Updatable and Evolvable Transforms for Virtual Databases

James F. Terwilliger
Microsoft Corporation
jamest@microsoft.com

Lois M. L. Delcambre
Portland State University
lmd@cs.pdx.edu

David Maier
Portland State University
maier@cs.pdx.edu

Jeremy Steinhauer
Portland State University
jsteinha@cs.pdx.edu

Scott Britell
Portland State University
britell@cs.pdx.edu

ABSTRACT

Applications typically have some local understanding of a database schema, a *virtual database* that may differ significantly from the actual schema of the data where it is stored. Application engineers often support a virtual database using custom-built middleware because the available solutions, including updatable views, are unable to express necessary capabilities. We propose an alternative means of mapping a virtual database to a physical database that guarantees they remain synchronized under data or schema updates against the virtual schema. One constructs a mapping by composing *channel transformations* (CTs) that encapsulate atomic transformations — including complex transformations such as pivoting — with known updatability properties. Applications, query interfaces, and any other services can behave as if the virtual database is the implemented schema. We describe how CTs translate queries, DML, and DDL, and the properties that are necessary for such translation to be correct. We describe two example CTs in detail, and evaluate an implementation of channels for completeness and performance.

1. INTRODUCTION

Database virtualization mechanisms present a perspective on persistent data that is different from the actual physical structures in order to match the model an application presents to a user, mask certain data for security, simplify the structure to aid in 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; the most well-known is relational views, expressed as named relational queries.

In this work, we seek 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). Our motivation is simple: we want application developers to benefit from fully supported logical data independence, using the schema of the virtual database as if it were the schema for the real, implemented database. We take a fresh look at the problem of database virtualization and consider the design space. To avoid confusion with existing mechanisms for database virtualization, we call the stored structure the *native schema*, and refer to the schema of the virtual database as the *natural schema*.

Query-defined views, e.g., as specified in SQL, are highly expressive — especially for read-only views — and offer an elegant implementation where the resulting expression following view substitution is fully optimizable by the DBMS query evaluator. But query-defined views fall short of our requirements for several reasons. First, while the view update problem has been well studied, there is no support for expressing DDL modifications, including key and foreign key constraints, against a view. If an application's demands on its natural schema change, the developer has no recourse but to manually edit the native schema and mapping.

Second, even if DBMSs in common use supported the full view update capability described in the research literature, database applications would still require more. In our experience, we find that the relationship between an application's natural and native schemas often requires discriminated union, value transformation according to functions or lookup tables, pivoting or unpivoting, and "auditing", where data is deprecated instead of deleted. 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 Audit is not expressible in SQL without stored procedures or temporal database capabilities (e.g., [9]).

Because there are not yet tools that support true virtual databases, applications 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.

Another solution to database virtualization is model management, including object-relational mapping tools (e.g., [12]) and data exchange tools (e.g., [6]). ORM tools expose data that conforms to uniqueness and referential integrity constraints, though not all constraints expressed against the virtual schema translate into constraints on the physical database. ORM tools have limited expressive capabilities in their mappings, while data exchange tools do not typically support updates. With one notable exception [5], neither category of tool is capable of supporting evolving schemas.

In this paper, we present a new approach to database virtual-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

ization called a *channel*. One constructs a channel by composing atomic schema transformations called *channel transforms* (CTs), each of which is capable of transforming arbitrary queries, data manipulation statements, schema evolution primitives, and referential integrity constraints addressing the natural (virtual) schema into equivalent constructs against its native (physical) schema. Our approach is similar to Relational Lenses [2] 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.

Our approach presents several challenges, including closure of constraints under CTs. The image of a constraint, such as a foreign key, expressed in the natural schema cannot be represented as an ordinary foreign key expressed in the native schema, in the presence of an unpivot CT, for example. Hence we introduce here a generalized class of constraints that have appropriate closure properties relative to channels. A second challenge is that some transforms, common in query-defined views, such as join and union, do not admit the unambiguous transformation of database operations that we require. We can, however, define CTs for discriminated union and a form of join over tables with union-compatible keys.

The major contribution of this paper is the definition of a channel as a new framework for database virtualization. We define an initial set of CTs that cover a large number of database restructuring operations seen in practice, including unpivot (and pivot, its inverse). We show how CTs can be formally defined by describing how they transform the full range of query, DML, and DDL statements. Our framework includes a definition of correctness criteria for CTs that guarantees indistinguishability. All operations must support one-way invertibility where operations issued against the natural database, after being propagated to the native database, have the same effect (as observed from all operations issued against the natural database) as if the operations had been issued against a materialized instance of the natural database. We make no claim that the current set of CTs is complete; a definition of completeness may arise in our future work. Rather, we demonstrate our channel framework is extensible by showing that even application-specific transforms can be described as CTs and enjoy the same correctness proofs, closure properties, and efficient implementation. It is also future work to use a declarative language to specify channels. It is clear that there are SQL-definable views that channels cannot express, and vice-versa, but it is not clear that SQL is the best language to specify database virtualization.

We have constructed a prototype implementation of channels, which we analyze for performance using a channel that reconstructs the middleware for a publicly available application, demonstrating that the overhead induced by a channel is negligible. While we use two specific transformations — horizontal merge and pivot — as running examples through Sections 3–5, any transformation that can faithfully translate statements issued against its natural schema into native schema statements can participate in a channel.

Section 2 introduces and formalizes the concept of a CT. Sections 3, 4, and 5 demonstrate how a CT translates queries, DML, and DDL statements, respectively. Section 6 presents what it means for a CT to be correct with respect to its semantics. We provide details of a prototype implementation of channels and evaluate its performance in Section 7. Finally, Section 8 analyzes related work and Section 9 concludes the paper.

2. AN INTRODUCTION TO CHANNELS

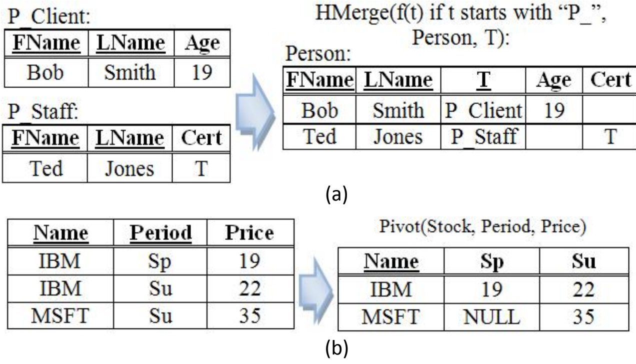
A *channel transformation* (CT) is a uni-directional mapping from the natural schema S to the native schema $\overline{S}$ that encapsulates an instance transformation. A CT represents an atomic unit of transformation that is known to be updatable. A *channel* is built by composing CTs. A channel is defined by starting with the natural schema and applying transformations one at a time until the desired native schema is achieved, which explains the naming conventions of the transformations. For instance, HMerge describes a horizontal merging of tables from the natural schema into a table in the native schema. Examples of CTs 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., $\vec{T}$) 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.
- $Apply(T, \vec{C}, \overline{\vec{C}}, f, g)$ applies an invertible function f with inverse g to each row in the table T. The function input is taken from columns $\vec{C}$, and output is placed in columns $\overline{\vec{C}}$.
- $Unpivot(T, A, V, \overline{T})$ 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 $\overline{T}$.
- $Adorn(T, e, \overline{\vec{A}}, \vec{C})$ adds columns $\overline{\vec{A}}$ to table T. The columns hold the output of function e, which returns the state of environment variables. Values in $\overline{\vec{A}}$ are initialized with the current value of e on insert, and refreshed on update whenever any values in columns $\vec{C}$ change.
- $Audit(T, \overline{B}, \overline{E})$ 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 natural schema instance corresponds to all rows from the native database where $\overline{E} \neq null$.

Note that in this list of example CTs, Apply implements data transformation, Adorn and Audit encapsulate functions that are commonly supported in application middleware, and the others support classical logical schema restructuring operations.

These informal definitions of CTs describe what each "does" to a fully materialized instance of a natural schema, but a natural schema is virtual and thus stateless. Thus, a CT maintains the operational relationship between natural and native schemas by translating all operations expressed against the natural schema into equivalent operations against the native schema.

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 CTs input (natural) schema into statements against its output (native) schema. Let



(a)

P_Client:

FName	LName	Age
Bob	Smith	19

P_Staff:

FName	LName	Cert
Ted	Jones	T

HMerge(f(t) if t starts with "P_", Person, T):

Person:

FName	LName	T	Age	Cert
Bob	Smith	P_Client	19	
Ted	Jones	P_Staff		T

(b)

Name	Period	Price
IBM	Sp	19
IBM	Su	22
MSFT	Su	35

Pivot(Stock, Period, Price)

Name	Sp	Su
IBM	19	22
MSFT	NULL	35

Figure 1: Instances transformed by an HMerge CT (a) and a Pivot CT (b)

$\mathcal{S}$ be the set of possible relational schemas and $\mathcal{D}$ be the set of possible database instances. Let 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* $\mathsf{S} : \mathcal{S} \to \mathcal{S} \cup \{\epsilon\}$. A channel transformation may have prerequisites on the input schema s, where $\mathsf{S}(s) = \epsilon$ if those prerequisites are not met. Function S must be injective (1-to-1) whenever $\mathsf{S}(s) \neq \epsilon$.
- Function I is an *instance transformation* $\mathsf{I} : \mathcal{S} \times \mathcal{D} \to \mathcal{D}$, 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* $\mathsf{Q} : \mathcal{S} \times Q \to Q$, defined 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* $\mathsf{U} : \mathcal{S} \times [\mathcal{U}] \to [\mathcal{U}] \cup \epsilon$, defined on pairs of input $(s, \vec{u})$ where $\mathsf{S}(s) \neq \epsilon$ and update transaction $\vec{u}$ 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 $\mathsf{U}(s, \vec{u}) \neq \epsilon$, and output a valid update transaction over $\mathsf{S}(s)$. Expression $\mathsf{U}(s, \vec{u})$ evaluates to the error state ϵ if $\vec{u}$ 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 natural schema (and an instance of it) into a native schema (and an instance of it). These functions are not used in our implementation, but allow us to reason about the correctness of functions Q and U as shown in Section 6. Neither query nor update functions require a database instance as input; a CT directly translates the statements themselves. We present two example CTs: Horizontal Merge and Pivot.[1] In this section, we define the action of the schema and instance functions.

[1]Additional examples are provided in Appendix D.

2.1 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 input schema that does not satisfy predicate f is left unaltered[2]. Figure 1(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 $\vec{T}^f$ be the set of all tables in input schema s that satisfy predicate f and $\mathbf{Cols}(t)$ be the set of columns for table t. We define $\mathsf{S}_{\mathbf{HM}}$ on schema s as follows: replace tables $\vec{T}^f$ with table $\overline{T}$ with columns $(\bigcup_{t \in \vec{T}^f} \mathbf{Cols}(t)) \cup \{\overline{C}\}$, the union of all columns from the source tables eliminating duplicates, plus the provenance column, whose domain is the names of the tables in $\vec{T}^f$. The key of $\overline{T}$ is the common key from tables $\vec{T}^f$ plus the column $\overline{C}$. $\mathsf{S}_{\mathbf{HM}}(s) = \epsilon$ if the keys are not union-compatible and identically named.

We define $\mathsf{I}_{\mathbf{HM}}$ on schema s and instance d by replacing the instances of $\vec{T}^f$ in d with $\uplus_{t \in \vec{T}^f}(t \times \{(name(t))\})$, where $\uplus$ 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.

2.2 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 1(b).

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

Let $\mathsf{I}_{\mathbf{PV}}$ be defined on schema s and instance d by replacing the instance of T in d with $⍁_{\mathbf{Dom}(A);A;V} T$, where $⍁$ 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, $\mathbf{PV}$ 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.

3. TRANSLATING QUERIES

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

- Left outer join (⟕) and left antisemijoin (▷)
- Pivot ($⍁_{\vec{C};A;V}$): For a set of values $\vec{C}$ on which to pivot, pivot column A, and pivot-value column V (translating a relation

[2]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 1(a).

Table 1: The DML and DDL statements that the channel transformations support.

Statement	Formalism	Explanation of Variables
Insert	$\mathbf{I}(T, \vec{C}, Q)$	Insert rows into table T into columns $\vec{C}$, using the values of $\vec{C}$ from the rows in Q. The value of Q may be a table constant or a query result.
Update	$\mathbf{U}(T, \vec{F}, \vec{C}, Q)$	Update rows in table T that satisfy all equality conditions $\vec{F}$ specified on key columns. Non-key columns $\vec{C}$ 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	$\mathbf{D}(T, \vec{F})$	Delete rows from table T that satisfy all equality conditions $\vec{F}$ specified on key columns. Not all key columns need to have a condition.
Add Table	$\mathbf{AT}(T, \vec{C}, \vec{D}, \vec{K})$	Add new table T, whose columns $\vec{C}$ have domains $\vec{D}$, with key columns $\vec{K} \subseteq \vec{C}$.
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.
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	$\mathbf{FK}(\vec{F} \vert T.\vec{X} \rightarrow \vec{G} \vert T'.\vec{Y})$	Add foreign key constraint from columns $T.\vec{X}$ to columns $T'.\vec{Y}$, so that for each tuple $t \in T$, if t satisfies conditions $\vec{F}$ and $t[\vec{X}] \neq null$, there must be tuple $t' \in T'$ such that t' satisfies conditions $\vec{G}$ and $t[\vec{X}] = t'[\vec{Y}]$.
Drop Foreign Key	$\mathbf{DFK}(\vec{F} \vert T.\vec{X} \rightarrow \vec{G} \vert T'.\vec{Y})$	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	$\mathbf{Loop}(t, Q, \vec{S})$	For each tuple t returned by query Q, execute transaction $\vec{S}$.
Error	$\mathbf{Error}(Q)$	Execute query Q, and raise an error if any rows are returned.

from key-attribute-value triples into a normalized, column-per-attribute form)

- Unpivot ($\boxed{\swarrow}_{\vec{C};A;V}$), the inverse operation to pivot
- Function application ($\alpha_{\vec{I},\vec{O},f}$): Apply function f on input columns $\vec{I}$ and place the result in output columns $\vec{O}$

The pivot query operator is defined as:

$$\boxed{\nearrow}_{\vec{C};A;V} Q \equiv (\pi_{columns(Q)-\{A,V\}} Q) ⟕ (\rho_{V \rightarrow C1} \pi_{columns(Q)-\{A\}} \sigma_{A=C1} Q)$$

$$⟕ \ldots ⟕ (\rho_{V \rightarrow Cn} \pi_{columns(Q)-\{A\}} \sigma_{A=Cn} Q) \text{ for } C1, \ldots, Cn = \vec{C}$$

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 [13]). Contrast this property with the Pivot CT, where the set of output columns is dynamic; we explore the relationship between the pivot query operator and the Pivot CT later in this section.

Figure 2 shows an example instance transformed by a pivot operator $\boxed{\nearrow}_{\vec{C};A;V}$, 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 $\vec{C}$, instance $\rho_{V \rightarrow C} \pi_{columns(Q)-\{A\}} \sigma_{A=C} Q$ 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.

We introduce a pivot operator into 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. We leave some details of the pivot query operator 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 the Pivot operator [19]).

The unpivot query operator is defined as follows:

$$\boxed{\swarrow}_{\vec{C};A;V} Q \equiv \bigcup_{C \in \vec{C}} (\rho_{C \rightarrow V} \pi_{columns(Q)-(\vec{C}-\{C\})} \sigma_{C<>null}(Q) \times \rho_{1 \rightarrow A}(name(C)))$$

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 Q_{PV}, the query translation function for Pivot, which translates all references to table T into $\boxed{\swarrow}_{\mathbf{Dom}(A);A;V} \overline{T}$. 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 $\boxed{\swarrow}$ 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 natural schema, and thus columns are added or deleted from the native schema. (Pivot will process the DDL statements for adding or dropping domain

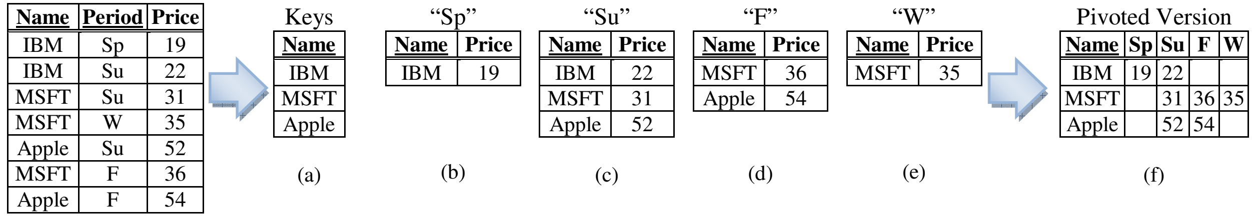


Figure 2: An example of an instance transformed by the pivot query operator $\boxslash_{\{Sp,Su,F,W\};Period;Price}$, first broken down into intermediate relations that correspond to the set of non-pivoted columns (a) and the subsets of rows corresponding to each named value in the pivot column "Period" (b–e). The pivoted instances are then outer joined with the first instance (Keys) to produce the pivot table (f).

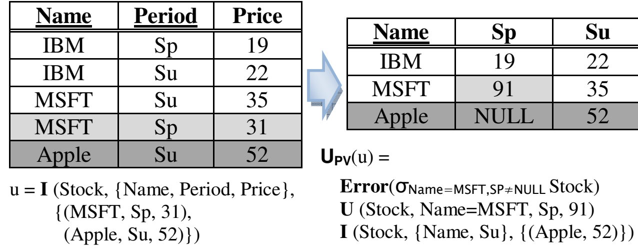


Figure 3: An example of inserts translated by a Pivot CT

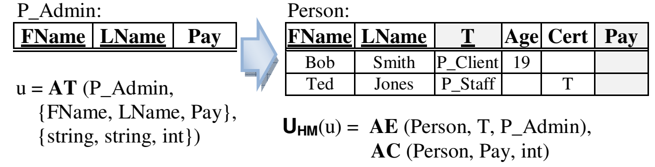


Figure 4: An Add Table statement translated by HMerge

elements — see Section 5 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.

4. 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 $\mathbf{Loop}(t, Q, \vec{S})$, 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 $\vec{S}$ execute. Statements in $\vec{S}$ 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 all of the key values for rows that match the statement's conditions, then issue an update or delete for each of the qualifying rows. 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 U includes computation of $\mathsf{U}(\mathbf{I}(T, \vec{C}, Q))$, $\mathsf{U}(\mathbf{D}(T, \vec{F}))$, 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.

As an example, consider $\mathsf{U}_{\mathsf{PV}}(\mathbf{I}(T, \vec{C}, Q))$, pushing an insert statement through a Pivot. Each tuple $(\vec{K}, A, V)$ inserted into the input 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. U_{PV} thus transforms the insert statement into an update statement that updates column A for the row with key $\vec{K}$ to be value V, if the row exists. In Figure 3, the inserted row with `Name = ''MSFT''` corresponds to a key value already found in the output schema; that insert row statement therefore translates to an update in the output 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 input 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 input schema of the CT $Pivot(T, A, V, \overline{T})$. We define the action of the CTs update function U_{PV} on an insert DML statement $\mathbf{I}(T, \vec{C}, Q)$ as follows:

$\mathsf{U}_{\mathsf{PV}}(s, \mathbf{I}(T, \vec{C}, Q)) =$

$\mathbf{Error}((\pi_{\mathbf{Keys}(T)} Q) \cap \pi_{\mathbf{Keys}(\overline{T})} \boxslash_{\mathbf{Dom}(A);A;V}(\pi_{\mathbf{Cols}(\overline{T})}(Q \bowtie \overline{T}))),$

(check that inserted rows do not collide with existing data)

$\forall_{a \in \mathbf{Dom}(A)} \mathbf{Loop}(t, \sigma_{A=a} Q \bowtie (\pi_{\mathbf{Keys}(\overline{T})} \overline{T}),$

$\mathbf{U}(\overline{T}, \forall_{c \in \mathbf{Keys}(\overline{T})} c = t[c], \{a\}, \pi_V t)),$

(update each row whose key is already present)

$\mathbf{I}(\overline{T}, \mathbf{Cols}(\overline{T}), \boxslash_{\mathbf{Dom}(A);A;V}(Q \overline{\ltimes} (\pi_{\mathbf{Keys}(\overline{T})} \overline{T})))$

(inserts for non-existent rows)

5. 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 a non-key column C is dropped, then any row that had a C value of E will have that value set to null. However, if 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.

5.1 Referential Integrity

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

To translate **FK** (and **DFK**) statements, we 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 $FK(true|T.\vec{X} \rightarrow true|U.\vec{Y})$ against a natural schema, its image in the native schema may involve arbitrarily complex queries.[3]

5.2 HMerge Translation of Add Table

Let $\mathsf{U_{HM}}$ be the update function for $HMerge(f, \overline{T}, \overline{C})$, and let s be its input schema. We define the action of $\mathsf{U_{HM}}$ on an Add Table statement for a table t that satisfies f as follows:

$\mathsf{U_{HM}}(s, \mathbf{AT}(t, \vec{C}, \vec{D}, \vec{K})) =$

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

$\mathbf{AT}(\overline{T}, \vec{C} \cup \{\overline{C}\}, \vec{D} \cup \{\{name(t)\}\}, \vec{K} \cup \{\overline{C}\})$

If the merged table already exists in the output schema, the function adds a new domain element to the provenance column to point to rows coming from the new table. Then, the function $\mathsf{U_{HM}}$ 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 4, assuming tables "P_Client" and "P_Staff" already exist in the input schema.

5.3 HMerge Translation of Add Column

Let $\mathsf{U_{HM}}$ be the update function for $HMerge(f, \overline{T}, \overline{C})$, and let s be its input schema. We define the action of $\mathsf{U_{HM}}$ 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)

6. CORRECTNESS

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

- For an input 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 input schema (Figure 5(a)).
- For an input schema s and a valid update transaction $\vec{u}$ against s, let $\overline{s} = \mathsf{S}(s)$ (the translated schema) and $\overline{u} = \mathsf{U}(s, \vec{u})$ (the translated update). Then, $\overline{u}(\overline{s}) = \mathsf{S}(\vec{u}(s))$. Running a translated update against a translated schema is equivalent to running the update first, then translating the result (Figure 5(b)).
- For an input schema s and a valid update transaction $\vec{u}$ 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 $\overline{u} = \mathsf{U}(s, \vec{u})$ (the translated update). Finally, let $\overline{q^t_u} = \mathsf{Q}(\vec{u}(s), q^t)$, the result of translating query q^t on schema s after it was updated by $\vec{u}$. Then, $\overline{q^t_u}(\overline{u}(\overline{s})) \equiv q^t(\vec{u}(s))$. 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)).

[3]For details on tiered FK statements, see Appendix C.

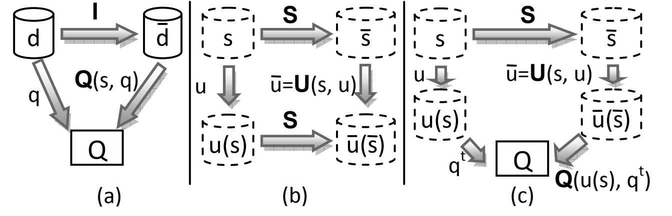


Figure 5: Three commutativity diagrams that must be satisfied for CTs that have defined instance-at-a-time semantics

We abuse the notation slightly in the last commutativity property 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 $u = \mathbf{I}(t, \vec{C}, Q)$, then $q^t(u(s)) \equiv t \cup Q$ (all of the rows that were in t plus the new rows Q added)
- If $u = \mathbf{D}(t, \vec{F})$, then $q^t(u(s)) \equiv \sigma_{\neg\vec{F}}t$ (all of the rows that were in t that do not satisfy conditions $\vec{F}$)
- If $u = \mathbf{AC}(t, C, D)$, then $q^t(u(s)) \equiv t \times \rho_{1 \rightarrow C}\{null\}$ (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:

- $\mathsf{U}(s, \vec{u}) = \epsilon \iff \mathsf{S}(\vec{u}(s)) = \epsilon$. Function U returns an error if and only if applying the update transaction to the input schema results in the schema no longer meeting the transformation's schema preconditions.
- If $\mathsf{U}(s, \vec{u}) \neq \epsilon$ and d is an arbitrary instance of schema s, $\mathsf{U}(s, \vec{u})(\mathsf{I}(s, d)) = \epsilon \iff \vec{u}(d) = \epsilon$. 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 native instance if and only if a violation would occur if a materialized instance of the natural schema were updated. Note that such a violation occurs when the transaction is *executed* rather than when it is *translated*.

7. EVALUATION

We constructed a prototype implementation of channels and channel transformations, including all of the transformations listed in Section 2. Our implementation uses a *provider model*, a common method of representing statements in an internal, DBMS-independent format in the channel before invoking a provider that translates the statements into implementation-specific SQL.

We implemented a representative sample of a publicly-available clinical application.[4] Tables not covered by our sample all map to the native schema using the same sequences of CTs as tables in our sample. Our sample represented over 40 of the hundreds of tables in the application's natural schema. We are evaluating the channel and not the application's specific native schema, so we do not consider alternative native schemas. The capabilities required by the application of its natural schema are: single-row inserts, key-based row update and deletion, and queries for rows based on key values or equality or range conditions. A channel, both formally and in our implementation, provides those capabilities.

The constructed channel comprised 38 CTs: 7 Unpivots, 7 VPartitions, 22 VMerges, and 2 HMerges. This channel was able to fully represent the mapping between the natural and native schemas, with one exception. The real application merges together tables with non-union-compatible keys that are only loosely semantically related. For completeness, we defined said transformation formally as a CT similar to HMerge, but did not implement it for the test.

We measured how much time was spent in the channel compared to the amount of time required to perform the native database operations. The tests that we ran are representative of the typical workload of a business application in general, and our target application in particular. We ran tests corresponding to create, retrieve, update, and delete operations, one test that represents typical user queries for data of a particular range, and two schema evolution tests:

1. Insert a new colonoscopy procedure record
2. Update data entries for a procedure with a specified ID
3. Delete a colonoscopy with a specified ID
4. Retrieve a particular procedure from the database based on its key value — in other words, a query of the form, "SELECT * FROM Colonoscopy WHERE proc_id = X"
5. Retrieve all colonoscopies performed in a specific date range, returning ˜5% of rows
6. Add a new column to a the Colonoscopy table
7. Drop a column from the Colonoscopy table

We measured time spent translating the operation within the channel, time spent in the provider generating DBMS-specific SQL, and time spent within the database executing the statement. Experiments were run on a Windows 7 64-bit PC with a 3 GHz, dual-core processor and 4 GB of RAM, using a commercially available DBMS. The database instance was 70 MB in size covering 25,000 artificially-generated clinical procedures, representative of a small database. Table 2 shows the results of our test runs in terms of milliseconds and percentage of total execution time (from when a statement is issued against the natural schema until processing completes). We manually checked channel output to verify no unnecessary work was being done to inflate database execution time, e.g., self-joins when none were needed or pivots on data that would eventually be projected away. We also verified that the generated statements were equivalent to the ones used by the application.

We ran statements in groups of 100, and ran each group 100 times. The standard deviation in recorded times was not statistically significant. Ignoring the row marked with an asterisk for a moment, database execution time is substantially larger than translation and provider time combined. For DML and DDL statements, translation and provider time comprise less than 4% of total time.

[4]See Appendix B for a description of the application.

Table 2: Performance of statements against a natural schema, averaged over 100 executions of 100 runs each of the test case.

	Time in milliseconds			Percent of total time		
Scenario	**Chan.**	**Prov.**	**DB**	**Chan.**	**Prov.**	**DB**
Insert	44.97	66.00	8933.23	0.50%	0.74%	98.8%
Update	41.96	47.67	2295.21	1.77%	2.00%	96.2%
Delete	5.10	1.16	1023.05	0.57%	0.13%	99.3%
Key Query	2.18	549.15	3000.39	0.06%	15.5%	84.5%
Key Query*	2760.02	548.27	3013.18	43.7%	8.67%	47.6%
Range Query	12.85	552.14	2.4×10^5	˜0%	0.23%	99.7%
Add Column	3.85	0.34	186.92	2.30%	0.18%	97.5%
Drop Column	4.13	0.01	354.61	1.80%	˜0%	98.2%

Recall from Section 3 that a CT translates a query by searching for table references in a query and replacing them in place. Our first (naïve) implementation of a channel had each CT traverse each query tree looking for table references, even when none were found. Thus query translation time increased linearly with the number of CTs in the channel. The row in Table 2 marked with an asterisk ("Key Query *") shows the performance result of running key-based queries through this naïve implementation.

Our solution to this problem was to pre-process a channel. For each table *t* in the natural schema, we translate the query "SELECT * FROM t", and save the result. The query tree is then traversed once only looking for table references and replacing with the saved results. Note that the provider and database time for key-based queries is virtually the same for the optimized and non-optimized versions, since the generated queries are the same, but that the time required by channel translation drops from 43.7% to only 0.06%. The provider has reasonable but sometimes significant execution time for queries due to the sheer size of the SQL output text generated from a large number of columns. Though not the main focus of our work, we are investigating ways to optimize the provider.

Because a channel is stateless, the translation time and provider time of a channel are not dependent upon the size of the database. To verify this hypothesis, we ran all tests on a database with 250,000 generated procedures (700 MB). In all cases, the channel overhead remained the same as in Table 2 while database execution time went up corresponding to the larger size. A channel can scale to databases of larger size without incurring additional overhead.

8. RELATED WORK

There is a wealth of research available on schema evolution [15], but little research has been done on propagating schema evolution through a mapping. MeDEA allows a developer to manually write policies that describe what physical actions to take on a database per change per mapping [5].

Both Relational Lenses [2] and PRISM [3] attempt to create an updatable schema mapping out of components that are known to be updatabale. 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. 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.

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 [6]). 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 [1]. Schema evolution has been considered in a data exchange setting [20]; the focus in such research is on repairing a mapping while leaving the source data instance unaltered rather than evolving it, thus new target columns or tables can be added without existential quantifiers. Pivot and unpivot are especially rare in data exchange (Clio is the exception [8]) because of the difficulty in expressing such transformations declaratively.

An extract-transform-load workflow is a composition of atomic data transformations (called *activities*) that determine flow of data through a system [18]. 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 [14].

Both-as-View (BAV) [10], 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 [7] and view-updatability literature [4]. 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 [11], though some evolutions require human involvement to propagate through a mapping.

9. CONCLUSION AND FUTURE WORK

We have introduced the channel as an abstraction that, like a traditional view, provides logical data independence by allowing applications access to a schema that is isolated from the native or implemented schema. A channel further permits schema modifications and integrity constraint declarations against a natural schema. Thus applications that employ a channel may treat a natural schema as indistinguishable from a native schema, so developers may express schema evolution against the natural schema knowing that all updates to data and schema will propagate to the database through the same mechanism. Performance results indicate that channel overhead is dominated by database execution time.

We have developed a technique for handling incremental changes to the channel itself that 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 in a channel 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 [16].

To fully support indistinguishability, a channel must encapsulate all of the data and query transformations that occur between an application's natural schema and its native schema. Such transformations may include business logic that is typically found in data access layers or stored procedures. The Adorn and Audit transformations introduced in Section 2 are examples of such transformations. While the other CTs in Section 2 are defined on materialized instances, a business logic CT may be non-deterministic, e.g., Adorn, whose output depends on the state of environment variables. Such a CT has semantics defined in terms of its effects on queries and updates rather than on instances, but can still be validated against all of the correctness properties except Figure 5(a).

10. REFERENCES

[1] M. Arenas, J. Peréz, and C. Riveros. The recovery of a schema mapping: bringing exchanged data back. *PODS 2008*, 13–22.

[2] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. *PODS 2006*, 338–347.

[3] C. Curino, H. Moon, and C. Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. *VLDB 2008*, 761–772.

[4] U. Dayal and P. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Transactions on Database Systems*, September 1982, 8(3):381–416.

[5] E. Domíngueza et al. MeDEA: A database evolution architecture with traceability. *Data and Knowledge Engineering*, 65(3) (2008).

[6] R. Fagin et al. Clio: Schema Mapping Creation and Data Exchange. *Conceptual Modeling: Foundations and Applications*, 2009, 198–236.

[7] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 1995, 18(2):3-18.

[8] M. Hernández, P. Papotti, and W. Tan. Data Exchange with Data-Metadata Translations. *VLDB 2008*, 260–273.

[9] D. B. Lomet et al. Immortal DB: transaction time support for SQL server. *SIGMOD 2005*, 939–941.

[10] P. McBrien and A. Poulovassilis. Data Integration by Bi-Directional Schema Transformation Rules. *ICDE 2003*, 227–238.

[11] P. McBrien and A. Poulovassilis. Schema Evolution in Heterogeneous Database Architectures, a Schema Transformation Approach. *CAiSE '02*, 484–499.

[12] S. Melnik, A. Adya, and P. A. Bernstein. Compiling Mappings to Bridge Applications and Databases. *SIGMOD 2007*, 461–472.

[13] Microsoft SQL Server 2005. http://www.microsoft.com/sql/default.mspx.

[14] G. Papastefanatos et al. What-if analysis for data warehouse evolution. *DaWaK 2007*, 23–33.

[15] E. Rahm and P. A. Bernstein. An Online Bibliography on Schema Evolution. *SIGMOD Record*, 35(4):30–31.

[16] J. F. Terwilliger. *Graphical User Interfaces as Updatable Views*. PhD thesis, Portland State University, 2009.

[17] D. Tsichritzis and A. C. Klug. ANSI/X3/SPARC DBMS Framework. Report of the study group on data base management systems, AFIPS Press, Arlington, Va., 1977.

[18] P. Vassiliadis et al. A generic and customizable framework for the design of ETL scenarios. *Information Systems*, 30(7):492–525.

[19] C. M. Wyss and E. L. Robertson. A Formal Characterization of PIVOT/UNPIVOT. *CIKM 2005*, 602–608.

[20] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings When Schemas Evolve. *VLDB 2005*, 1006–1017.

Patient(P-Id, Gender, Race, …)
Staff(Id, LastName, FirstName, …)
Medications(M-Id, MedName, …)
Colonoscopy(Proc-id, P-Id, Date, Pulse, Weight, Aspirin, Smoking, …) With many additional tables – in 1-to-1 relationship to Colonoscopy: **SurgicalHistory**(Proc-Id, KidneyTransplant, gastrectomy, …) **IndicationsConstipation**(Proc-Id, Frequency, …) **IndicationsFamHistory**(Proc-Id, PolypChild, PolypSibling, …) **…**
Each Colonoscopy may have zero or more findings: **Finding**(F-Id, Proc-Id, x, y, kind, …)
One kind of finding: **Polyp**(F-Id, Proc-Id, ImageTaken, SnareWCautery, …)
Polyps can have zero or more therapies: **TherapyHemostatic**(F-Id, Proc-Id, Laser, Injection, …) With additional tables in 1-to-1 relationship to TherapyHemostatic: **BandingHem**(F-Id, Proc-Id, Device, Placed, …) **HeatProbe**(F-Id, Proc-Id, Instrument, Watts, …) **…**

Figure 6: The view schema for a real-world application capturing data from clinical endoscopies

Patient(P-Id, Gender, Race, …)
Staff(Id, LastName, FirstName, …)
Medications(M-Id, MedName, …)
ProcedureMain(Proc-Id, P-Id, Proc-Type, Date, ..)
Procedure-Text(Proc-Id, Proc-Att, Finding-Att, Textvalue)
Procedure-String(Proc-Id, Proc-Att, Finding-Att, Stringvalue)
Procedure-Real(Proc-Id, Proc-Att, Finding-Att, Realvalue)
Procedure-Bool(Proc-Id, Proc-Att, Finding-Att, Booleanvalue)

Figure 7: The logical schema for the application from Figure 6

APPENDIX

A. ACKNOWLEDGEMENTS

This work was supported in part by NSF grant #0534762, grant #1R21LM009550 from the National Library of Medicine (NLM), and a grant from the Collins Medical Trust. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the National Library of Medicine, National Institutes of Health or the National Science Foundation.

B. SAMPLE APPLICATION

To motivate this work, consider a real-world application[5] that supports data entry and retrieval for medical reports, e.g., colonoscopies. The application has a complex user interface, operating on a natural schema with tables for patients, staff, and central tables for the main reports with 178 descriptive attributes, along with dozens of weak entities corresponding to the various findings, therapies, indications, and other details of the report. Figure 6 shows a fragment of the application's natural schema (which is far too large to fit in a single figure).

The native schema uses a generic structure for all procedure-related data where each non-null attribute value from the natural schema is represented in a separate row with the procedure ID, the attribute, and the value, broken into four tables based on the value's data type. Figure 7 shows a fragment of the native schema. Anecdotally, such a generic schema frequently occurs in business applications, especially medical records software, because the columns in the natural schema may be substantial in number and frequently changing, the data values are often sparsely populated, and schema updates would otherwise break the interface between the many software artifacts and the database.

[5]www.cori.org

```
CREATE VIEW Colonoscopy AS
SELECT Proc-id, P-Id, Date, Pulse, Weight, Aspirin, Smoking, …
FROM ProcedureMain LEFT JOIN (Proc-Text PIVOT (MAX(textvalue) ON VALUES …) LEFT JOIN
     (Proc-String PIVOT (MAX(stringvalue) ON VALUES …) LEFT JOIN …
```
(a)

(b)	(c)
INSERT INTO Colonoscopy (Proc-id, P-Id, Weight, Aspirin) VALUES (101, 1, 180, true)	INSERT INTO Proc-Main (Proc-id, P-Id) VALUES (101, 1) INSERT INTO Proc-Real (Proc-id, Proc-attr, Find-id, realvalue) VALUES (101, Weight, 0, 180) INSERT INTO Proc-Bool (Proc-id, Proc-attr, Find-id, boolvalue) VALUES (101, Aspirin, 0, true)

(d)	(e)
ALTER TABLE Colonoscopy ADD COLUMN Coumadin Bool	No schema changes necessary, but note that the new value "Coumadin" is a possible value for Proc-Bool.Proc-attr

Figure 8: For the application from Figure 6, the query that defines the Colonoscopy table in the natural schema, along with how to translate an insert and a column addition specified against the natural schema

Figure 8(a) shows the skeleton of the query that would serve as the SQL view definition for natural schema table `Colonoscopy`. The query is not considered updatable by any current DBMS or research effort, yet the insert shown in Figure 8(b) against the natural schema is unambiguously equivalent to the sequence of inserts in Figure 8(c) against the logical schema. To query and manipulate data through the natural schema, the application employs a custom data transformation layer in the form of stored procedures and program code, which is a common solution. The result is, effectively, to make the view updatable through manual means.

When the time comes to modify the natural schema of the application — say, if new columns or tables are required — the developer must update the database to accommodate the new changes, as well as change the data transformation code, a manual and error-prone process. For this application, the Add Column statement specified in Figure 8(d) against the natural schema is unambiguously equivalent to allowing a new possible value that can appear in a column in the logical schema, shown in Figure 8(e).

C. TRANSLATING FOREIGN KEYS

A foreign key constraint in the standard relational model is a containment relationship between two queries, $\pi_{\vec{C}} T \subseteq \pi_{\vec{K}} T'$, where $\vec{C}$ is the set of columns in T comprising the foreign key and $\vec{K}$ is the key for T'. Figure 9(a) shows a traditional foreign key between two tables. Figure 9(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 9(c) shows the same tables as Figure 9(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, we observe that in many cases, the translated query is in the form $\pi_{\vec{C}'} \sigma_{\vec{F}'} T'$ or even $\pi_{\vec{C}'} T'$, 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 per-

formance.

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 CTs 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, so we do not discuss them further here.

C.1 Tiered Foreign Keys

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

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

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

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

where $\vec{Y}$ is a (not necessarily proper) subset of the primary key columns of table T', and $\vec{F}$ and $\vec{G}$ are sets of conditions on key columns (for their respective relations) with AND semantics. Figure 9(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 $\textbf{FK}(true|T.\vec{X} \rightarrow true|T'.\vec{Y})$ is precisely a Tier 1 foreign key when $\vec{Y}$ is the primary key for T'. We represent Tier 1 FKs using Tier 2 FK syntax $\textbf{FK}(\vec{F}|T.\vec{X} \rightarrow \vec{G}|T'.\vec{Y})$ because it simplifies the description of a CT, and because it is trivial to check at runtime whether $\vec{F}$ and $\vec{G}$ are empty and $\vec{Y}$ 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 arbitrarily complex relational algebra over a single schema, expressed as $\textbf{Check}(Q \subseteq Q')$.

The example in Figure 9(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.

C.2 Tier 2 FK as a Trigger

A Tier 2 foreign key $\textbf{FK}(\vec{F}|T.\vec{X} \rightarrow \vec{G}|T'.\vec{Y})$ 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
```

Sales:

ID	Buyer	Item	Vendor
1	101	Soup	A
2	224	Bread	B

Food:

Item	Vendor	Stock
Bob	Smith	19
Bob	Jones	44
Sue	Jones	95

FK(true | Sales(Item, Vendor) → true | Food(Item, Vendor))

(a)

Sales:

ID	Buyer	Item	Vendor
1	101	Soup	A
2	224	Bread	B

AllItems:

Item	Vendor	Type	Stock
Soup	A	Food	19
Soup	B	Food	44
Bread	B	Food	95
Yarn	B	Textile	34

FK(true | Sales(Item, Vendor) → Type=Food | AllItems(Item, Vendor))

Note: cannot insert row (3, 645, Yarn, B) into Sales, since qualifying row in AllItems does not meet condition Type=Food specified in FK.

(b)

Sales:

ID	Buyer	Item	Vendor
1	101	Soup	A
2	224	Bread	B

Food:

Item	A	B
Soup	19	44
Bread	NULL	95

Check ($\pi_{Item,Vendor}$ Sales $\subseteq \pi_{Item,Vendor} \swarrow\!\!\square_{\{A,B\},Vendor,Item}$ Food)

(c)

Figure 9: Examples of Tier 1 (a), Tier 2 (b), and Tier 3 (c) foreign keys

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 $T.\vec{X}$ and $T'.\vec{Y}$, and the triggers may be able to operate using index-only scans.

C.3 HMerge Translation of Tier 2 FK

Let $\mathsf{U_{HM}}$ be the update function for the CT $HMerge(f, \overline{T}, \overline{C})$, and let s be its input schema. We define the action of $\mathsf{U_{HM}}$ on a Tier 1 or 2 foreign key as follows:

$\mathsf{U_{HM}}(s, \textbf{FK}(\vec{F}|T.\vec{X} \rightarrow \vec{G}|T'.\vec{Y})) =$

If $T \models f$ and $T' \not\models f$, then

$\textbf{FK}(\vec{F} \wedge (\overline{C} = T)|\overline{T}.\vec{X} \rightarrow \vec{G}|T'.\vec{Y})$

Else, if $T \not\models f$ and $T' \models f$, then

$\textbf{FK}(\vec{F}|T.\vec{X} \rightarrow \vec{G} \wedge (\overline{C} = T')|\overline{T}.\vec{Y})$

Else, if $T \models f$ and $T' \models f$, then

$\textbf{FK}(\vec{F} \wedge (\overline{C} = T)|\overline{T}.\vec{X} \rightarrow \vec{G} \wedge (\overline{C} = T')|\overline{T}.\vec{Y})$

Else, $\textbf{FK}(\vec{F}|\overline{T}.\vec{X} \rightarrow \vec{G}|T'.\vec{Y})$

This result follows from query translation — one can translate the fragment into its Tier 3 equivalent, translate the two constituent queries through $\mathsf{Q_{HM}}$, 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.

C.4 Pivot Translation of Tier 2 FK

Let $\mathsf{U_{PV}}$ be the update function for the CT $Pivot(T_p, A, V, \overline{T_p})$, and let s be its input schema. The action of $\mathsf{U_{PV}}$ has several cases based on the tables, columns, and conditions in a Tier 1 or 2 foreign key definition; for brevity, we describe two of the interesting cases:

Case 1: $T = T_p$, $T' \neq T_p$, and $A \in \vec{X}$. One of the source columns is pivoted — this is the case demonstrated in Figure 9).

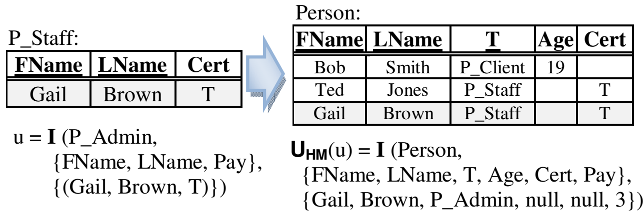


Figure 10: An example of an insert statement translated by an HMerge CT

$\mathsf{U_{PV}}(s, \mathbf{FK}(\vec{F}|T.\vec{X} \to \vec{G}|T'.\vec{Y})) =$

$\quad\quad \mathbf{Check}(\pi_{\vec{X}}\sigma_{\vec{F}}\not\boxtimes_{\mathbf{Cols}(T_p)-\mathbf{Keys}(T_p);A;V}\overline{T_p}, \pi_{\vec{Y}}\sigma_{\vec{G}}T')$

Figure 9(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: $T = T_p$, $T' \neq T_p$, $\exists_{(c=v)\in\vec{F}} c = A$, $V \in \vec{X}$ and $A \notin \vec{X}$. The source table is pivoted, there is a condition on the pivot attribute column, and the value column V participates in the foreign key.

$\mathsf{U_{PV}}(s, \mathbf{FK}(\vec{F}|T.\vec{X} \to \vec{G}|T'.\vec{Y})) =$

$\quad\quad \mathbf{FK}(\vec{F} - \{(c = v)\}|\overline{T_p}.(\vec{X} - \{V\} \cup \{v\}) \to \vec{G}|T'.\vec{Y})$

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

D. ADDITIONAL EXAMPLES

D.1 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 input schema table t as a selection condition on the provenance column, and a projection down to the columns in the input 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.

D.2 HMerge Translation of Inserts

Let $\mathsf{U_{HM}}$ be the update function for the CT $HMerge(f, \overline{T}, \overline{C})$, and let s be its input schema. We define the action of $\mathsf{U_{HM}}$ on an Insert statement $\mathbf{I}(t, \vec{C}, Q)$ where $t \models f$ as follows:

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

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 10. Since the output consists entirely of insert statements, proving that $\mathsf{U_{HM}}$ respects the commutativity properties for insert statements reduces to showing that the newly added rows, when queried, appear in the input schema in their original form. In short, we 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.

D.3 Pivot Translation of Drop Element

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

$\mathsf{U_{PV}}(s, \mathbf{DE}(T, C, E)) =$

$\quad\quad$ If $C = A$, then $\mathbf{DC}(\overline{T}, E)$

$\quad\quad$ Else if $C = V$, then $\forall_{c\in\mathbf{Dom}(A)}\mathbf{DE}(\overline{T}, c, E)$

$\quad\quad$ Else, $\mathbf{DE}(\overline{T}, C, E)$

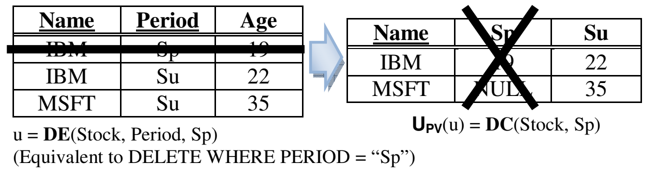


Figure 11: An example of a Drop Element statement translated by a Pivot CT

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.

E. EXAMPLE PROOF

To demonstrate how to prove the correctness properties of a CT, we offer as an example a proof for translating the Drop Element statement $\mathbf{DE}(T, A, E)$ through a Pivot CT $Pivot(T, A, V, \overline{T})$ — dropping an element from the attribute column, an example of which appears in Figure 11. We need to prove the second and third commutativity properties. To prove the second commutativity property, we 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:

$\mathsf{U_{PV}}(s, \{\mathbf{AT}(T, \vec{C} \cup \{A\}, \vec{D} \cup \{D' - \{E\}\}, \vec{K} \cup \{A\})\})$

$\quad = \mathsf{U_{PV}}(s, \{\mathbf{AT}(T, \vec{C} \cup \{A\}, \vec{D} \cup \{D'\}, \vec{K} \cup \{A\}), (\mathbf{DE}(T, A, E_0))\})$.

Proof: $\mathsf{U_{PV}}(s, \mathbf{AT}(T, \vec{C} \cup \{A\}, \vec{D} \cup \{D' - \{E\}\}, \vec{K} \cup \{A\}))$

$= \mathbf{AT}(\overline{T}, (\vec{C} - \{V\}) \cup D' \cup \{E_0\},$

$\quad\quad \vec{D} - \{\mathbf{Dom}(V)\} \cup \{\forall_{a\in D'-\{E\}}\mathbf{Dom}(V)\}, \vec{K})$

(Push the Add Table statement through the Pivot)

$= \mathbf{AT}(\overline{T}, (\vec{C} - \{V\}) \cup D', \vec{D} - \{\mathbf{Dom}(V)\} \cup \{\forall_{a\in D'}\mathbf{Dom}(V)\}, \vec{K}),$

$\quad\quad \mathbf{DC}(\overline{T}, E, \mathbf{Dom}(V))$

(DDL equivalence)

$= \mathsf{U_{PV}}(s, \{\mathbf{AT}(T, \vec{C} \cup \{A\}, \vec{D} \cup \{D'\}, \vec{K} \cup \{A\}), \mathbf{DE}(T, A, E)\})$

(View the statements in their pre-transformation image)

□

Next, we need to prove the commutativity property from Figure 5(c):

Proposition: Let s be a schema with T defined. Then:

$\mathsf{Q_{PV}}(\mathbf{DE}(T, C, E)(s), q^T)(\mathbf{DC}(T, E)(\mathsf{S_{PV}}(s))) \equiv q^T(\mathbf{DE}(T, C, E)(s))$

Proof: $\mathsf{Q_{PV}}(\mathbf{DE}(T, C, E)(s), q^T)(\mathbf{DC}(T, E)(\mathsf{S_{PV}}(s)))$

$= (\not\boxtimes_{\mathbf{Dom}(A)-\{E\};A;V}\overline{T})(\mathbf{DC}(\overline{T}, E)(\mathsf{S_{PV}}(s)))$

(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)

□