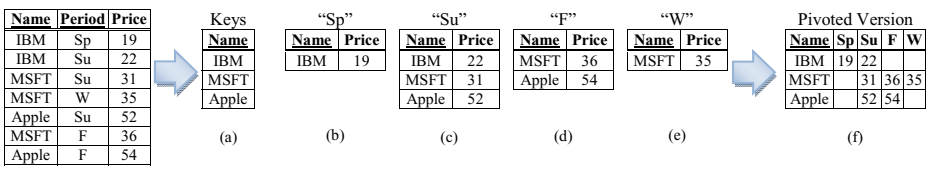


Fig. 7. An example of an instance transformed by the pivot query operator $\not\boxtimes^{\rightarrow}_{\{Sp,Su,F,W\};Period;Price}$, first broken down into intermediate relations that correspond to the set of non-pivoted columns (a) and the subsets of rows corresponding to each named value in the pivot column "Period" (b–e). The pivoted instances are then outer joined with the first instance (Keys) to produce the pivot table (f).

[2] For a good primer on the relational data model and relational algebra, consider [15]

Note that the pivot (and unpivot) query operators above have an argument giving the precise values on which to pivot (or columns to unpivot, respectively); as a result, both query operators have fixed input and output schemas. This flavor of the pivot and unpivot operator is consistent with implementations in commercial databases (e.g., the PIVOT ON clause in SQL Server [29]). Contrast this property with the Pivot CT, where the set of output columns is dynamic. Some research extensions to SQL, including SchemaSQL [23] and FISQL [45,46], introduce relation and column variables that can produce the effect of dynamic pivoting, but over unconstrained or manually constrained domains. The Pivot CT lays between these two cases in functionality, where the set of pivot columns is dynamic but still constrained.

Figure 7 shows an example instance transformed by a pivot operator $⍁_{C;A;V}$, with the transformation broken down into stages. First, columns A and V are dropped using the project operator, with only the key for the pivoted table remaining. Then, for each value C in the set $\boldsymbol{C}$, instance $\rho_{V\to C}\pi_{columns(Q)-\{A\}}\sigma_{A=C}Q$ is constructed consisting of all rows in the instance that have value C in the pivot column A, with the "value" column V renamed to C to disambiguate it from other value columns in the pivot table. Finally, each resulting table is left-outer-joined against the key table, filling the key table out with a column for each value C.

A pivot operator is useful in the algebra because, like joins, there are well-known $N \log N$ algorithms involving a sort of the instance followed by a single pass to fill out the pivot table. Some details of the pivot query operator are left aside, such as what to do if there exist multiple rows in the instance with the same key-attribute pair, since the exact semantics of what to do in these cases have no bearing on the operation of a channel (Wyss and Robertson have an extensive formal treatment of Pivot [45]).

The unpivot query operator is defined as follows:

$$⍂_{\boldsymbol{C};A;V}Q \equiv \bigcup_{C\in\boldsymbol{C}}(\rho_{C\to V}\pi_{columns(Q)-(\boldsymbol{C}-\{C\})}\sigma_{C<>null}(Q) \times \rho_{1\to A}(name(C)))$$

where $name(C)$ represents the name of attribute C as a constant (to disambiguate it from a reference to instance data).

Each CT translates a query — including any query appearing as part of a DML or DDL statement — in a fashion similar to view unfolding. That is, function Q looks for all references to tables in the query and translates them in-place as necessary.

As an example, consider $\mathsf{Q}_{\mathbf{PV}}$, the query translation function for Pivot, which translates all references to table T into $⍂_{\mathbf{Dom}(A);A;V}\overline{T}$. That is, the query translation introduces an unpivot operator into the query to effectively undo the action that the Pivot CT performs on instances. Of particular note is that the first parameter to ⍂ is populated by the CT with the elements in the domain of column A at the time of translation. Thus, the queries generated by the Pivot transformation will always reference the appropriate columns in the pivoted logical schema, even as elements are added or deleted from the domain of the attribute column in the virtual schema, and thus columns are added or deleted from the physical schema. (Pivot will process the DDL statements for adding or dropping domain elements — see Section 4.3 for an example.)

Because the set of columns in T without V is a superkey, there can never be two rows with the same key-attribute combination; thus, unlike the pivot relational query operator in general, the Pivot CT need not deal with duplicate key-attribute pairs.

HMerge Translation of Queries. Function $\mathsf{Q_{HM}}$ translates all references to a table $t \models f$ into the expression $\pi_{\mathbf{Cols}(t)}\sigma_{\overline{C}=t}\overline{T}$. That is, $\mathsf{Q_{HM}}$ translates a table reference t into a query that retrieves all rows from the merged table that belong to virtual schema table t as a selection condition on the provenance column, and a projection down to the columns in the virtual schema for t.

To prove that function $\mathsf{Q_{HM}}$ respects the commutativity properties, one must show that the translation effectively undoes the outer-union operation, which follows from relational algebra equivalences.

4.2 Translating DML Statements

The set of update statements accepted by a channel is shown in Table 1. A channel transformation supports the insert, update, and delete DML statements. Update and delete conditions must be equality conditions on key attributes, and updates are not allowed on key attributes, assuming that the application will issue a delete followed by an insert. Channels also support a loop construct, denoted as $\mathbf{Loop}(t, Q, \boldsymbol{S})$, similar to a cursor: t is declared as a row variable that loops through the rows of the result of Q. For each value t takes on, the sequence of statements $\boldsymbol{S}$ execute. Statements in $\boldsymbol{S}$ may be any of the statements from Table 1 and may use the variable t as a row constant. Using **Loop**, one can mimic the action of arbitrary update or delete conditions by using a query to retrieve the key values for rows that match the statement's conditions, then issue an update or delete for each qualifying row. Channels support an error statement $\mathbf{Error}(Q)$ that aborts the transaction if the query Q returns a non-empty result.

A complete definition of the update translation function U includes computation of $\mathsf{U}(\mathbf{I}(T, \boldsymbol{C}, Q))$, $\mathsf{U}(\mathbf{D}(T, \boldsymbol{F}))$, etc. for each statement in Table 1 for arbitrary parameter values. The results are concatenated based on the original transaction order to form the output transaction. For instance, for an update function U, if $\mathsf{U}(s, [u_1]) = [\overline{u_1}, \overline{u_2}]$ and $\mathsf{U}(s, [u_2]) = [\overline{u_3}, \overline{u_4}, \overline{u_5}]$, then $\mathsf{U}(s, [u_1, u_2]) = [\overline{u_1}, \overline{u_2}, \overline{u_3}, \overline{u_4}, \overline{u_5}]$. An error either on translation by U (i.e., U evaluates to ϵ on a given input) or during execution against the instance aborts a transaction.

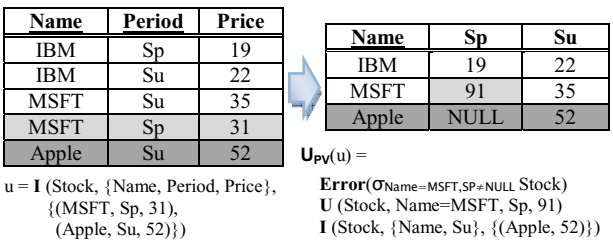


Fig. 8. An example of inserts translated by a Pivot CT

HMerge Translation of Inserts. Let $\mathsf{U_{HM}}$ be the update function for the transformation $HMerge(f, \overline{T}, \overline{C})$, and let s be its virtual schema. Define the action of $\mathsf{U_{HM}}$ on an Insert statement $\mathbf{I}(t, \boldsymbol{C}, Q)$ where $t \models f$ as follows:

$$\mathsf{U}_{\mathbf{HM}}(s, \mathbf{I}(t, \boldsymbol{C}, Q)) = \mathbf{I}(\overline{T}, \boldsymbol{C} \cup \{\overline{C}\}, Q \times \{name(t)\})$$

where $name(t)$ is the string-valued name of table t. The translation takes all rows Q that are to be inserted into table t and attaches the value for the provenance column in the output. An example is shown in Figure 9. Since the output consists entirely of insert statements, proving that $\mathsf{U}_{\mathbf{HM}}$ respects the commutativity properties for insert statements reduces to showing that the newly added rows, when queried, appear in the virtual schema in their original form. In short, one must show that $\pi_{\mathbf{Cols}(t)}\sigma_{\overline{C}=t}(Q \times \{name(t)\}) = Q$, which can be shown to be true by relational equivalences.

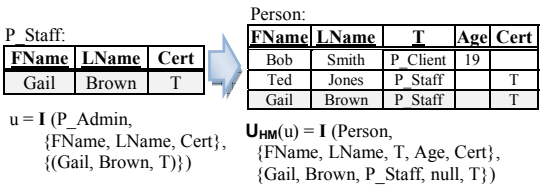


Fig. 9. An example of an insert statement translated by an HMerge CT

Pivot Translation of Inserts. Now consider $\mathsf{U}_{\mathbf{PV}}(\mathbf{I}(T, \boldsymbol{C}, Q))$, pushing an insert statement through a Pivot. Each tuple $(\boldsymbol{K}, A, V)$ inserted into the virtual schema consists of a key value, an attribute value, and a data value; the key value uniquely identifies a row in the pivoted table, and the attribute value specifies the column in the pivoted table. $\mathsf{U}_{\mathbf{PV}}$ thus transforms the insert statement into an update statement that updates column A for the row with key $\boldsymbol{K}$ to be value V, if the row exists. In Figure 8, the inserted row with `Name = ''MSFT''` corresponds to a key value already found in the physical schema; that insert row statement therefore translates to an update in the physical schema. The other row, with `Name = ''Apple''`, does not correspond to an existing key value, and thus translates to an insert.

The Pivot CT adds an error statement to see if there are any key values in common between the new rows and the existing values in the output table, and if so, returns an error, as this situation indicates that a primary key violation would have occurred in a materialized virtual schema. Next, using a Loop statement, for each row in Q that corresponds to an existing row in the output table, generated statements find the correct row and column and set its value. A final insert statement finds the rows in Q that do not correspond to existing rows in the output table, pivots those, and inserts them.

Let s be the virtual schema of the CT $Pivot(T, A, V, \overline{T})$. Define the action of the CT's update function $\mathsf{U}_{\mathbf{PV}}$ on an insert DML statement $\mathbf{I}(T, \boldsymbol{C}, Q)$ as follows:

$\mathsf{U}_{\mathbf{PV}}(s, \mathbf{I}(T, \boldsymbol{C}, Q)) =$
  $\mathbf{Error}((\pi_{\mathbf{Keys}(T)}Q) \cap \pi_{\mathbf{Keys}(\overline{T})}\not\boxtimes_{\mathbf{Dom}(A);A;V}(\pi_{\mathbf{Cols}(\overline{T})}(Q \bowtie \overline{T}))),$
    (check that inserted rows do not collide with existing data)
  $\forall_{a \in \mathbf{Dom}(A)}\mathbf{Loop}(t, \sigma_{A=a}Q \bowtie (\pi_{\mathbf{Keys}(\overline{T})}T),$
    $\mathbf{U}(\overline{T}, \forall_{c \in \mathbf{Keys}(\overline{T})}c = t[c], \{a\}, \pi_V t)),$
    (update each row whose key is already present)
  $\mathbf{I}(\overline{T}, \mathbf{Cols}(\overline{T}), \not\boxtimes_{\mathbf{Dom}(A);A;V}(Q \not\boxtimes (\pi_{\mathbf{Keys}(\overline{T})}\overline{T})))$
    (inserts for non-existent rows)

4.3 Translating DDL Statements

Table 1 includes the full list of supported schema and constraint update statements. The domain-element DDL statements are unique to our approach. If a domain element E in column C is dropped, and C is not a key column, then any row that had a C value of E will have that value set to null. However, if instead C is a key attribute, then any such row will be deleted. In addition, the Rename Element DDL statement will automatically update an old domain value to the new one. Since renaming an element can happen on any column, key or non-key, renaming elements is a way to update key values in-place. Note that the set of changes in Table 1 is complete in that one can evolve any relational schema S to any other relational schema S' by dropping any elements they do not have in common and adding the ones unique to S' (a similar closure argument has been made for object-oriented models, e.g. [5]).

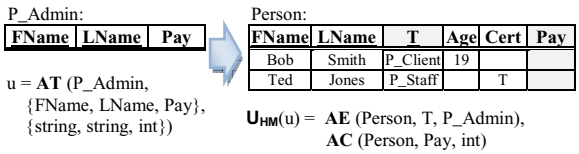


Fig. 10. An Add Table statement translated by HMerge

HMerge Translation of Add Table. Let $\mathsf{U}_{\mathbf{HM}}$ be the update function for transformation $HMerge(f, \overline{T}, \overline{C})$, and let s be its virtual schema. Define the action of $\mathsf{U}_{\mathbf{HM}}$ on an Add Table statement for a table t that satisfies f as follows:

$\mathsf{U}_{\mathbf{HM}}(s, \mathbf{AT}(t, \boldsymbol{C}, \boldsymbol{D}, \boldsymbol{K})) =$
If $\overline{T}$ exists, then $\mathbf{AE}(\overline{T}, \overline{C}, t)$, and for each column c in table t,
$(\nexists_{s \models f} c \in \mathbf{Cols}(s)) \rightarrow \mathbf{AC}(\overline{T}, c, \mathbf{Dom}(c))$
If $\overline{T}$ not yet created, then
$\mathbf{AT}(\overline{T}, \boldsymbol{C} \cup \{\overline{C}\}, \boldsymbol{D} \cup \{\{name(t)\}\}, \boldsymbol{K} \cup \{\overline{C}\})$

If the merged table already exists in the physical schema, the function adds a new domain element to the provenance column to point to rows coming from the new table. Then, the function $\mathsf{U}_{\mathbf{HM}}$ adds any columns that are unique to the new table. If the new table is the first merge table, the output table is created using the input table as a template. An example is shown in Figure 10, assuming tables "P_Client" and "P_Staff" already exist in the virtual schema.

HMerge Translation of Add Column. Let $\mathsf{U}_{\mathbf{HM}}$ be the update function for the transformation $HMerge(f, \overline{T}, \overline{C})$, and let s be its virtual schema. Define the action of $\mathsf{U}_{\mathbf{HM}}$ on an Add Column statement $\mathbf{AC}(t, C, D)$ for one of the merged tables $t \models f$ as follows:

$\mathsf{U_{HM}}(s, \mathbf{AC}(t, C, D)) =$
If C is not a column in any other merged table besides t, then
$\mathbf{AC}(\overline{T}, C, D)$
If C exists in another merged table t', and $t'.C$ has a different domain, then ϵ (abort — union compatibility violated)
If C exists in other merged table(s), all with the same domain, then $\emptyset$ (leave output unchanged)

Pivot Translation of Drop Element. Let $\mathsf{U_{PV}}$ be the update function for the transformation $Pivot(T, A, V, \overline{T})$, and let s be its virtual schema. Define the action of $\mathsf{U_{PV}}$ on Drop Element DDL statements as follows:

$\mathsf{U_{PV}}(s, \mathbf{DE}(T, C, E)) =$
If $C = A$, then $\mathbf{DC}(\overline{T}, E)$
Else if $C = V$, then $\forall_{c \in \mathbf{Dom}(A)} \mathbf{DE}(\overline{T}, c, E)$
Else, $\mathbf{DE}(\overline{T}, C, E)$

If dropping an element from the attribute column, translate into a Drop Column. If dropping an element from the value column, translate into Drop Element statements for each pivot column. Otherwise, leave unaffected (also leave unaffected for any Drop Element statement on tables other than T). An example of Drop Element translation is in Figure 11.

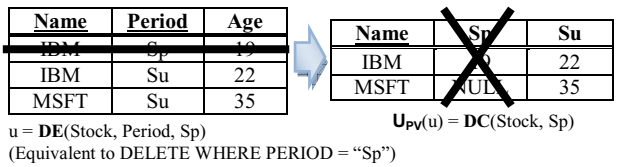


Fig. 11. An example of a Drop Element statement translated by a Pivot CT

Let us use this translation as an example of how to prove the correctness of a CT's update translation, in particular dropping an element from the "attribute" column A in the input to the Pivot CT. Proving the first property is omitted for space and left as an exercise for the reader. To prove the second commutativity property, one must demonstrate that the schema that results from adding the pivot table with the element still present through the pivot followed by dropping the element has the same result as pushing the table's schema through without the element.

Proposition: Let s be a schema with T undefined. Then:
$\mathsf{U_{PV}}(s, \{\mathbf{AT}(T, \boldsymbol{C} \cup \{A\}, \boldsymbol{D} \cup \{D' - \{E\}\}, \boldsymbol{K} \cup \{A\})\})$
$= \mathsf{U_{PV}}(s, \{\mathbf{AT}(T, \boldsymbol{C} \cup \{A\}, \boldsymbol{D} \cup \{D'\}, \boldsymbol{K} \cup \{A\}), (\mathbf{DE}(T, A, E_0))\})$.
Proof: $\mathsf{U_{PV}}(s, \mathbf{AT}(T, \boldsymbol{C} \cup \{A\}, \boldsymbol{D} \cup \{D' - \{E\}\}, \boldsymbol{K} \cup \{A\}))$
$= \mathbf{AT}(\overline{T}, (\boldsymbol{C} - \{V\}) \cup D' \cup \{E_0\},$
$\boldsymbol{D} - \{\mathbf{Dom}(V)\} \cup \{\forall_{a \in D' - \{E\}} \mathbf{Dom}(V)\}, \boldsymbol{K})$

(Push the Add Table statement through the Pivot)
$= \mathbf{AT}(\overline{T}, (\boldsymbol{C} - \{V\}) \cup D', \boldsymbol{D} - \{\mathbf{Dom}(V)\} \cup \{\forall_{a \in D'} \mathbf{Dom}(V)\}, \boldsymbol{K}),$
$\quad \mathbf{DC}(\overline{T}, E, \mathbf{Dom}(V))$
(DDL equivalence)
$= \mathsf{U}_{\mathsf{PV}}(s, \{\mathbf{AT}(T, \boldsymbol{C} \cup \{A\}, \boldsymbol{D} \cup \{D'\}, \boldsymbol{K} \cup \{A\}), \mathbf{DE}(T, A, E)\})$
(View the statements in their pre-transformation image)
□

Finally, one needs to prove the commutativity property from Figure 5(c):

Proposition: Let s be a schema with T defined. Then:
$\mathsf{Q}_{\mathsf{PV}}(\mathbf{DE}(T, C, E)(s), q^T)(\mathbf{DC}(T, E)(\mathsf{S}_{\mathsf{PV}}(s))) = q^T(\mathbf{DE}(T, C, E)(s))$
Proof: $\mathsf{Q}_{\mathsf{PV}}(\mathbf{DE}(T, C, E)(s), q^T)(\mathbf{DC}(T, E)(\mathsf{S}_{\mathsf{PV}}(s)))$
$= (\not\boxtimes_{\mathbf{Dom}(A)-\{E\};A;V} \overline{T})(\mathbf{DC}(\overline{T}, E)(\mathsf{S}_{\mathsf{PV}}(s)))$
(Transforming the query q^T, but on a schema where column A has lost element E)
$= \not\boxtimes_{\mathbf{Dom}(A)-\{E\};A;V} \pi_{\mathbf{Cols}(\overline{\mathbf{T}})-\{\mathbf{E}\}} \overline{T}$
(Dropping a column has the effect of projecting it away)
$= \sigma_{A \neq E} \not\boxtimes_{\mathbf{Dom}(A);A;V} \overline{T}$
(Extended relational algebra equivalence for unpivot)
$= \sigma_{A \neq E} q^T(s)$
(Pull query back through transformation on original schema)
$= q^T(\mathbf{DE}(T, C, E)(s))$
(Effect of Drop Element statement on a key column is to delete all rows with that value)
□

4.4 Translating Foreign Keys

Consider three levels — or *tiers* — of referential integrity, offering a trade-off between expressive power and efficiency. A Tier 1 foreign key is a standard foreign key in the traditional relational model. A Tier 3 foreign key $\mathbf{Check}(Q_1 \subseteq Q_2)$ is a containment constraint between two arbitrary queries. A Tier 2 foreign key falls between the two, offering more expressiveness than an ordinary referential integrity constraint but with efficient execution.

A Tier 2 foreign key statement $\mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G}|U.\boldsymbol{Y})$ is equivalent to statement $\mathbf{Check}(\sigma_F \pi_X T \subseteq \sigma_G \pi_X U)$, where $\boldsymbol{Y}$ is a (not necessarily proper) subset of the primary key columns of table U, and $\boldsymbol{F}$ and $\boldsymbol{G}$ are sets of conditions on key columns (for their respective relations) with AND semantics. The statement $\mathbf{FK}(true|T.\boldsymbol{X} \rightarrow true|U.\boldsymbol{Y})$ is therefore a Tier 1 primary key — a foreign key in the traditional sense — if $\boldsymbol{Y}$ is the key for table U.

To translate **FK** (and **DFK**) statements, one can leverage the insight that any **FK** statement can be restated as a **Check** statement. Statements **Check** (and **DCheck**) have behavior specified as queries, so their translation follows directly from query translation. It becomes immediately clear why additional levels of referential integrity are required; if one specifies a standard integrity statement $FK(true|T.\boldsymbol{X} \rightarrow true|U.\boldsymbol{Y})$ against a virtual schema, its image in the physical schema may involve arbitrarily complex queries.

A foreign key constraint in the standard relational model is a containment relationship between two queries, $\pi_{\boldsymbol{C}} T \subseteq \pi_{\boldsymbol{K}} T'$, where $\boldsymbol{C}$ is the set of columns in T comprising the foreign key and $\boldsymbol{K}$ is the key for T'. Figure 12(a) shows a traditional foreign key between two tables. Figure 12(b), shows the same two tables and foreign key after the target table of the foreign key has been horizontally merged with other tables. The foreign key now points to only part of the key in the target table and only a subset of the rows, a situation that is not expressible using traditional relational foreign keys. Figure 12(c) shows the same tables as Figure 12(a), but this time, the target table has been pivoted. Now, the "target" of the foreign key is a combination of schema and data values.

Thus, propagating an ordinary foreign key through a CT may result in a containment query involving arbitrary extended relational algebra. It is possible to translate a foreign key constraint $Q_1 \subseteq Q_2$ through a CT simply by translating queries Q_1 and Q_2. However, one can observe that in many cases, the translated query is in the form $\pi_{\boldsymbol{C}'} \sigma_{\boldsymbol{F}'} T'$ or even $\pi_{\boldsymbol{C}'} T'$, though not necessarily covering a table's primary key. A containment constraint using these simple queries may be enforced by triggers with reasonable and predictable performance.

Table 1 lists the two statements that can establish integrity constraints, **FK** and **Check**. The update function U for a CT translates a **Check** statement by translating its constituent queries via the CT's query translation function Q. Note that as a consequence, if a CT translates an **FK** statement into a Tier 3 foreign key requiring a **Check** statement, it will stay as a **Check** statement through the rest of the channel.

There are two additional statements listed in Table 1 that drop referential integrity constraints — **DFK** and **DCheck**. A CT translates these statements in the same fashion as their "add" analog.

Tiered Foreign Keys. A Tier 1 foreign key defined from columns $T.\boldsymbol{X}$ to table T' with primary key Y is equivalent to the following logical expression:

$$\forall_{t \in T} t[\boldsymbol{X}] \neq \mathit{null} \rightarrow \exists_{t' \in T'} t[\boldsymbol{X}] = t'[\boldsymbol{Y}].$$

A Tier 2 foreign key statement $\mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})$ is equivalent to the following logical expression:

$$\forall_{t \in T} \exists_{t' \in T'} (t \models \boldsymbol{F} \wedge t[X] \neq \mathit{null}) \longrightarrow (t[X] = t'[Y] \wedge t' \models \boldsymbol{G}).$$

where $\boldsymbol{Y}$ is a (not necessarily proper) subset of the primary key columns of table T', and $\boldsymbol{F}$ and $\boldsymbol{G}$ are sets of conditions on key columns (for their respective relations) with AND semantics. Figure 12(b) shows an example of a Tier 2 foreign key enforced on table instances, and the statement used to create the foreign key.

The foreign key $\mathbf{FK}(\mathit{true}|T.\boldsymbol{X} \rightarrow \mathit{true}|T'.\boldsymbol{Y})$ is precisely a Tier 1 foreign key when $\boldsymbol{Y}$ is the primary key for T'. One can represent Tier 1 FKs using Tier 2 FK syntax $\mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})$ because it simplifies the description of a CT, and because it is trivial to check at runtime whether $\boldsymbol{F}$ and $\boldsymbol{G}$ are empty and $\boldsymbol{Y}$ is a key for T'. Thus, our implementation can determine at runtime when a Tier 2 FK can be implemented in a database as a Tier 1 FK (a standard relational foreign key).

A Tier 3 foreign key is a containment constraint between two queries Q and Q' in arbitrary relational algebra over a single schema, expressed as $\mathbf{Check}(Q \subseteq Q')$.

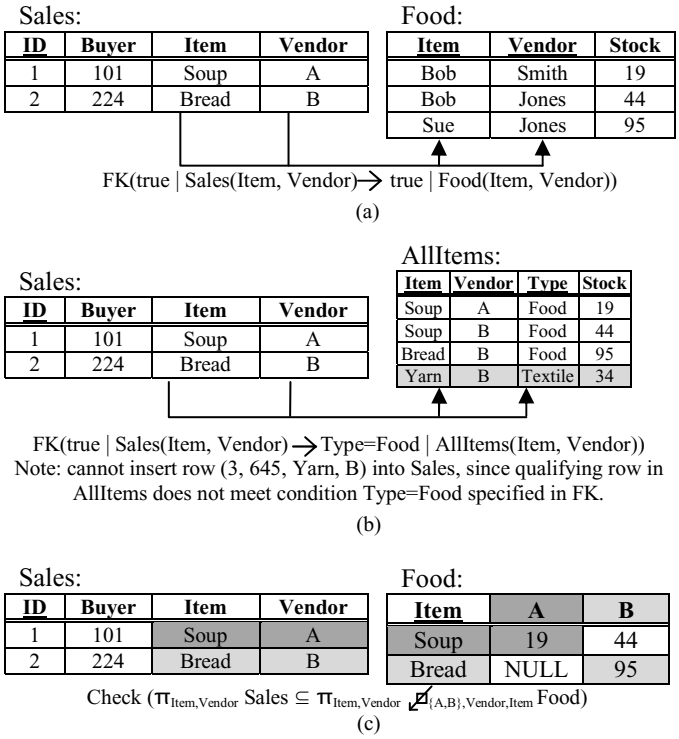


Fig. 12. Examples of Tier 1 (a), Tier 2 (b), and Tier 3 (c) foreign keys

The example in Figure 12(c) can be expressed as a Tier 3 foreign key, where the target of the foreign key is a pivoted table. Since Tier 3 FKs may be time-consuming to enforce, a channel designer should take note of when a CT demotes a Tier 1 or 2 foreign key to Tier 3, i.e., any time a **Check** statement appears in the logic for translating an **FK** statement and consider the tradeoff.

Tier 2 FK as a Trigger. A Tier 2 foreign key **FK**($F|T.X \rightarrow G|T'.Y$) can be enforced in a standard relational database using triggers — specifically, insert and update triggers on the source table T and a delete trigger on the target table T':

```
begin insert trigger (T)
  if new tuple satisfies conditions F
    for each tuple t in T
      if t[Y] = new tuple[X] and t satisfies G
        accept insert
    reject insert
end trigger
(update trigger follows same pattern as insert)
```

```
begin delete trigger (T')
  if deleted tuple satisfies conditions G
    for each tuple t in T
      if t[X] = deleted tuple[Y] and t satisfies F
        delete tuple t
end trigger
```

The worst-case performance for enforcing a Tier 2 foreign key is that tables T and T' must be scanned once. The best-case scenario is that there is an index on $T.\boldsymbol{X}$ and $T'.\boldsymbol{Y}$, and the triggers may be able to operate using index-only scans.

HMerge Translation of Tier 2 FK. Let $\mathsf{U}_{\mathbf{HM}}$ be the update function for the transformation $HMerge(f, \overline{T}, \overline{C})$, and let s be its virtual schema. Define the action of $\mathsf{U}_{\mathbf{HM}}$ on a Tier 1 or 2 foreign key as follows:

$\mathsf{U}_{\mathbf{HM}}(s, \mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})) =$
If $T \models f$ and $T' \not\models f$, then
$\quad \mathbf{FK}(\boldsymbol{F} \wedge (\overline{C} = T)|\overline{T}.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})$
Else, if $T \not\models f$ and $T' \models f$, then
$\quad \mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G} \wedge (\overline{C} = T')|\overline{T}.\boldsymbol{Y})$
Else, if $T \models f$ and $T' \models f$, then
$\quad \mathbf{FK}(\boldsymbol{F} \wedge (\overline{C} = T)|\overline{T}.\boldsymbol{X} \rightarrow \boldsymbol{G} \wedge (\overline{C} = T')|\overline{T}.\boldsymbol{Y})$
Else, $\mathbf{FK}(\boldsymbol{F}|\overline{T}.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})$

This result follows from query translation — one can translate the fragment into its Tier 3 equivalent, translate the two constituent queries through $\mathsf{Q}_{\mathbf{HM}}$, then translate the result back to an equivalent Tier 2 fragment to arrive at the result above. Note that the translation of a Tier 2 FK through a Horizontal Merge results in a Tier 2 foreign key.

Pivot Translation of Tier 2 FK. Let $\mathsf{U}_{\mathbf{PV}}$ be the update function for the transformation $Pivot(T_p, A, V, \overline{T_p})$, and let s be its virtual schema. The action of $\mathsf{U}_{\mathbf{PV}}$ has several cases based on the tables, columns, and conditions in a Tier 1 or 2 foreign key definition; for brevity, here are two of the interesting cases:

Case 1: $T = T_p$, $T' \neq T_p$, and $A \in \boldsymbol{X}$. One of the source columns is pivoted — this is the case demonstrated in Figure 12).
$\mathsf{U}_{\mathbf{PV}}(s, \mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})) =$
$\quad \mathbf{Check}(\pi_{\boldsymbol{X}}\sigma_{\boldsymbol{F}} \not\boxtimes_{\mathbf{Cols}(T_p)-\mathbf{Keys}(T_p);A;V} \overline{T_p}, \pi_{\boldsymbol{Y}}\sigma_{\boldsymbol{G}}T')$
Figure 12(c) is such a case, where the target of the foreign key references the pivot attribute column, so a **Check** statement is needed to describe the integrity constraint over the logical schema.

Case 2: $T = T_p$, $T' \neq T_p$, $\exists_{(c=v)\in \boldsymbol{F}} c = A$, $V \in \boldsymbol{X}$ and $A \notin \boldsymbol{X}$. The source table is pivoted, there is a condition on the pivot attribute column, and the value column V participates in the foreign key.

$$\mathsf{U}_{\mathsf{PV}}(s, \mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})) = \\ \mathbf{FK}(\boldsymbol{F} - \{(c = v)\}|\overline{T_p}.(\boldsymbol{X} - \{V\} \cup \{v\}) \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})$$

The result is a single FK involving only one pivoted column v in the source table, matching the original condition on column A.

5 Business Logic and Channel Transformations

To fully support the property of indistinguishability, a channel must encapsulate all of the data and query transformations that occur between an application's virtual schema and its physical schema. Such transformations may include business logic that is typically found in data access layers or stored procedures. While the restructuring CT's in Section 4 are defined on materialized instances, other kinds of business logic may be non-deterministic.

For instance, consider an application designer who would like all tables in the database to include a column whose value for a given row is the user that last edited the data in that row. The application does not display that data, but reports may be run over the database to look for patterns in data usage based on user. Such a transformation can still be defined as a CT, even considering the non-determinism of the operation. Such *business logic transformations* include:

- $Adorn(T, e, \overline{\boldsymbol{A}}, \boldsymbol{C})$ adds columns $\overline{\boldsymbol{A}}$ to table T. The columns hold the output of function e, which returns the state of environment variables. Values in $\overline{\boldsymbol{A}}$ are initialized with the current value of e on insert, and refreshed on update whenever any values in columns $\boldsymbol{C}$ change.
- $Trace(\boldsymbol{T}, \overline{T})$ records all operations to any tables in the set $\boldsymbol{T}$ and records them to table $\overline{T}$. For instance, given an delete operation to table T_1, $Trace$ would insert into table $\overline{T}$ a row containing a textual representation of the statement as well as the time of the operation and the current user. The CT could be further parameterized to include the set of statements to monitor, the exact names of the columns in the table $\overline{T}$, etc.
- $Audit(T, \overline{B}, \overline{E})$ adds columns $\overline{B}$ and $\overline{E}$ to table T, corresponding to a lifespan (i.e., valid time) for each tuple. Rows inserted at time t have $(\overline{B}, \overline{E})$ set to $(t, null)$. For rows deleted at t, set $\overline{E} = t$. For updates at time t, clone the row; set $\overline{E} = t$ for the old row, and set $(\overline{B}, \overline{E}) = (t, null)$ for the new row. The virtual schema instance corresponds to all rows from the output database where $\overline{E} = null$.

5.1 Business Logic Transformations and Correctness

As noted in Section 3.1, channel transformations are ordinarily defined in terms of four functions corresponding to their effect on schemas, instances, queries, and updates. Recall that the instance function I is never implemented, but serves as part of the semantic definition of the transformation. For all of the restructuring transformations listed in Section 4, the function I makes sense because each restructuring CT is fully deterministic. Thus its effect on a fully-materialized instance is the most logical way to think about its definition.

For a Business Logic CT (BLCT), defining such a transformation is not as important given that its definition is non-deterministic. Said another way, whereas the most natural way to describe the operation of a restructuring CT is to describe its operation on instances, the most natural way to describe the operation of business logic CT's is to describe its operation on updates. Note that the descriptions of Adorn and Audit in the introduction to this section are even described in terms of how they operate on updates.

BLCT's are still defined as a tuple of functions S, I, Q, and U. BLCT's must still satisfy the correctness properties in Figure 5. The instance function I is still defined for a BLCT, but its output may contain "placeholders" that would be filled in with the current value of some variable or environment at the time of execution. However, where the restructuring CT's may use the properties in Figure 5 to infer an update transformation from an instance transformation, one may use Figure 5 to infer I from U by defining I as if it were a sequence of insert statements inserting all of its rows.

5.2 Example Transformation: Audit

The *Audit* transformation is primarily a way to ensure that no data in a database is ever overwritten or deleted. As a motivating application of the transformation, consider an application where the data in the database must be audited over time to detect errors that may have been overwritten, or to allow data recovery in the event that data is accidentally overwritten or deleted. This operation has also been called *soft delete* in database literature [1].

Audit is in many ways a way to compensate for the lack of first-class temporal support in database systems. There have been many extensions to relational systems that add temporal features (e.g., [24,35]). Such systems maintain a queryable history of the data in the database, and therefore allow users to write queries that ask questions like, "what would have been the answer to query Q as of time T?", or "what is the result of this aggregate operation over time?" However, since such systems have typically stayed in the research realm and because some applications' needs in the temporal area are fairly small, many applications merely implement some of the temporal functionality as part of their data access layer. The Audit transformation serves this same purpose.

The rest of this section discusses the Audit transformation in depth in two ways. First, the section provides a detailed discussion of a simple version that meets the transformation's basic requirements as an insulator from DML update and delete statements. Second, the section provides insights into how to construct a more complete version of the transformation that may go beyond the needs of many applications but covers more advanced scenarios.

5.3 Translating Schema

As mentioned in the description of the CT, Audit requires two additional columns to be added to any audited table. Each row in the physical schema will have a value for column $\overline{B}$ indicating when the row comes into existence, and a (possibly null) value for column $\overline{E}$ indicating when the row is no longer valid. Temporally speaking, the two

values for $\overline{B}$ and $\overline{E}$ put together form an interval closed on the left and open on the right, and possibly unbounded on the right if $\overline{E} = \mathit{null}$, representing the case where the row is still current.

In addition to the schema translation S adding two extra columns, it also adds column $\overline{B}$ to the primary key of the table. The need for an altered primary key is motivated by the fact that a single row with primary key K may map to multiple rows in the physical schema. Each of those rows, however, must have a unique value for $\overline{B}$, however, since each row must be in existence at different points in time.

A simplified version of Audit may stop here, as it provides the proper client-side effects. Specifically, if the only access to the database is through the channel, no additional modifications are necessary to the schema. However, it is important to note at this stage is that merely adding $\overline{B}$ to the schema of the output table does not prevent a malicious or unknowing user to create a database instance in the physical schema that is nonsensical. Consider, for example, the following two tuples:

$$[A = 10, Z = \text{Bob}, \overline{B} = 5, \overline{E} = 10]$$
$$[A = 10, Z = \text{Alice}, \overline{B} = 7, \overline{E} = 12]$$

Treating the values for $\overline{B}$ and $\overline{E}$ as intervals, the two tuples above have overlapping lifespans for the row with key value $A = 10$, implying that at a given point in time, two tuples with the same key existed. Similarly, consider the following pair of tuples:

$$[A = 10, Z = \text{Bob}, \overline{B} = 5, \overline{E} = \mathit{null}]$$
$$[A = 10, Z = \text{Alice}, \overline{B} = 7, \overline{E} = \mathit{null}]$$

For this pair, the key value $A = 10$ corresponds to two different "active" rows, which violates our invariants. Finally, consider this tuple:

$$[A = 10, Z = \text{Alice}, \overline{B} = 7, \overline{E} = 3]$$

This tuple is invalid as it has ended before it begun. Such constraints in a relational database are indeed handled by true temporal databases, but are not enforced in a relational database without triggers, such as the following example, where "PK" is the set of key columns not including lifespan origin point $\overline{B}$:

```
begin insert trigger (T)
  if (new tuple[E] is not null and new tuple[E] <= new tuple[B])
    reject insert
  for each tuple t in T
    if t[PK] = new tuple[PK]
      if new tuple[E] is null and t[E] is null
        reject insert
      else if new tuple[E] is null and t[E] is not null
        if new tuple[B] < t[E]
          reject insert
      else if new tuple[E] is not null and t[E] is null
        if t[B] < new tuple[E]
          reject insert
```

```
      else
        if not (t[E] <= new tuple[B] or new tuple[E] <= t[B])
          reject insert
  accept insert
end trigger
(update trigger follows same pattern as insert)
```

This kind of trigger cannot be mimicked by any of the constraints considered in Table 1. Therefore, to account for this kind of constraint, one would need to add a new kind of constraint to Table 1, in which case all CT's would need to "learn" how to translate such constraints, and one would need a set of constraints that is closed under all CT's. Given the difficulty and computational complexity of this constraint, most applications that employ an Audit transformation opt not to include the constraint and rely on the application and data access layers to produce correct data.[3]

5.4 Translating DML Statements

True to the "nothing deleted" spirit of the transformation, the most interesting operation of the Audit CT is on update statements. Informally, the update function U has the following effect on DML statements:

- For insert statements, add the current system time as the value for the "begin" timestamp column $\overline{B}$.
- For delete statements, change the statement to an update statement that sets the "end" timestamp column $\overline{E}$ to the current system time for all affected rows.
- For update statements, take all affected rows and clone them, where the new cloned row has a current timestamp for column $\overline{B}$. Then, set column $\overline{E}$ for the old rows to the current time as well.

Formally, the function U can be defined as follows:

$$
\begin{aligned}
\mathsf{U}_{\mathbf{Audit}}(T, \mathbf{I}(t, \boldsymbol{C}, Q)) &= I(T, \boldsymbol{C} \cup \{\overline{B}, \overline{E}\}, Q \times \{Now(), null\}) \\
\mathsf{U}_{\mathbf{Audit}}(T, \mathbf{D}(t, \boldsymbol{F})) &= Loop(t, \sigma_{\boldsymbol{F} \wedge \overline{E}=null} T, \{ \\
&\qquad U(T, \forall_{c \in \mathbf{Keys}(\overline{T})} c = t[c], \{\overline{E}\}, \{Now()\})\}) \\
\mathsf{U}_{\mathbf{Audit}}(T, \mathbf{U}(t, \boldsymbol{F}, \boldsymbol{C}, Q)) &= Loop(t, \sigma_{\boldsymbol{F} \wedge \overline{E}=null} T, \{ \\
&\qquad U(T, \forall_{c \in \mathbf{Keys}(\overline{T})} c = t[c], \{\overline{E}\}, \{Now()\}), \\
&\qquad I(T, \mathbf{Cols}(T), \pi_{\mathbf{Cols}(T)-\{\overline{B},\overline{E}\}} t \times \{Now(), null\})\}
\end{aligned}
$$

In the above, $Now()$ represents the current timestamp at the time of execution. Examples of the action of an Audit CT on an insert statement can be found in Figure 13(a), where Figure 13(b) shows the same CT operating on an update and a delete.

[3] This pattern of relying on correct application and data access behavior is a common one, and will be revisited in the section on object-relational mappings.

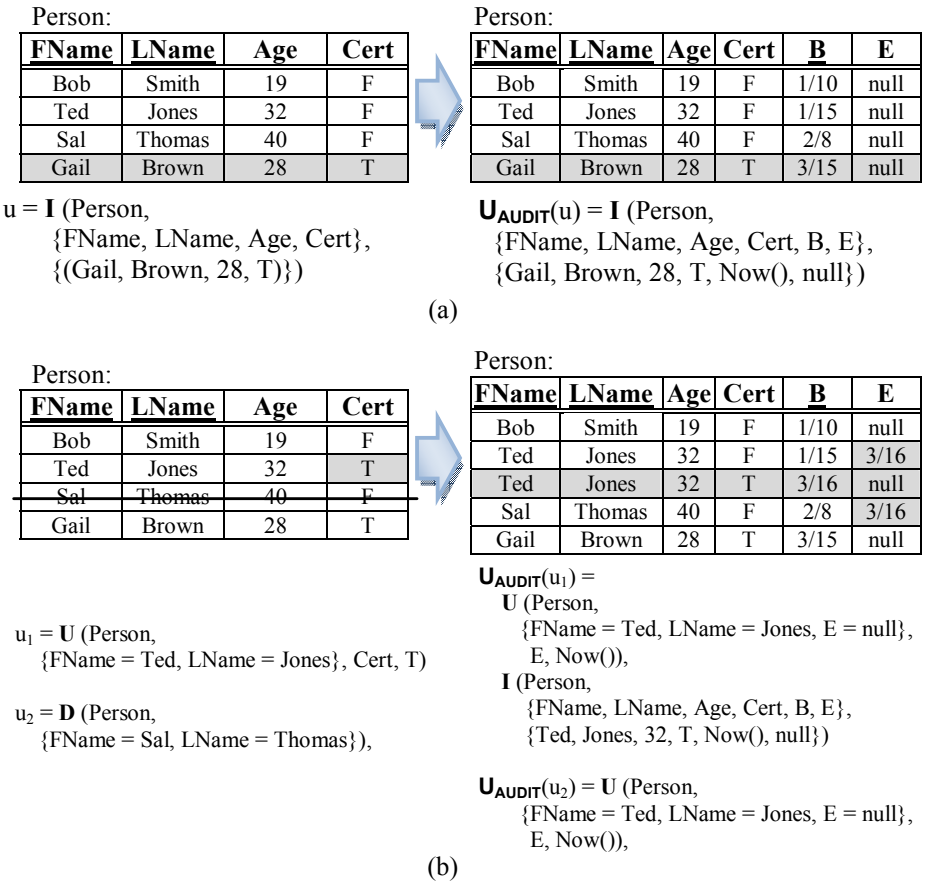


Fig. 13. The action of the Audit transformation on an insert operation (a) as well as an update and a delete operation (b)

5.5 Translating Instances and Queries

The instance transformation $\mathsf{I}_{\mathbf{Audit}}$ is defined as discussed above, where $\mathsf{I}_{\mathbf{Audit}}$ processes the instance as if it were a sequence of instance statements. In this case, that means appending to each row in the instance of table T the value *Now*() for $\overline{B}$ and *null* for $\overline{E}$.

Query translation for the Audit CT is straightforward. If T is the relational algebra query for all data in audited table T, then:

$$\mathsf{Q}_{\mathbf{Audit}}(T) = \pi \boldsymbol{C} \sigma_{\overline{E}=null} T$$

where $\boldsymbol{C}$ in this case is the set of columns of T on the transformation's input side. In essence, a query for any data in T will return only the non-deprecated rows in the data store, and only the columns that are visible in the input. Note here that it is trivial to prove the property in Figure 5(a), since $\mathsf{Q}_{\mathbf{Audit}}$ effectively shears off the columns that are added by $\mathsf{I}_{\mathbf{Audit}}$.

5.6 Translating DDL Statements

The Audit transformation as described in this section is simplified in that its action softens only the impact of DML statements. This definition may be sufficient for many applications; however, this section discusses what changes would be necessary to modify the definition and operation of the Audit transformation to account for DDL transformations as well. Such handling would allow the user to, for instance, drop a column from the virtual schema but retain that column in the physical schema.

Even the simplified version of the Audit transformation, before taking into account any schema modification handling, must make two accommodations for DDL statements:

- Translates an Add Table statement to add the two temporal "endpoint" columns to the table's definition, where the starting point is also added to the table's primary key
- Checks to make sure that an Add Column statement does not collide with either endpoint column, and if so, return an error

All other table, column, and element DDL statements would pass through unaltered for the simple version of Audit. However, consider the implications of taking the "never lose data" property to include schema modifications:

- When dropping a table, the table itself must somehow be deprecated. The table has been dropped from the virtual schema, so the application is no longer aware of it. However, Audit must define some behavior for the case where the application tries to add a new table with the same name and a possibly different schema.
- When dropping a column, the column must somehow be deprecated. Just as with dropping a table, the column disappears from the application schema and is thus invisible, but must be insulated from being overwritten by a subsequent add column call.
- When dropping an element, the domain element must somehow be deprecated. The semantics for dropped elements can be preserved here, where for key columns, treat as if it were a deleted row for a key column and an update to null for a non-key column. To allow CT's later in the channel to preserve the semantics, Audit would need to change the drop element operation to Update and Delete statements, since the element itself would not be able to be dropped (or else the original value would be lost).
- Audit would also need to maintain a policy for renamed elements. Audit could either allow the operation to complete without change, or alternatively treat the operation like it treats a DML update, keeping both the old and the new elements and "deprecating" the old ones.

The full version of Audit would then need to maintain a list of all deleted or renamed schema artifacts, implying that those would need to be added to the parameters of the transformation. The physical schema of the CT would always be the union of all of the schemas that had appeared over time. True temporal systems would be able to manage multiple versions of schemas internally rather than require a single unified schema.

5.7 Translating Foreign Keys

Section 4.4 defines an approach where foreign key translation follows directly from query translation. With the simple form of Audit, the same logic applies. Consider translation of a foreign key statement $\mathbf{FK}(\boldsymbol{F}|T.\boldsymbol{X} \rightarrow \boldsymbol{G}|T'.\boldsymbol{Y})$, where the table T' is run through an Audit CT. Simply by running query translation $\mathsf{Q}_{\mathsf{Audit}}$ and simplifying the resulting relational algebra, the constraint becomes $\mathbf{Check}(\sigma_{\boldsymbol{F}}\pi_{\boldsymbol{X}}T \subseteq \sigma_{\boldsymbol{G}}\pi_{\boldsymbol{X}}\sigma_{\overline{E}=null}T')$. This statement essentially says that affected tuples in T must match against "current", non-deprecated tuples in T'. Said another way, if one only considers non-deprecated tuples, the foreign key statement is still enforced as before.

Just as in Section 5.3, the result of the above foreign key translation keeps foreign keys enforced for all operations on current data, but leaves historical data unenforced. Additional constraints would be necessary to enforce constraints on the historical data as well, which would mean that foreign keys would need additional processing beyond what is inferred from just query translation.

6 Object-Relational Mappings

The previous sections describe a methodology to handle relational-to-relational transformations. However, in the typical application development scenario, the client development model and data schema is object-oriented[4]. It requires some method to transform the object-oriented client schema into a relational one that meets the same requirements as the relational-only transformations. There are two noteworthy solution strategies in the design space:

1. Construct a language of transformations where each individual transformation describes an incremental change of a specific object-oriented artifact into a relational one. For instance, a transformation may dictate that a given collection-valued property should be mapped to a given table with a foreign key relationship.
2. Construct a single transformation whose logic is powerful enough to dictate all of the details about how to translate an entire object-oriented schema into a relational one.

Each option has trade-offs. For instance, option 1 above is well-documented and studied as part of the DB-MAIN project [18]. DB-MAIN as a methodology allows the data developer to dictate the transformation procedure from one model to another one step at a time, incrementally translating artifacts in the first model into artifacts in the second. The translation from object-oriented to relational schemas is a supported special case of that framework.

Option 2 above is well-implemented in tool support from object-relational mapping tools. Object-relational mapping tools (ORMs) have become a central fixture in application programming over relational databases as a way to bridge the gap between the two data models. Such tools typically have a single indivisible declarative mapping layer

[4] We assume that the object-oriented data model is in particular the Entity Data Model [7], which is similar to the Entity Relationship model extended with generalization hierarchies.

that handles all translations from object classes to relations in a single declarative pass. The ORM uses this mapping to translate queries and updates against the model into semantically-equivalent ones against the relational database.

A key differentiator between these two options is which design dimension holds more complexity. For option 1, each individual transformation is small and composable in nature, and thus easy to understand. However, because single transformations are not expected to bring an entire schema from one model to another, schemas on both the input and output of a transformation may not conform to a single model. DB-MAIN compensates for this situation by formally treating each transformation as a mapping to and from a single unified model that contains the constructs from all relevant source and target schemas. As a result, for a DB-MAIN-style transformation to fit into the channels framework, it would need to be defined on the full set of incremental transformations over the generalized model.

For option 2, the design complexity lies in the transformation operation itself. The system only requires as input the set of incremental transformations over an object-oriented schema. With a single transformation being assigned the monolithic task of the full model translation, the difficulty becomes determining how that complex and highly configurable transformation reacts to schema evolution inputs.

The rest of this part of the paper describes a technique that makes option 2 tractable to fit in the channels framework. Given incremental schema evolution primitives, an object-relational mapper can automatically modify itself and its physical schema.

6.1 Query and Update Statements

Table 2 shows a list of transformations similar to that of Table 1, only the statements represent DML and DDL actions in an object-oriented environment. Recall that the operations in Table 1 have a closure property — any CT that transforms a relational schema into another relational schema must be closed with respect to that set of statements under its update transformation U. An ORM modeled as a CT must be closed in the sense that it must translate statements from Table 2 into statements from Table 1. Table 2 is also complete in the same sense as Table 1 is complete, where one can get from any schema to any other schema incrementally using add and drop statements, with the other statements serving as shorter and better instance-preserving shortcuts [5] (e.g., Move Property, which could be modeled as a drop followed by an add).

There is one additional catch to be noted when considering an ORM as a CT, and that is query translation. The query language for a relational CT is simply relational algebra, and thus can be modeled as SQL. The output query language from an ORM is indeed SQL or relational algebra, but the input language must be a query language over objects. There are many possible extensions to SQL that can allow it to handle querying over object data; this paper assumes that the query language is *Entity SQL* or *ESQL*, a language that accompanies the Microsoft Entity Framework [28].

ESQL has the property that it can be translated through an ORM into traditional SQL whenever the ORM exhibits a valid mapping, but with one catch. Because ESQL is a query language over objects, it is allowed to return data as objects and also construct objects on the fly. Traditional SQL lacks that capability. As such, whenever an input query requires an object constructor, the Entity Framework translates the query into

SQL that includes additional columns that encode how to construct the objects when the objects are returned. For instance, consider the following simple ESQL query over the example in Figure 1:

```
SELECT p FROM Person
```

The set of all `Person` objects is a polymorphic set that includes patients and staff, and the query itself returns objects. The generated SQL will include extra constant columns that dictate for each row what kind of object it is, and when the data is returned, the Entity Framework will construct objects of the correct type for each row[5]. This paper assumes that query translation proceeds in that fashion.

6.2 The Impedance Mismatch and Schema Evolution

The primary goal of any ORM is to overcome the *impedance mismatch*, the difference in data models that must be overcome to allow full-fidelity communication between objects and relations. Therefore, a primary feature of an ORM is to map object-oriented constructs to relational ones. For any given construct that requires mapping, there are often several ways to accomplish the task. For instance, given a class hierarchy, there are three basic ways to map that hierarchy to relations:

- *Table-per-Type* (TPT), in which each type in the hierarchy is assigned its own table containing only the non-inherited properties of that type.
- *Table-per-Concrete Class* (TPC), in which each non-abstract type in the hierarchy is assigned its own table with all properties of that type, inherited or otherwise.
- *Table-per-Hierarchy* (TPH), in which the entire hierarchy is mapped to a single table.

Examples of each of these mapping scheme can be found later in this section[6].

Mapping schemes may also be mixed and matched in the same hierarchy. In that case, handling schema evolution becomes non-trivial. Consider an application with a model as shown in Figure 14. A new application version may require that new types be added to the type hierarchy, including the three shown in the figure. With ORM tools, these conceptual model artifacts must map to persistent storage. Thus, one must determine what the mapping should be and what changes should be made to the physical schema.

If the entire hierarchy in Figure 14 is mapped to storage using a consistent pattern — for instance, by mapping the entire hierarchy to a single table (as Ruby on Rails does [34]) or instead by mapping each type to its own table — then it is natural to map the new type using that same pattern (regardless of where in the hierarchy of Figure 14 one chooses to add the type). However, for more complex mappings, especially ones that do not employ a uniform mapping pattern, the answer is more complicated. In Figure 14, the choice of mapping and physical storage may differ for each of the three locations for adding a new type.

To treat an ORM as a channel transformation, the correct means for propagating a schema change from a conceptual model to a physical database must be determined by

[5] More examples may be found in Entity Framework literature (e.g., [28]).

[6] A more complete treatment of the options for mapping objects to relations may be found in [13], among other places.

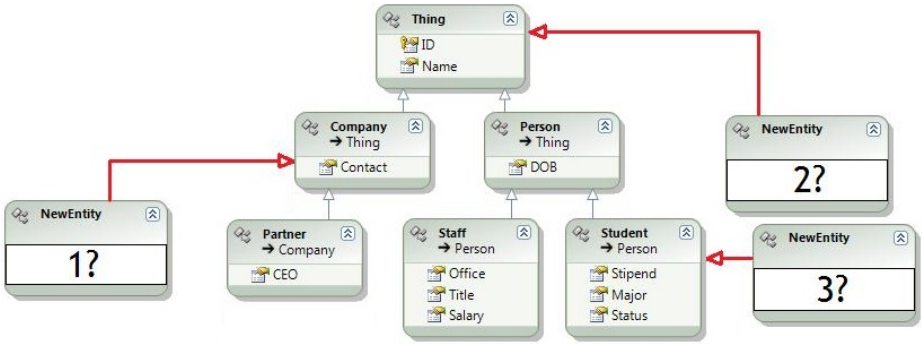


Fig. 14. A type hierarchy with three locations for adding a new type to the hierarchy

using the existing mapping to guide future incremental changes, even when the mapping scheme is not uniform across a hierarchy. If there is a consistent pattern in the immediate vicinity of the change, then that pattern is preserved after the change. As a special case, if an entire hierarchy is mapped using a single scheme, then it is mapped using that scheme for new artifacts. Given a list of incremental conceptual model changes and the previous version of the model and mapping:

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

There are many ORM tools available today, including TopLink [31], Hibernate [20], Ruby on Rails [34], and Entity Framework (EF) [7]. Each of these tools has different syntax and expressive power for mapping conceptual models to databases, but none provides a method for propagating conceptual model changes into mapping or store changes. In the research project MeDEA, given an incremental change to a conceptual model, a developer chooses a rule to guide mapping evolution [12]. However, the new mapping need not be consistent with the existing mapping; if such consistency is important, the developer must maintain intricate knowledge of the mapping. By contrast, the technique presented here provides the developer with an automated model design experience with changed artifacts mapped consistently with existing artifacts. Therefore, the requirements of using a ORM as a CT are fulfilled, as it can faithfully translate schema evolution primitives without requiring intervention.

One final note about mappings and constraints. As with the Audit transformation from Section 5, object-relational mappers are very good at ensuring a consistent client-side experience. However, they often fall short (intentionally) in enforcing constraints on the database that ensure that direct access to the database does not violate any assumptions. Each of the three mapping paradigms — Table per Type, Table per Concrete Class, and Table per Hierarchy — has such assumptions that could be expressed using

triggers but are often costly to enforce. For example, if the class hierarchies on the object side of the mapping do not allow multiple inheritance:

- For TPT mappings, if table T and table T' represent types in the object space that are siblings in the class hierarchy, then there must not be any overlapping keys in common between T and T'.
- For TPC mappings, if $\boldsymbol{T}$ represents the set of all tables that map to a given class hierarchy, the union of the primary keys from all tables in $\boldsymbol{T}$ must be a distinct set.
- For TPH mappings, there must be a mutual exclusion property enforced. If a table T represents an entire class hierarchy, given a row r in the table, $r[c]$ must be null for any column c that does not map to a property of the class to which r's instance belongs.
- Foreign key mappings may become incomplete in translation. For instance, consider the association between types `Procedure` and `Patient` in Figure 1. Because the `Person` hierarchy is mapped TPC, it happens that the association becomes a foreign key in Figure 2. However, if `Person` were instead mapped as TPH, the association would become a foreign key that would only point to *part* of the table that stores all `Person` instances. Note that in this situation, Tier 2 or 3 foreign keys are expressive enough to model the relationship.

Most object-relational mapping systems consider these limitations to be acceptable. However, just as the statement $Check(Q1 \subseteq Q2)$ formalizes a kind of constraint that can be expressed as a trigger, so too could the necessary constraints for full fidelity of these mapping be formalized. For example, the needed constraint for full fidelity of TPC mappings could be expressed as a statement $Disjoint(\boldsymbol{Q})$, which takes a set of queries and ensures that their results do not overlap. Such a statement could be processed through a CT using the query pipeline in much the same way as *Check* statements. This paper leaves such extensions as future work.

6.3 The Mapping Relation

Given the conceptual model from Figure 14, consider a mapping that combines several schemes for the hierarchy, as in Figure 15. The mapping between the conceptual model (a) and its physical storage (b) has the following characteristics:

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

Table 2. Object-oriented DML and DDL statements supported by channel transformations

Statement	Formalism	Explanation of Variables
Insert	$\mathbf{I}(T, Q)$	Insert objects into entity set T from the result of Q. The value of Q may be a constant or a query result.
Update	$\mathbf{U}(T, \boldsymbol{F}, \boldsymbol{C}, Q)$	Update objects in entity set T that satisfy all equality conditions $\boldsymbol{F}$ specified on key properties. Non-key properties $\boldsymbol{C}$ hold the new values specified by query or constant Q. Query Q may refer to the pre-update object values as constants. Not all key properties need to have a condition.
Delete	$\mathbf{D}(T, \boldsymbol{F})$	Delete objects from entity set T that satisfy all equality conditions $\boldsymbol{F}$ specified on key properties. Not all key properties need to have a condition.
Add Simple Type	$\mathbf{AT}(T, \boldsymbol{P}, \boldsymbol{D}, A)$	Add new type T without a key, whose properties $\boldsymbol{P}$ have domains $\boldsymbol{D}$. Parameter A is a Boolean value indicating whether the type is abstract.
Add Identifiable Type	$\mathbf{AT}(T, \boldsymbol{P}, \boldsymbol{D}, \boldsymbol{K}, A)$	Add new type T, whose properties $\boldsymbol{P}$ have domains $\boldsymbol{D}$, with key properties $\boldsymbol{K} \subseteq \boldsymbol{P}$. Parameter A is a Boolean value indicating whether the type is abstract.
Add Derived Type	$\mathbf{AT}(T, T', \boldsymbol{P}, \boldsymbol{D}, A)$	Add new type T, derived from T', whose new (non-inherited) properties $\boldsymbol{P}$ have domains $\boldsymbol{D}$. Domains may include references to other types. Parameter A is a Boolean value indicating whether the type is abstract.
Rename Type	$\mathbf{RT}(T_o, T_n)$	Rename type T_o to be named T_n. Throw error if T_n already exists.
Change Type Abstraction	$\mathbf{CT}(T, A)$	Change type T to be abstract if $A = true$, concrete otherwise.
Drop Type	$\mathbf{DT}(T)$	Drop the type named T. Throw an error if type T has any derived types (only drop if the type is a child type).
Add Property	$\mathbf{AP}(T, P, D)$	Add to type T a property named P with domain D.
Rename Property	$\mathbf{RP}(T, P_o, P_n)$	In type T, rename the property P_o to be named P_n. Throw error if P_n already exists.
Move Property	$\mathbf{MP}(T, T', P)$	Move the non-key property P from type T to type T', provided that T is either a descendant or an ancestor of T'. Any instance that has property P both before and after the move will not lose data.
Change Property Facet	$\mathbf{CP}(T, P, F, V)$	For the property P of the type T, change its domain facet F to have the value V.
Drop Property	$\mathbf{DP}(T, P)$	In type T, drop the non-key property P.
Add Element	$\mathbf{AE}(T, P, E)$	In type T, in property P, add a new possible domain value E.
Rename Element	$\mathbf{RE}(T, P, E_o, E_n)$	In type T, in property P, rename domain element E_o to be named E_n. Throw error if E_n conflicts with an existing element.
Drop Element	$\mathbf{DE}(T, P, E)$	In type T, in property C, drop the element E from the domain of possible values.
Add Association	$\mathbf{AA}(A, \boldsymbol{F}\|T \leftrightarrow_C \boldsymbol{G}\|T')$	Add an association type named A between types T and T', where the association is limited by conditions $\boldsymbol{F}$ on type T and by conditions $\boldsymbol{G}$ on type T'. The association has cardinality C, which is one of the following options: one-to-one, zero-or-one-to-one, one-to-many, zero-or-one-to-many, or many-to-many.
Rename Association	$\mathbf{RA}(A, A')$	Renames an association A to A'. Throws an error if there is already an association or type named A'.
Change Association Cardinality	$\mathbf{CA}(A, C)$	Changes the cardinality of the association A to be C.
Drop Association	$\mathbf{DA}(A)$	Drops the association named A.
Loop	$\mathbf{Loop}(t, Q, \boldsymbol{S})$	For each object t returned by query Q, execute transaction $\boldsymbol{S}$.
Error	$\mathbf{Error}(Q)$	Execute query Q, and raise an error if the result is non-empty.

For this mapping, there is no single mapping scheme for the entire hierarchy. However, one can make some observations about the three different locations from Figure 14, specifically regarding the types that are "nearby":

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

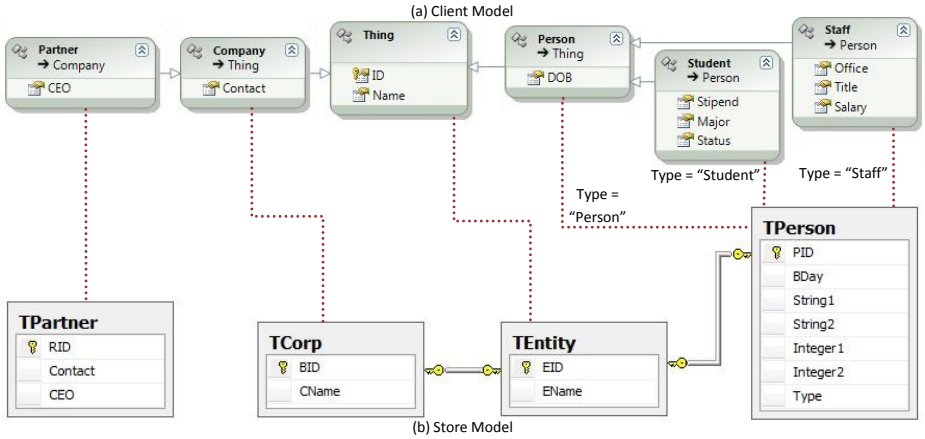


Fig. 15. Example client (a) and store (b) models, with a mapping between them

Using this informal reasoning, one can argue that types added at locations 1, 2, and 3 should be mapped using TPC, TPT, and TPH respectively. Formally speaking, we have two challenges to overcome. First, we need a definition of what it means to be "nearby" in a hierarchy. We address this in Section 7. Second, while TPC, TPT, and TPH are well-known and well-understood concepts to developers, they may not be expressed directly in the mapping language, so some analysis is needed to recognize them in a mapping. Moreover, finer-grained mapping notions like column mapping are not present in any available ORM.

Different object-relational mapping tools have different methods of expressing mappings. We assume in our work and our running example that the mappings are specified using EF, whose mappings need to be analyzed to identify mapping schemes like TPT, TPC, or TPH. In EF, a mapping is a collection of mapping fragments, each of which is an equation between select-project queries. Each fragment takes the form $\pi_{\boldsymbol{P}}\sigma_\theta E = \pi_{\boldsymbol{C}}\sigma_{\theta'} T$, where $\boldsymbol{P}$ is a set of properties of client-side entity E, $\boldsymbol{C}$ is a set of columns of table T, and θ and θ' are conditions over E and T respectively. Conditions θ and θ' may be of the form $c = v$ for column or property c and value v, c IS NULL, c IS NOT NULL, type tests IS T or IS ONLY T for type T, or conjunctions of such conditions.[7]

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

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

[7] The formal specification in [28] allows disjunction in conditions. This paper considers only conjunction because it simplifies exposition and because EF as implemented only allows conjunction. The techniques described in this paper are applicable when disjunction is allowed, with some adaptation.

A mapping relation is a pivoted form of mapping, where each row represents a property-to-property mapping for a given set of conditions. As an example, consider the model pair and mapping shown in Figure 15. One can express the mapping in the figure using Entity Framework as follows:

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

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

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

To translate an entire EF mapping to a mapping relation instance, one performs the above translation to each constituent mapping fragment. Table 3 shows the mapping relation for the models and mapping in Figure 15.

Table 3. The mapping relation for the models and mapping in Figure 15 (column *CC* not shown, since the mapping has no client conditions)

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

The rows in the mapping relation do not need to maintain `IS` or `IS ONLY` conditions because they are intrinsic in the mapping relation representation. The `IS` condition is satisfied by any instance of the specified type, while the `IS ONLY` condition is only satisfied by an instance of the type that is not also an instance of any derived type. In the mapping relation, the `IS` condition is represented by rows in the relation where non-key entity properties have exactly one represented row (e.g., `Thing.Name` in Table 3). The `IS ONLY` condition is represented by properties that are mapped both by the declared type and by derived types (e.g., `Company.Contact` and `Partner.Contact` in Table 3).

7 Similarity and Local Scope

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

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

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

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

Once the above steps have been completed, every type in the hierarchy will be assigned a pair, like those shown in Figure 16. The *priority score* $\mathcal{P}(E, E_0)$ for an entity type E in a hierarchy relative to E_0 is computed from its pair (m, n) as $\mathcal{P}(E, E_0) = 1 + m - 2^{-n}$. The priority score imposes a partial ordering on entity types, as indicated by the second (lower) number on each node in Figure 16.

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

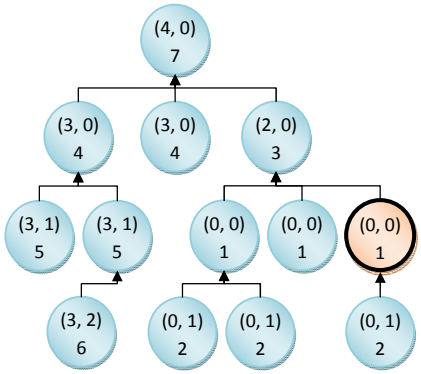


Fig. 16. An example of similarity pairs and the numeric order of hierarchy nodes relative to a given node (in bold outline)

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

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

7.1 Mapping Patterns

Using the mapping relation and local scope function Φ, we can use the mapping itself as data to mine the various mapping schemes. A *mapping pattern* is a query Q^+ that probes for the existence of the requested mapping scheme and returns either `true` or `false`. The first set of patterns search for one of the three prominent hierarchy mapping schemes mentioned in Section 6.3, given a local scope $\Phi(E)$ for an entity type E:

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

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

Table-per-Type (TPT): Given an entity type E and a child type E', map them to tables T and T' respectively, with properties of E mapped to T and properties of E' not present in E mapped to T'. Given local scope $\Phi(E)$, we define the TPT pattern as a query that tests for two properties. First, any pair of entity types in scope will have non-overlapping sets of mapped tables. Second, if $\mathcal{A}$ is the least common ancestor of all entity types in $\Phi(E)$, then for each entity type in scope that is not the common ancestor, the non-key

properties of $\mathcal{A}$ are not re-mapped (i.e., there are no matching rows in the mapping relation):

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

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

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

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

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

Remap by column name (RBC): If types E and E' are cousin types in a hierarchy[8], and both E and E' have a property named P with the same domain, then $E.P$ and $E'.P$ are mapped to the same store column. This scheme maps all properties with like names to the same column, and is the scheme that Ruby on Rails uses by convention [34]. Given hierarchy table T, the RBC pattern is:

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

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

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

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

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

$$\wedge(\forall_{X \in \pi_D \sigma_{ST=T \wedge \neg K} \mathcal{M}} \exists_{E \in \pi_{CE} \sigma_{ST=T} \mathcal{M}} |\pi_{CP} \sigma_{CE=E \wedge ST=T \wedge D=X \wedge \neg K} \mathcal{M}|$$

$$= |\pi_{SC} \sigma_{ST=T \wedge D=X \wedge \neg K} \mathcal{M}|).$$

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

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

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

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

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

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

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

Strictly speaking, we do not need patterns for these final two schemes — the algorithm for generating new mapping relation rows (Section 7.2) carries such schemes forward automatically. Other similar schemes include vertical partitioning and merging, determining whether a TPH hierarchy uses a discriminator column (as opposed to patterns of `NULL` and `NOT NULL` conditions), and association inlining (i.e., whether one-to-one and one-to-many relationships are represented as foreign key columns on the tables themselves or in separate tables).

Note that each group of patterns is not complete on its own. The local scope of an entity may be too small to find a consistent pattern or may not yield a consistent pattern (e.g., one sibling is mapped TPH, while another is mapped TPC). In our experience, the developer is most likely to encounter this situation during bootstrapping, when the client model is first being built. Most mappings we see are totally homogeneous, with entire models following the same scheme. Nearly all the rest are consistent in their local scope (specifically, all siblings are mapped identically). However, for completeness in our implementation, we have chosen the following heuristics for the rare case when consistency is not present: If we do not see a consistent hierarchy mapping scheme (e.g., TPT), we rely on a global default given by the user (similar to [12]). If we do not see a consistent column mapping scheme, we default to the disjoint pattern. If we do not see consistent condition patterns like store constants or horizontal partitioning, we ignore any store and client conditions that are not relevant to TPH mapping.

7.2 Evolving a Mapping

Once it is known that a pattern is present in the mapping, one can then effect an incremental change to the mapping and the store based on the nature of the change. The changes in Table 2 fall into four categories, based on the nature of the change and its effect on the mapping relation.

Constructive Changes. Setting an abstract entity type to be concrete is also a change of this kind. For these changes, new rows may be added to the mapping relation, but existing rows are left alone. For example, consider the cases of adding a new derived type to a hierarchy, or adding a new property to an existing type.

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

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

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

For condition column CX (and SX), template generation follows a slightly different path. For any condition $C = v$, C `IS NULL`, or C `IS NOT NULL` that appear in *every* CX (or SX) field in $\boldsymbol{R}$ (treating a conjunction of conditions as a list that can be searched), and the value v is the same for each, add the condition to the template. If each row $r \in \boldsymbol{R}$ contains an equality condition $C = v$, but the value v is distinct for each row r, add condition $C = \otimes$ to the template. Ignore all other conditions.

Table 4 shows an example of generating a mapping template for a set of rows corresponding to a TPH relationship; the rows for the example are drawn from Table 3, with additional client and store conditions added to illustrate the effect of the algorithm acting on a single horizontal partition and a store constant. Note that the partition conditions and store conditions translate to the template; note also that the name of the store column remains consistent even though it is not named the same as the client property.

Table 4. Creating the mapping template for a type added using a TPH scheme, over a single horizontal partition where "Editor=Tom" and with a store-side constant "Source=A" — the final row shows the template filled in for a new type *Alumnus*

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

The function `NewMappingRow`(F, E) takes a template F and fills it in with details from E. Any $\otimes$ values in CE, CX, ST, and SX are filled with value E. Translating these new mapping table rows back to an EF mapping fragment is straightforward. For each horizontal partition, take all new rows collectively and run the algorithm from Section 6.3 backwards to form a single fragment.

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

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

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

Manipulative Changes. One can change individual attributes, or "facets,', of artifacts. Examples include changing the maximum length of a string property or the nullability of a property. For such changes, the mapping relation remains invariant, but is used to guide changes to the store.

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

For the `Student.Major` example, the property only maps to a single column called `TPerson.String1`. All properties that map to `TPerson.String1` are shown in Table 5. If `Student.Major` changes to length 50, and `Staff.Office` has maximum length 40, then `TPerson.String1` must change to length 50 to accommodate. However, if `TPersonString1` has a length of 100, then it is already large enough to accommodate the wider `Major` property.

Destructive Changes. Setting a concrete entity type to be abstract also qualifies in this category. For changes of this kind, rows may be removed from the mapping relation, but no rows are changed or added.

Consider as an example dropping a property from an existing type. Dropping a property follows the same algorithm as changing that property's domain from the previous section, except that the results of the query Q are used differently. If query Q returns more than one row, that means multiple properties map to the same column, and dropping one property will not require the column to be dropped. However, if r is the row corresponding to the dropped property, then we issue a statement that sets $r.SC$ to `NULL` in table $r.ST$ for all rows that satisfy $r.SX$. So, dropping `Student.Major` will execute `UPDATE TPerson SET String1 = NULL WHERE Type='Student'`. If query Q returns only the row for the dropped property, then we delete the column.[9] In both cases, the row r is removed from $\mathcal{M}$. We refer to the process of removing the row r and either setting values to `NULL` or dropping a column as `DropMappingRow`(r).

Table 5. A listing of all properties that share the same mapping as `Student.Major`

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

Refactoring Changes. Renaming constructs, moving a property, and changing an association's cardinality fit into this category. Changes of this kind may result in arbitrary mapping relation changes, but such changes are often similar to (and thus re-use logic from) changes of the other three kinds. For example, consider the case of moving a property.

[9] Whether to actually delete the data or drop the column from storage or just remove it from the storage model available to the ORM is a policy matter. One possible implementation would issue Drop Column statements.

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

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

8 Main Example, Revisited

With the machinery of channels in hand, one can now return to the example introduced in Section 2 and demonstrate how to construct a mapping that satisfies all of the requirements laid out in Section 1.1. Starting with the object-oriented application schema in Figure 1:

1. Apply an ORM CT that maps the `Person` hierarchy using the TPC mapping pattern, and the `Procedure` hierarchy using the TPH mapping pattern with the Reuse-by-Name paradigm.
2. Vertically partition the Procedure table to save off all columns with a text domain (except the few core attributes) into the table TextValues.
3. Unpivot the table TextValues.
4. Audit the table TextValues, then adorn it with a column with the current user.
5. Vertically partition the Procedure table to save off all columns with a numeric domain (except the few core attributes) into the table NumericValues.
6. Unpivot the table NumericValues.
7. Audit the table NumericValues, then adorn it with a column with the current user.

Given the steps above, it is a straightforward task to translate each step into CT's to form a channel. With a channel so defined, the application using said channel has the same business logic and data mapping as before, and the same query and data update capabilities. In addition, the application can now evolve its schema either at design-time or at run-time, and can perform arbitrary query or set-based updates, capabilities that it did not even have before without manual intervention.

Note that the solution above starts with a ORM CT, followed by a sequence of relational-only CT's. An alternative approach may instead consider CT's that that operate on and also produce object-oriented schemas; while not discussed in this paper,

one can certainly define CT's that operate over the statements in Table 2 rather than Table 1. There is no unique solution to developing a suitable mapping out of CT's, and whether one can define an optimization framework over CT's is an open and interesting question. At the very least, it is possible to define algebraic re-writing rules over some CT's as well as cost estimates over the impact of CT's on instances and queries [36].

9 Further Reading and Future Directions

This paper has centered on the concept of database virtualization, where an application schema may be treated as if it were the physical storage for that application. Virtualization isolates the application from several complicating levels of data independence, including changes in data model, significant data restructuring, and business logic. Enabling an application with database virtualization provides the application with the bidirectionality it requires to operate without risk of data loss during operation and while allowing schema evolution as the application evolves. The paper introduces the notion of a channel, a mapping framework composed of atomic transformations, each of which having provable bidirectional properties that are amenable to the requirements of the application.

Though a wealth of research has been done on schema evolution [33], very little has been done on co-evolution of mapped schemas connected by a mapping. Channels offer a such a solution.

The work on object-relational mapping has been implemented and demonstrated [38], but work is ongoing. For instance, a prominent feature of the Entity Framework (and possibly other mapping frameworks as well) is compilation of a high-level formal specification into other artifacts. Mapping compilation provides several benefits, including precise mapping semantics and a method to validate that a mapping can round-trip client states. The computational cost for compiling and validating a mapping can become large for large models, and is worst-case exponential in computational complexity [28]. An active area of research is to translate incremental changes to a model into incremental changes to the relational algebra trees of the compiled query and update views, with results that are still valid and consistent with the corresponding mapping and store changes.

Incremental or Transformation-Based Mappings. Channels are by no means the only language that has been devised to construct a mapping between two schemas from atomic components. One such framework — DB-MAIN — has already been referred to in Section 6 as a language for mitigating the effect of translating between instances of different metamodels a step at a time [18]. What follows are alternative incrementally-specified mapping languages, each introduced for a different scenario.

Both Relational Lenses [8] and PRISM [9] attempt to create an updatable schema mapping out of components that are known to be updatable. Instead of translating update statements, a lens translates database state, resolving the new state of the view instance with the old state of the logical instance. Some recent research has been performed investigating varieties of lenses that operate on descriptions of edits instead of

full states [11,21]. PRISM maps one version of an application's schema to another using discrete steps, allowing DML statements issued by version X of an application to be rewritten to operate against version Y of its database. While more complex transformations such as pivot have not been explored in either language, it may be possible to construct such operators in those tools; like channels, the key contribution of those tools is not the specific set of operators, but rather the abstractions they use and the capabilities they offer. The key difference between channels and these approaches is that neither Lenses or PRISM can propagate schema modifications or constraint definitions through a mapping.

Both-as-View (BAV) [25], describes the mapping between global and local schemas in a federated database system as a sequence of discrete transforms that add, modify, or drop tables according to transformation rules. Because relationships in these approaches are expressed using views, processing of updates is handled in a similar fashion as in the materialized view [17] and view-updatability literature [10]. The ability to update through views, materialized or otherwise, depends on the query language. Unions are considered difficult, and pivots are not considered. Schema evolution has also been considered in the context of BAV [26], though some evolutions require human involvement to propagate through a mapping.

An extract-transform-load workflow is a composition of atomic data transformations (called *activities*) that determine flow of data through a system [42]. Papastefanatos et al. addressed schema evolution in a workflow by attaching *policies* to activities. Policies semi-automatically adjust each activity's parameters based on schema evolution primitives that propagate through activities [32].

This collection of transformation-based mapping techniques covers a wide selection of model management scenarios. In addition, there is significant overlap in the expressive power of these techniques. For instance, each of the above (including channels) is capable of expressing a horizontal merge operation, even if that specific transformation has not yet been defined in the literature for each tool (e.g., horizontal merge can be defined as a relational lens, even though the literature does not explicitly do so). An interesting and open question is whether one can construct a unifying framework to compare, contrast, and taxonomize these tools. Pointfree calculus and data refinement [30] offer one possible underlying formalism for such a framework.

Monolithic Mappings. An alternative approach to mapping schemas is a declarative specification, compiled into routines that describe how to transfer data from one schema to the other. Some tools compile mappings into a one-way transformation as exemplified by data exchange tools (e.g., Clio [14]). In data exchange, data flow is uni-directional, so updatability is not generally a concern, though recent research has attempted to provide a solution for inverting mappings [3].

Schema evolution has been considered in a data exchange setting, modeled either as incremental client changes [43] or where evolution is itself represented as a mapping [47]. Both cases focus on "healing" the mapping between schemas, leaving the non-evolved schema invariant. New client constructs do not translate to new store constructs, but rather add quantifiers or Skolem functions to the mapping, which means new client constructs are not persisted. Complex restructuring operations — especially ones like pivot and unpivot that have a data-metadata transformation component — are especially

rare in data exchange (Clio is the exception [19]) because of the difficulty in expressing such transformations declaratively.

NoSQL. No contemporary discussion of application development can go without at least mentioning the wide variety of tools commonly referred to as *noSQL*. noSQL is a vague term essentially meaning a modern database management system that has in some way broken away from the assumptions of relational database systems. A noSQL system may have a query language, but it is not SQL. It may have an underlying data model, but it may not be relational, and is almost certainly not in first normal form. Such systems have become commonplace in internet applications and other applications where access to large, scalable quantities of data need to be very fast but need not have the same consistency requirements as relational systems provide.

There is no standard language or model among noSQL systems. One model that is shared among many self-identifying noSQL systems is the *key-value store*. Such a store operates much like system memory, where the key is an address and the value is the data at that address. Depending on the system, the data in the value may be highly non-normalized or atomic in nature, possibly containing references to other keys. Data in this format can be easily partitioned and accessed across a substantial number of nodes based on a partition of the key space. Recently, some effort has been made to establish links between relational and key-value stores, asserting that the two models are in fact mathematical duals of one another, and therefore not only could one query language be used to standardize access to noSQL systems, but that the same language may be targeted at relational systems as well [27].

Notable Future Directions. The mapping relation is a novel method of expressing an O-R mapping, and as such, it may have desirable properties on its own that are yet unstudied. For instance, it may be possible to express constraints on a mapping relation instance that can validate a mapping's roundtripping properties; such constraints would be useful given the high potential cost of validating an object-relational mapping.

The overall technique presented in this paper allows for client-driven evolution of application artifacts; the application schema changes, and the mapping and storage change to accommodate, if necessary. One additional dimension of changes to consider is the set of changes one can make to the mapping itself while leaving the client invariant. One possible way to handle the evolution of a channel involves translating the difference between the old channel and the new one into its own "upgrade" channel. An alternative possibility is to transform each inserted, deleted, or modified CT into DML and DDL. For instance, an inserted Pivot transformation would generate a Create Table statement (to generate the new version of the table), an insert statement (to populate the new version with the pivoted version of the old data), and a Drop Table statement (to drop the old version), each pushed through the remainder of the channel [36].

There remains a possibility that the mapping relation technique may have other applications outside of object-relational mappings. The mapping relation is a way to take a "monolithic" operation like an object-relational mapping and make it amenable to analysis for patterns, assuming that such patterns may be identified in the relationship between the source and target metamodels. An interesting and unanswered question is

whether a similar technique can be applied to a data exchange setting. One would need to define patterns over the the expressible mappings, and a mapping table representation for first-order predicate calculus, in which case similar techniques could be developed.

The set of CT's presented in this paper is not intended to be a closed set. While the requirements laid out in Section 1.1 are generally applicable to applications and their data sources, the exact set of CT's needed will likely be vastly different from one application to another. The set of CT's presented here are inspired by an examination of commercially-available software packages and have been implemented but also formally proven. Formal proofs are not likely to be acceptable or sufficient should one want to enable the individual developer to implement their own CT's and thus create an open ecosystem of CT's. An open area of research is what the implementation contract of a CT should be, and what algorithms may serve as a suitable "certification" process for a candidate CT.

References

1. Ambler, S.W., Sadalage, P.J.: Refactoring Databases. Addison-Wesley Publisher (2006)
2. Arenas, M., Barceló, P., Libkin, L., Murlak, F.: Relational and XML Data Exchange. Morgan and Claypool Publishers (2010)
3. Arenas, M., Peréz, J., Riveros, C.: The recovery of a schema mapping: bringing exchanged data back. In: PODS 2008, pp. 13–22 (2008)
4. Bancilhon, F., Spyratos, N.: Update Semantics of Relational Views. ACM Transactions on Database Systems 6(4), 557–575 (1981)
5. Banerjee, J., Kim, W., Kim, H., Korth, H.F.: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In: SIGMOD 1987, pp. 311–322 (1987)
6. Bernstein, P.A., Chiu, D.-M.W.: Using Semi-Joins to Solve Relational Queries. J. ACM 28(1), 25–40 (1981)
7. Blakeley, J.A., Muralidhar, S., Nori, A.K.: The ADO.NET Entity Framework: Making the Conceptual Level Real. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 552–565. Springer, Heidelberg (2006)
8. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: PODS 2006, pp. 338–347 (2006)
9. Curino, C., Moon, H., Zaniolo, C.: Graceful Database Schema Evolution: the PRISM Workbench. PVLDB 1(1), 761–772 (2008)
10. Dayal, U., Bernstein, P.: On the Correct Translation of Update Operations on Relational Views. ACM Transactions on Database Systems 8(3), 381–416 (1982)
11. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 61–76. Springer, Heidelberg (2010)
12. Domínguez, E., et al.: MeDEA: A database evolution architecture with traceability. Data and Knowledge Engineering 65(3), 419–441 (2008)
13. Embley, D., Thalheim, B.: Handbook of Conceptual Modeling. Springer (2011)
14. Fagin, R., et al.: Clio: Schema Mapping Creation and Data Exchange. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S., et al. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 198–236. Springer, Heidelberg (2009)
15. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems - the complete book, 2nd edn. Pearson Education (2009)

16. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Transactions on Database Systems 13(4), 486–524 (1988)
17. Gupta, A., Mumick, I.S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. IEEE Data Engineering Bulletin 18(2), 3–18 (1995)
18. Hainaut, J.-L.: The Transformational Approach to Database Engineering. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 95–143. Springer, Heidelberg (2006)
19. Hernández, M., Papotti, P., Tan, W.: Data Exchange with Data-Metadata Translations. PVLDB 1(1), 260–273 (2008)
20. Hibernate, http://www.hibernate.org/
21. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: POPL 2012 (2012)
22. Keller, A.M.: Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In: PODS, pp. 154–163 (1985)
23. Lakshmanan, L.V.S., Sadri, F., Subramanian, S.N.: On efficiently implementing SchemaSQL on a SQL database system. In: VLDB 1999, pp. 471–482 (1999)
24. Lomet, D.B., et al.: Immortal DB: transaction time support for SQL server. In: SIGMOD 2005, pp. 939–941 (2005)
25. McBrien, P., Poulovassilis, A.: Data Integration by Bi-Directional Schema Transformation Rules. In: ICDE 2003, pp. 227–238 (2003)
26. McBrien, P., Poulovassilis, A.: Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 484–499. Springer, Heidelberg (2002)
27. Meijer, E., Bierman, G.M.: A co-relational model of data for large shared data banks. Commun. ACM 54(4), 49–58 (2011)
28. Melnik, S., Adya, A., Bernstein, P.A.: Compiling Mappings to Bridge Applications and Databases. ACM TODS 33(4), 1–50 (2008)
29. Microsoft SQL Server (2005), http://www.microsoft.com/sql/default.mspx
30. Oliveira, J.N.: Transforming Data by Calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
31. Oracle TopLink, http://www.oracle.com/technology/products/ias/toplink/
32. Papastefanatos, G., et al.: What-If Analysis for Data Warehouse Evolution. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654, pp. 23–33. Springer, Heidelberg (2007)
33. Rahm, E., Bernstein, P.A.: An Online Bibliography on Schema Evolution. SIGMOD Record 35(4), 30–31 (2006)
34. Ruby on Rails, http://rubyonrails.org/
35. Snodgrass, R.T.: Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann Publishers Inc., San Francisco (2000)
36. Terwilliger, J.F.: Graphical User Interfaces as Updatable Views. PhD thesis, Portland State University (2009)
37. Terwilliger, J.F., Bernstein, P.A., Unnithan, A.: Automated Co-evolution of Conceptual Models, Physical Databases, and Mappings. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 146–159. Springer, Heidelberg (2010)
38. Terwilliger, J.F., Bernstein, P.A., Unnithan, A.: Worry-Free Database Upgrades: Automated Model-Driven Evolution of Schemas and Complex Mappings. In: SIGMOD 2010, pp. 1191–1194 (2010)
39. Terwilliger, J.F., Delcambre, L.M.L., Logan, J.: Querying Through a User Interface. Journal of Data and Knowledge Engineering 63(3), 774–794 (2007)
40. Terwilliger, J.F., Delcambre, L.M.L., Logan, J., Maier, D., Archer, D.W., Steinhauer, J., Britell, S.: Enabling revisitation of fine-grained clinical information. In: IHI 2010, pp. 420–424 (2010)

41. Terwilliger, J.F., Delcambre, L.M.L., Maier, D., Steinhauer, J., Britell, S.: Updatable and Evolvable Transforms for Virtual Databases. PVLDB 3(1), 309–319 (2010)
42. Vassiliadis, P., et al.: A generic and customizable framework for the design of ETL scenarios. Information Systems 30(7), 492–525 (2005)
43. Velegrakis, Y., Miller, R.J., Popa, L.: Preserving Mapping Consistency Under Schema Changes. VLDB Journal 13(3), 274–293 (2004)
44. Wei, H., Elmasri, R.: PMTV: A Schema Versioning Approach for Bi-Temporal Databases. In: TIME 2000, pp. 143–151 (2000)
45. Wyss, C.M., Robertson, E.L.: A Formal Characterization of PIVOT/UNPIVOT. In: CIKM 2005, pp. 602–608 (2005)
46. Wyss, C.M., Wyss, F.I.: Extending relational query optimization to dynamic schemas for information integration in multidatabases. In: SIGMOD 2007, pp. 473–484 (2007)
47. Yu, C., Popa, L.: Semantic Adaptation of Schema Mappings When Schemas Evolve. In: VLDB, pp. 1006–1017 (2005)