Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 63 (2007) 774–794
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Querying through a user interface

James F. Terwilliger [a,*], Lois M.L. Delcambre [a], Judith Logan [b]

[a] Department of Computer Science, Portland State University, Portland, OR 97207, USA
[b] Department of Medical Informatics and Clinical, Epidemiology, School of Medicine, Oregon Health and Science University, Portland, OR 97239, USA

Received 7 April 2007; accepted 16 April 2007
Available online 29 April 2007

Abstract

In contrast to a traditional setting where users express queries against the database schema, we assert that the semantics of data can often be understood by viewing the data in the context of the user interface (UI) of the software tool used to enter the data. That is, we believe that users will understand the data in a database by seeing the labels, drop-down menus, tool tips, or other help text that are built into the user interface. Our goal is to allow domain experts with little technical skill to understand and query data. In this paper, we present our GUi As View (Guava) framework and describe how we use forms-based UIs to generate a conceptual model that represents the information in the user interface. We then describe how we generate a query interface from the conceptual model. We characterize the resulting query language using a subset of the relational algebra. Since most application developers want to craft a physical database to meet desired performance needs, we present here a transformation channel that can be configured by instantiating one or more of our transformation operators. The channel, once configured, automatically transforms queries from our query interface into queries that address the underlying physical database and delivers query results that conform to our query interface. In this paper, we define and formalize our database transformation operators. The contributions of this paper are that first, we demonstrate the feasibility of creating a query interface based directly on the user interface and second, we introduce a general purpose database transformation channel that will likely shorten the application development process and increase the quality of the software by automatically generating software artifacts that are often made manually and are prone to errors.
© 2007 Elsevier B.V. All rights reserved.

Keywords: User interface; Query interface; View updates; Schematic heterogeneity

1. Introduction

The user interface for a data capture tool is typically designed to be easy to use by users who are knowledgeable in the application domain. For example, a great deal of effort goes into making sure that the user interface of medical software can be understood by clinicians and other medical staff. However, if the same

* Corresponding author.
E-mail addresses: jterwill@cs.pdx.edu (J.F. Terwilliger), lmd@cs.pdx.edu (L.M.L. Delcambre), loganju@ohsu.edu (J. Logan).

0169-023X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2007.04.007

users want to run queries over the data that is captured by this software, there are generally only two options available: have someone write a special query interface, or use SQL to express queries against the database schema. The first option requires developer time and effort, and the second option presents two challenges: (1) the user must learn SQL and (2) the user must understand the semantics of the data based on the structure and names that appear in the schema (perhaps with additional documentation, e.g., in a data dictionary). It is possible for someone to be well-versed in clinical terminology and medical procedure, but not have the skill to use SQL. More than that, the physical database, as described in the schema, may bear little resemblance to the structure and names of the data fields as they appear in the user interface. We would like to leverage the effort that goes into making useful user interfaces to create a query interface that can be used by expert users.

Our research has two primary goals. First, we want to use the user interface of the software tool that creates the data directly as the conceptual model for users, and allow the users to express queries against the resulting conceptual model. Second, we want to provide a mechanism that accommodates a broad range of physical database designs but hides that complexity from the users of the new query interfaces. Our GUi As View (Guava) framework achieves both of these goals by allowing users to pose queries effectively against the user interface and then rewriting the query into an equivalent form that one can execute against the physical database.

In this paper, we introduce the Guava conceptual model, describe how Guava creates a query interface and processes queries, and formally define the transformation operators that can appear in the channel. The remainder of this paper is organized as follows. Section 2 provides the motivation for this work. The Guava framework and conceptual model is presented in Section 3. The process for writing queries in Guava is described in Section 4. Section 5 formally defines the transformation channel, the Guava artifact that handles all communication between the user interface (and query interface) and the underlying database. Section 6 introduces ways to extend the channel as well as make it more efficient. Section 7 briefly discusses related work, and the paper concludes with a discussion of contributions and future work in Sections 8 and 9, respectively.

2. Motivation

The development of Guava is motivated by our work with the Clinical Outcomes Research Initiative (CORI) [1]. CORI seeks to improve the practice of endoscopy by conducting retrospective studies on de-identified patient data (i.e., endoscopy reports). To this end, CORI develops and distributes a software reporting tool that allows the clinician to enter data that describes endoscopic procedures and then generates endoscopy reports suitable for inclusion in the patient medical record. Endoscopy reports from nearly 70 sites across the US are being compiled by CORI in a data warehouse on an ongoing basis.

CORI supports a number of data analysts who conduct various retrospective studies. A retrospective study is an attempt to study data that has already been collected for other purposes – in this case, the reports that have been collected through the CORI software by physicians over the course of patient care. Each study requires that the analyst select an appropriate subset of the reports in the warehouse, classify the source data into categories of interest in the study, as appropriate, and then hand off the selected data, post-classification, for analysis in a statistical package.

Before they can perform a study over the data in a data warehouse, the analyst must first understand the data and schema. Understanding a database can be arbitrarily difficult, and the CORI warehouse is no exception; the database was designed for speed and space concerns and not human readability. The table and column names do not give hints as to the meaning of their contents. The analysts have a data dictionary that was written by a database developer, but that developer has long since left the organization. The dictionary has long been neglected and now no longer reflects the contents of the warehouse, and none of the current developers have the expertise, time, or specific knowledge about the warehouse to update it.

There are two additional reasons why it is difficult for the CORI analysts to query the data. One is that the native format of the data in the CORI software's database is in a generic format, where all data is stored as key-attribute-value triples, and where data that corresponds to a number of different entity classes are stored in the same table. After selecting only those rows that refer to the needed attributes, one must pivot the data to make it useful to the studies. Pivoting refers to the process of taking data out of a generic layout and into the

more familiar structure where attribute names are part of the schema (columns) rather than in the data. The situation where a schema is generic but the desired schema is not is called *schematic heterogeneity* [8]. Some effort [6] has been devoted to adding features to the SQL query language that support higher-order functions such as variables that range over columns or tables. The resulting language solves the schematic heterogeneity problem, but is difficult for database experts – let alone ordinary users – to master.

The second problem encountered by the analysts is that the data in the CORI database is encoded. Some data that represents long strings of text, such as "patient unconscious after procedure", was reduced to a small integer, such as "0", representing the index of a radio list or drop-down. For another example, the database stores a single integer that was the bitmask of the values of many Boolean values, such as the answers to a collection of checkboxes. Thus, the structure of the data and schema in the CORI database differs significantly from what the analysts need to perform studies.

To solve these problems, the same expert that put together the original data dictionary also assembled a program that separated and pivoted the data into tables in a non-generic format and decoded some of the data, as well as filtered out any records that contained errors. This program resulted in a data warehouse that the CORI analysts could use without needing to perform queries any more complicated than conjunctive queries. However, since the author of the program is no longer available and no one knows exactly how it runs – the exact semantics of that program are now lost. CORI would like to rewrite the program from scratch, but a fundamental problem still remains. CORI would be charging technical personnel – with no knowledge of how studies work – with the task of transforming and cleaning the data, two processes that can have profound impact on the data's semantics.

In short, the CORI analysts suffer from the following problems that affect the efficiency of their work and the reliability of their studies:

- the analysts need a comprehensive understanding of the contents of the data warehouse, but both the data and the schema are arcane,
- the primary data repository has a generic data structure and encoded data values, which makes querying difficult, if not impossible, and
- fixing either of the previous problems requires time from and communication with the development staff, and puts decisions that affect the semantics of the data in the hands of people who do not themselves run studies and thus may not appreciate the semantics.

The Guava architecture [14] solves these problems in two ways. First, Guava takes the original user interface for any forms-based application and creates a query interface that resembles it. We assume that, because CORI analysts are domain experts and can understand clinical terminology, they can understand the data-entry forms in the CORI software. So, creating a query interface that looks just like the forms that the physician used to enter the data gives the analyst an interface that can be learned quickly. Second, Guava allows developers and database professionals to express the relationship between two database schemas as a sequence of high-level operations, each with formal semantics. Unlike existing technologies such as Extract-Transform-Load (ETL) [15], these operations are bi-directional and can handle all traffic between a user interface and its physical database, allowing for flexibility in physical design while hiding the complexity from the analysts. With Guava, analysts can knowledgeably perform queries without ever contacting technical personnel, regardless of the complexity of the physical database.

3. Introduction to guava and the conceptual model of the user interface

This section introduces the GUi As View (Guava) framework. As opposed to the typical case (Fig. 1a) where users must either use SQL or a custom, separately-designed query interface to access the data, the query interface in Guava (Fig. 1b) is automatically generated from the user interface. First, the complete structure of the user interface is represented in a conceptual model – a hierarchical structure called a Guava-tree (*g*-tree). Guava automatically generates a *g*-tree from the user interface controls based on our extensions to an integrated development environment. Next, Guava translates a *g*-tree into a simple relational table structure with what we call a *natural schema*. Finally, a database designer can transform the natural schema into the

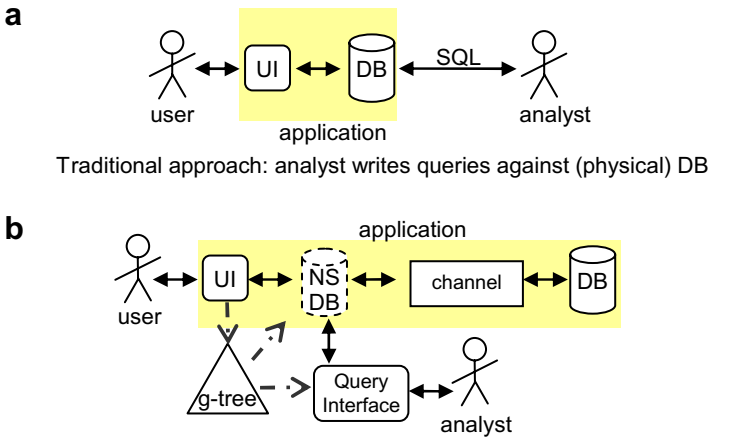


Fig. 1. The GUi As View (Guava) software engineering framework.

underlying physical database schema by instantiating our database operators. A collection of these operators form a *channel* that transforms the natural schema into the desired physical schema (at DB design time) and transforms simple queries from the application-UI-based query interface from the natural schema to the physical schema (at run time).

The Guava framework seeks to exploit the hierarchical nature of forms-based user interfaces to provide a simple representation of its information. Fig. 2 shows an example user interface that is hierarchical; the controls on each screen form a hierarchy based on the "contains" relationship, and the forms of an application are structured as a hierarchy because each form is launched from an event on another, save for the form that appears at application launch that serves as the root. In Fig. 2, the form on the right is launched by the details button of the form on the left. Guava then uses this simple representation to create a query interface that resembles the original user interface. The rest of this section explains the steps that Guava uses to generate a *g*-tree.

A Guava-tree (*g*-tree) represents the information present on a user interface, including the relationships between forms. Also of interest are the *context elements* for the controls, such as the control's type (e.g., text

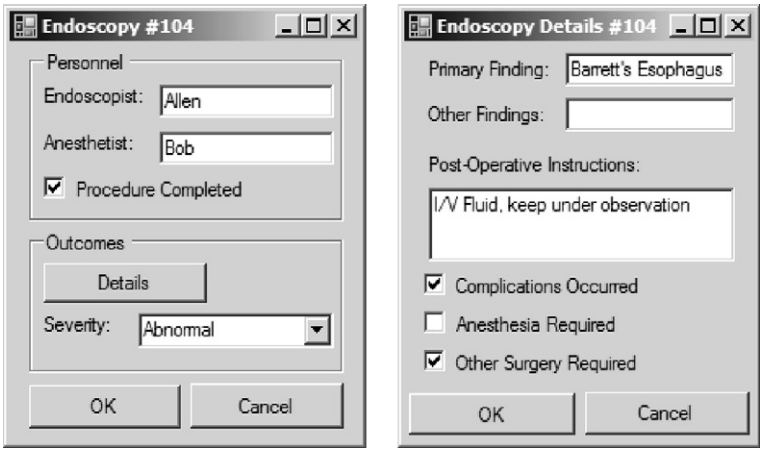


Fig. 2. A simple forms-based application with two forms. The second form provides additional details for the same person represented by the first form, and that the second form appears by clicking the first form's 'Details' button.

box or checkbox), its default value, and its text. A control's text may be simple to find for checkboxes and group boxes, where the text is simply part of the control, but is often harder to find for text boxes and drop-down lists where the text is actually in an adjacent label. These context elements are informative for anyone using the application, and for users that want to query the data.

Formally, a *g*-tree is a tree with a set of nodes N and a set of edges E such that:

- Each $n \in N$ is labeled with one of the values *Entity*, *Attribute*, *Container*, or *Control*.
- Each $n \in N$ has an attribute *Name* whose value is unique in the tree.
- Each $n \in N$ has a partial function h: *String* $\Rightarrow$ *String* that associates context element names with the value of that context element for the control.
- Each $e \in E$ is labeled with one of the values *Contains*, *Single-launch*, or *Multiple-launch*.

To illustrate the context element function h in action, consider the check box on the first form in Fig. 2. It has several obvious context elements, such as control text, default value, size, and location. So, for the *g*-tree node associated with that check box, the function h is defined as follows:

- $h(\text{``}ControlText\text{''}) = \text{``}Procedure\ Completed\text{''}$
- $h(\text{``}DefaultValue\text{''}) = \text{``}False\text{''}$
- $h(\text{``}Size\text{''}) = \text{``}(100, 10)\text{''}$
- $h(\text{``}Location\text{''}) = \text{``}(10, 90)\text{''}$

Each type of control supports its own set of context elements. Some context elements, such as size and location, will exist for all control types. Others, such as text length, will only be supported by certain control types (in this case, text boxes).

In addition, every attribute node a in a *g*-tree has a domain, denoted as *Domain*(a). The domain of a can be one of the following:

- Any subset of one of the standard atomic data type domains, including Boolean, Integer, Real, and String.
- A reference to an entity node e, coupled with a view expression v. This relationship is denoted as $Domain(a) = EntitySet(e)$.

The view expression v produces a string representation of an entity. Formally, v is a function v: *Nodes* $\rightarrow$ *String* that takes as input a subset of the attribute nodes that are descendants of an entity node (without going through another entity node) and produces a user-friendly representation of the referenced entity. For instance, if the view expression is the function $v(FirstName, LastName) = \text{``}LastName, FirstName\text{''}$, then in that control you will see values that look like "Thomas, Bob" when in fact the control stores an arcane object ID value referencing a row in another table.

We also define the following useful functions over *g*-trees:

- For any *g*-tree g, *rootnode*(g) is the root node of the *g*-tree.
- For any node n, *Entity*(n) is the nearest entity node to n above it in the *g*-tree, including n itself. Conceptually, since n represents a control on a form, *Entity*(n) is the node corresponding to that form. By definition, $Entity(n) = n$ if n is an entity node.
- For any entity node e, *Attributes*(e) is the collection of attribute nodes that are descendants of e without following a path through another entity node. Conceptually, since e represents a form, *Attributes*(e) represents all of the controls on that form that display data elements.
- For any entity node e, *Parent*(e) is the nearest entity node to n strictly above it in the *g*-tree. Conceptually, since n represents a form, *Parent*(n) represents the form that launched it. By convention, $Parent(e) = null$ if e is the root node of a *g*-tree.

Translating a user interface into a *g*-tree is straightforward. Each form in the user interface becomes an entity node, each data-bound control becomes an attribute node, each container control (such as a group

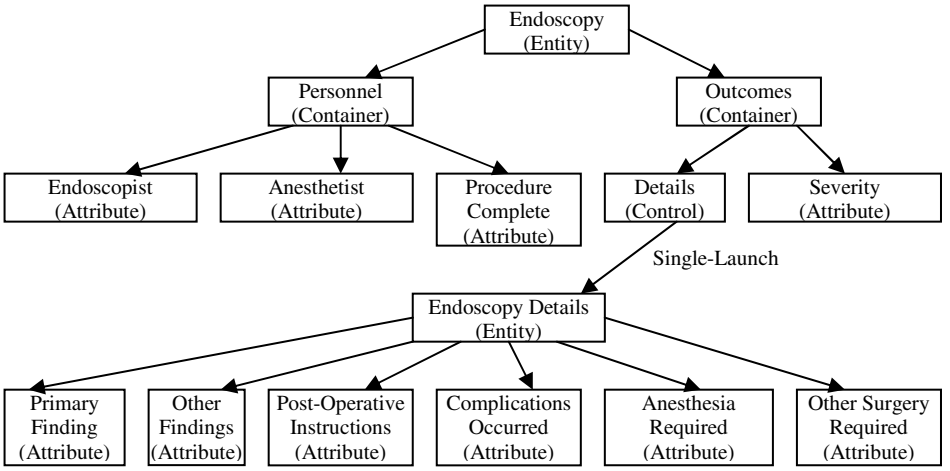


Fig. 3. An example *g*-tree corresponding to the application in Fig. 2. Any edge that is not labeled is a *Contains* edge.

box) becomes a container node, and everything else becomes a control node. The name of each node is derived from the name the developer gives the control in the application code. If one form or control c_1 contains another control c_2 (e.g., a group box containing a text box), a *Contains* edge is drawn from the node for c_1 to the node for c_2. If a control launches another form, but the new form merely contains more details about the first form, a *Single-Launch* edge is drawn from the control to the form. If, instead, the new form allows creation of several instances for each instance of the first form, a *Multiple-Launch* edge is drawn.

Fig. 3 shows the *g*-tree that corresponds to the UI in Fig. 2. The parent form turns into the root of the tree, its group boxes are its children, and the controls in the group boxes become child nodes of the group box nodes. The child form also becomes a node, and is a child node of the control that launched it. The edge is labeled as single-launch because the child and parent forms share a one-to-one relationship; there is no new/edit/delete functionality in the parent form and only a single button launches the form.

Notice that not every *g*-tree corresponds to a working user interface. For instance, a single-launch edge leading to an attribute node does not make sense, because that implies clicking a button launches a text box or a checkbox, not another form. We define a *g*-tree to be *valid* if it satisfies these properties:

- The root node of the tree is of type *Entity*.
- The in-edge for all non-root *Entity* nodes is of type *Single-launch* or *Multiple-launch*.
- The in-edge for all non-*Entity* nodes is of type *Contains*.
- The out-edges for all *Entity* nodes are of type *Contains*.

In Guava, we generate what we call a *natural schema*, a relational schema where each form corresponds to a single table using the following algorithm:

Algorithm 1. To translate a valid *g*-tree (N, E) into its natural database schema:

- For each *Entity* node $n \in N$, create a table with name *n.Name*, and add a column called *id*, an artificially-generated primary key.
- For each *Entity* node $n \in N$ that is not the root node, let $p = Parent(e)$. If *n*'s in-edge is of type *Single-launch*, create a foreign key from (*n.Name*).*id* to (*p.Name*).*id*. If *n*'s in-edge is of type *Multiple-launch*, create a new column (*n.Name*).*fk* and a foreign key from the new column to (*p.Name*).*id*.
- For each *Attribute* node $a \in N$, let $p = Entity(a)$. Create a column named *a.Name* in *p*'s table. If *a* has domain *EntitySet*(*e*) for some node *e*, then set the new column's domain to be the domain of artificially-generated keys, and create a foreign key from *a* to *e.id*. Otherwise, set the domain of the new column to *Domain*(*a*).

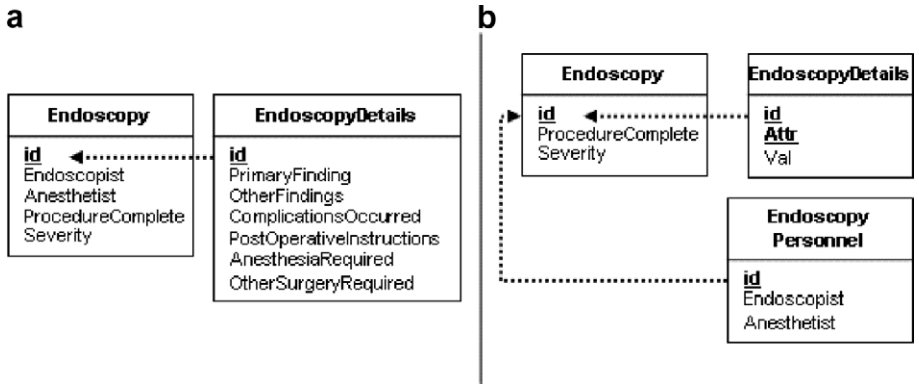


Fig. 4. The natural schema corresponding to Fig. 3, before (a) and after (b) the application of a channel with operators Unpivot (EndoscopyDetails, {id}) and VPartition(Endoscopy, {id}, {ProcedureComplete, Severity}, EndoscopyPersonnel).

Fig. 4a shows the result of running Algorithm 1 on the *g*-tree in Fig. 3, and is consequently the natural schema corresponding to the forms in Fig. 2. The natural schema can also serve as the schema of the underlying physical database; however, it is more likely that the schema on disk will be significantly different, to accommodate physical design issues such as retrieval and update speed. Fig. 4b shows a possible physical database schema that contains the same data as Fig. 4a, but after transformation by a channel, to be discussed in Section 5.

4. Queries in Guava

The Guava query interface is graphical in nature, and resembles the original interface very closely, as shown in Fig. 5a. Because a *g*-tree contains all of the context information for an application, Guava uses it to generate the query interface for the application. To create a query using the interface in Fig. 5a, the user navigates the forms just as they would the original user interface. The user specifies which attributes to return by right clicking on a control and selecting the ''print'' option. The user also specifies the conditions on which to filter rows by filling in data into the various data fields, similar to QBE [19]. Unlike QBE, the user does not need to specify any kind of join conditions, since the joins are implied by the relationships between forms. This interface is not finished yet, but when it is complete, the user will have the option of viewing the results in the context of the forms or in a table.

Other query interfaces are also possible, such as one that exposes the controls in an entire application in a single tree as shown in Fig. 5b, so long as the context elements for each node are available to the user. The interface in Fig. 5b is similar to the previous interface, except that the user prints a data control by checking the box next to that control's name in the tree. The context of each control, as well as the interface to specify conditions, can be viewed in a separate window. The user can also search the tree for data controls; the interface takes a search term and finds all occurrences of the term in the context elements for each control. The interface highlights and expands each matching control, and lists the context elements that matched the term.

Guava takes the query as specified by the user and translates it into relational algebra against the natural schema. The first step in this process is to accumulate the query into a data structure.

Definition. A *decorated* *g*-tree is a *g*-tree where, to each entity and attribute node, we attach two additional pieces of data:

- A boolean value *Print* representing the user's decision to return the value of the attribute (or ID for the entity), similar to the print flag in QBE.
- A boolean function *Condition* in the form of $Node\theta Value$ where *Value* is any constant value, *Node* is the current node, and θ is any of the standard six comparison operators. The function can also be a conjunction of such expressions, or *True* to indicate that there is no filter for that node.

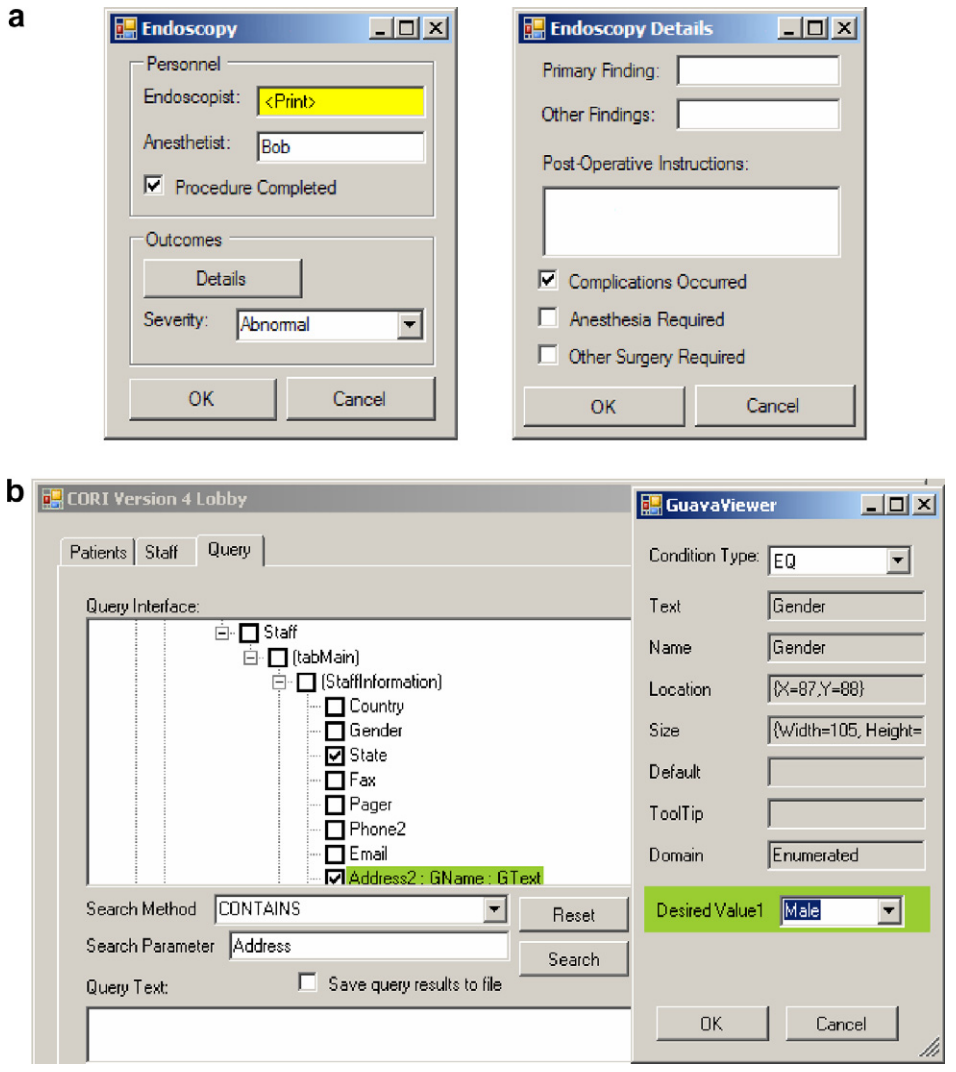


Fig. 5. Two possibilities for the Guava query interface. One interface would exactly mock up each form (a), where each data entry control now becomes a place to enter print and filter statements. Another interface would create a tree-structure (b) that mimics the structure of the *g*-tree.

Intuitively, Guava constructs a decorated *g*-tree by taking the specified print statements and filtering conditions from the query interface and attaching them to the node in the *g*-tree representing the control on which they were found. We construct the formal definition of a query in the Guava architecture as follows:

Definition. A node n in a decorated *g*-tree is *non-trivial* if its $n.Print = true$ or its boolean condition expression is anything other than "*true*".

Definition. A *g-subtree* of a *g-tree* g is a subtree of g where the root node is an entity node. A *pruned* *g*-tree is the smallest g-subtree of a decorated *g*-tree that contains all of the non-trivial query nodes. This corresponds with the idea that a query may be able to start with a sub-form of the application, and not have to start with the root form.

Definition. A *Guava query* over a *g-tree* g is a forest of g-subtrees of g where all of the following is true:

(1) One of the *g*-trees g is set aside as a "distinguished" tree, and g is pruned.
(2) Every other tree t in the forest can be associated with an attribute node a in another tree where $Domain(a) = EntitySet(rootnode(t))$.

(3) No two trees are associated with the same attribute node in the same tree.
(4) The associations between trees forms a tree. In other words, if we were to take each associated attribute/entity node pair and create an edge between the two, the resulting structure would still be a tree.
(5) At least one of the nodes in the forest has a boolean print value set to "true".

In addition, the query also provides a function U that takes entity nodes in the forest and maps them to a unique name. We need this function because the subtrees in the forest may overlap, so we need some way to disambiguate different instances of the same entity nodes.

Finally, we define $SingleTableQuery(e) = \rho_{e.Name \rightarrow U(e)}(\sigma_V(e.Name))$, where e is an Entity node in a decorated g-tree, ρ is the relational renaming operator and V is the conjunction of all boolean condition expressions in $Attributes(e)$ and that of e itself. With this machinery in place, we can translate a Guava query into relational algebra.

Algorithm 2. To translate a Guava query q over g-tree g into relational algebra expressed against the natural schema of g:

(1) Begin with the root node r of the distinguished g-subtree in the query. Set $TQ = SingleTableQuery(r)$.
(2) Also, if $Print(r) = True$, set $TP = \{U(r).ID\}$. Otherwise, set $TP = \emptyset$.
(3) Traverse the subtree in depth-first fashion, visiting the children of each node in any order.
(4) For each entity node e found with a *Single-Launch* in-edge, set $TQ = TQ \bowtie_{U(Parent(e)).ID = U(e).ID} SingleTableQuery(e)$.
(5) For each entity node e found with a *Multiple-Launch* in-edge, set $TQ = TQ \bowtie_{U(Parent(e)).ID = U(e).FK} SingleTableQuery(e)$.
(6) For each entity node e found with $Print(e) = True$, set $TP = TP \cup \{U(e).ID\}$.
(7) For each attribute node a found that is associated with another g-subtree with root node e by relationship $Domain(a) = EntitySet(rootnode(e))$, set $TQ = TQ \bowtie_{U(Entity(a)).a = U(e).ID} SingleTableQuery(e)$. Then, proceed to traverse the g-subtree in depth-first fashion. When traversal is complete, return to the previous g-subtree. In this fashion, the entire forest of g-subtrees will be traversed because of the tree structure of associations between them.
(8) For each attribute node a found with $Print(a) = True$, set $TP = TP \cup \{U(Parent(a)).(a.Name)\}$.
(9) Once traversal of the entire forest is complete, return $\pi_{TP}(TQ)$.

By construction, we see that the resulting query language is equivalent to relational algebra restricted to the following:

- T for any table in the natural schema
- $Exp1 \bowtie Exp2$ only if there is a foreign key from $Exp2$ to $Exp1$
- $\pi_C(Exp)$ for any collection of columns
- $\sigma_{C\theta V}(Exp)$ for any column C, value V, and comparator θ

In other words, the Guava query language is equivalent in expressive power to single-statement conjunctive queries where joins are restricted to foreign keys and selection can use any of the six comparators to relate a column against a constant, but not a column against another column.

5. Guava channels: formal foundations

The Guava back-end allows queries expressed against the natural schema to be processed against an underlying physical database whose schema may be different. Our approach is to define database transformation operators that allow a database designer the freedom to structure the physical database as he or she chooses. In the formal definition that we present here, instances of these operators provide automatic generation of the physical database schema, as well as automatic translation of the data input from the user interface for storage

Table 1
Descriptions of eight database transformation operators

Operator	Description
Apply	Uses an invertible function to transform the values in a table
Rename	Uses an invertible function to rename the tables or columns in the schema
VPartition	Partitions a table vertically into two tables, and creates a foreign key between the two
VMerge	Joins two tables together, provided that the key for one table serves as a foreign key for the key of the other
HPartition	Partitions a table horizontally based on the values in a given column
HMerge	Merges a collection of union-compatible tables, using a new column to keep track of the table from which each tuple came
Pivot	Transforms a table so that values appearing in a specific column become column headings
Unpivot	Transforms a table so that a group of column names become data values

in the physical database. The operators are invertible, so they also describe how to translate the physical data back into the natural schema to make it available for query processing.

Table 1 lists the transformation operators that Guava provides, with a short description. The parameters required to instantiate each database transformation operator are present in Table 2. Table 3 formally describes how each operator transforms a schema and an instance of a database. Each operator takes a schema or database instance as input, and produces a schema or database instance as output. The operator definitions use the following notation:

- **Tables**: The set of tables in the schema
- **Cols**(T): The set of columns in the schema for table T
- *inst*(T): The instance of table T
- *Name*(D): The name of the column or table D, returned as a data value
- A subscript of "in", such as $inst_{in}$, refers to the input of the operator
- A subscript of "out", such as $inst_{out}$, refers to the output of the operator
- Anything in **boldface** is a set
- *LOJ* refers to a left-outer join

The operators defined in Table 3 have an inverse operator, as shown in Table 4. A DB designer can specify the sequence of operators necessary to transform the natural schema associated with the user interface into the stored DB. Once the sequence of transformation operators is specified, we show that the stored database instance can be transformed into an instance of the natural schema, using the sequence of inverse operators in reverse order.

Definition. A *cell* is an instantiation of a database operator with specific input parameters provided. A cell includes an instantiation of the operator's inverse. As shown in Table 4, some parameters for the inverse operator in a cell are taken from the input parameters to the operator in this cell. Thus, a cell provides a mechanism to "remember" specific details of the original transform operator. For example, the HPartition operator produces a set of new tables ($\mathbf{Ts}^{\mathbf{result}}$) from a given table ($T$). The inverse operator, HMerge, within the cell uses these table names (merging $\mathbf{Ts}^{\mathbf{result}}$ to produce T). To apply a cell *forward* is to apply the operator; to apply a cell in *reverse* is to apply the inverse operator.

Theorem 1. *Given a cell with operator O from Table 1 with specific parameters P and the inverse operator O' from Table 1 with parameters P' as shown in Table 4, $S = O'(P')(O(P)(S))$ and $D = O'(P')(O(P)(D))$ for any schema S and its instance D on which the operator O is valid. Note that $O(P)$ (as well as $O'(P')$) returns an instantiated operator that can then be applied to either a schema or an instance.*

Proof. The claim is true for Apply and Rename because the particular functions (shown as f in Table 3) are invertible and defined for all input (from the natural schema). The fact that the claim is true for the remaining operators follows from their definitions in relational algebra as shown in Table 3. □

Table 2
Eight transformation operators, their parameters and restrictions

Operator	Input parameters
Apply	T: The input table name, $T \in \mathbf{Tables_{in}}$ $\mathbf{C}_1$: A set of column names, $\mathbf{C}_1 \subset \mathbf{Cols_{in}}(T)$ $\mathbf{C}_2$: A set of column names, $\mathbf{C}_2 \cap \mathbf{Cols_{in}}(T) = \emptyset$ f: A total, invertible function with $\lvert\mathbf{C}_1\rvert$ inputs and $\lvert\mathbf{C}_2\rvert$ outputs
Rename	f: A total, invertible function whose input and output are string values
VPartition	T: The input table name, $T \in \mathbf{Tables_{in}}$ **Ks**: The set of key column names, $\mathbf{Ks} \subset \mathbf{Cols_{in}}(T)$ **Cs**: A set of non-key column names from T, $\mathbf{Cs} \subset (\mathbf{Cols_{in}}(T) - \mathbf{Ks})$ T^{new}: The name of the new table, $T^{new} \notin \mathbf{Tables_{in}}$
VMerge	T^L: The primary input table name, $T^L \in \mathbf{Tables_{in}}$ T^R: The name of the table to merge, $T^R \in \mathbf{Tables_{in}}$ **Ks**: A set of column names where **Ks** comprises the key for both T^L and T^R and a foreign key exists from T^R.**Ks** to T^L.**Ks**
HPartition	T: The input table name, $T \in \mathbf{Tables_{in}}$ C: A column name, $C \in \mathbf{Cols_{in}}(T)$ whose values will be used to distribute the table's tuples $\mathbf{Ts}^{result}$: The set of output table names, $\mathbf{Ts}^{result} \cap \mathbf{Tables_{in}} = \emptyset$
HMerge	**Ts**: A set of table names, $\mathbf{Ts} \subseteq \mathbf{Tables_{in}}$ where the tables in **T** are union-compatible T^{result}: The output table name, $T^{result} \notin \mathbf{Tables_{in}}$ C: A column name such that, $\forall T \in \mathbf{Ts}$, $C \notin \mathbf{Cols_{in}}(T)$
Pivot	T: The input table name, $T \in \mathbf{Tables_{in}}$ **Ks**: The set of key column names from T A: The name of a column holding attribute names, $A \in \mathbf{Ks}$ V: The name of a column holding attribute values, $V \notin \mathbf{Ks}$
Unpivot	T: The input table name, $T \in \mathbf{Tables_{in}}$ **Ks**: The set of key column names from T where the columns $(\mathbf{Cols_{in}}(T) - \mathbf{Ks})$ have the same data type A: The name of a column to hold attribute names, $A \notin \mathbf{Cols_{in}}(T)$ V: The name of a column to hold attribute values, $V \notin \mathbf{Cols_{in}}(T)$

Definition. A *channel* is a sequence of cells, as shown in Fig. 6. We use the notation $O_i(P_i)^{-1}$ to represent the inverse of $O_i(P_i)$ (the inverse was shown as $O'(P')$ in Theorem 1). Formally, for a channel C, $C = O_n(P_n) \circ \ldots \circ O_1(P_1)$, and $C^{-1} = O_1(P_1)^{-1} \circ \ldots \circ O_n(P_n)^{-1}$. We use the notation $C_{schema}(S)$ (or $C^{-1}_{schema}(S)$) for a forward (reverse) application of the cells in forward (reverse) order. Similarly, we use $C_{data}(D)$ and $C^{-1}_{data}(D)$ to indicate the application of a forward or reverse data transformation. C is a *valid channel* for a schema S and instance D if, when applied to S and D, none of the restrictions on the operators in C, as described in Table 2, are violated.

The following theorem guarantees that the original fully materialized natural schema is equivalent to the data returned from the physical database associated with a valid channel. This provides the foundation for query processing in Guava.

Theorem 2. *Given a schema S with instance D, and a channel C that is valid on S and D, $S = C^{-1}_{schema}(C_{schema}(S))$ and $D = C^{-1}_{data}(C_{data}(D))$.*

Proof. The proof follows from the fact that each cell in the channel represents an invertible function and the composition of invertible functions is invertible. Note, we expect that each operator in a channel to be total with respect to the input data that appear as input when using the channel in the forward direction. Thus, the function f used with the Apply operator must be defined for all values that can appear in the relevant attributes and the HPartition operator must have an input table name in $\mathbf{Ts}^{\mathbf{result}}$ for each possible value that can appear in the column upon which the HPartition is based. □

Table 3
Defining the action of eight transformation operators

Operator	Schema transformation	Data transformation
Apply $(T, \mathbf{C}_1, \mathbf{C}_2, f)$	$\mathbf{Cols_{out}}(T) = (\mathbf{Cols_{in}}(T) - \mathbf{C}_1) \cup \mathbf{C}_2$	$inst_{out}(T) = \{(g_1, \ldots, g_{n-m}, f_1(g_{n-m+1}, \ldots, g_n), \ldots, f_p(g_{n-m+1}, \ldots, g_n)) \mid (d_1, \ldots, d_n) \in inst_{\text{in}}(T)\}$ where $m = \lvert\mathbf{C}_1\rvert$, $p = \lvert\mathbf{C}_2\rvert$, $n = \lvert\mathbf{Cols_{in}}(T)\rvert$, $g_1, \ldots, g_{n-m} = \pi_{\mathbf{Cols_{in}}(T)-\mathbf{C}_1} d_1, \ldots, d_n$, $g_{n-m+1}, \ldots, g_n = \pi_{\mathbf{C}_1} d_1, \ldots, d_n$, and f_i is the i'th component of the output of function f
Rename (f)	$\mathbf{Tables_{out}} = \{f(T) \mid T \in \mathbf{Tables_{in}}\}$ $\forall T \in \mathbf{Tables_{in}}$, $\mathbf{Cols_{out}}(f(T)) = \{f(C) \mid C \in \mathbf{Cols_{in}}(T)\}$	None
VPartition $(T, \mathbf{Ks}, \mathbf{Cs}, T^{new})$	$\mathbf{Tables_{out}} = \mathbf{Tables_{in}} \cup \{T^{new}\}$ $\mathbf{Cols_{out}}(T) = \mathbf{Ks} \cup \mathbf{Cs}$ $\mathbf{Cols_{out}}(T^{new}) = \mathbf{Ks} \cup (\mathbf{Cols_{in}}(T) - \mathbf{Cs})$	$inst_{out}(T) = \pi_{\mathbf{Ks} \cup \mathbf{Cs}}(inst_{\mathbf{in}}(T))$ $inst_{out}(T^{new}) = \{t \mid t \in \pi_{\mathbf{Cols_{in}(T)}-\mathbf{Cs}}(inst_{\mathbf{in}}(T)) \wedge \exists C \in \mathbf{Cols_{in}}(\mathbf{T}) - \{\mathbf{Cs} \cup \mathbf{Ks}\}(t.C <> null)\}$
VMerge $(T^L, T^R, \mathbf{Ks})$	$\mathbf{Cols}(T^L) = \mathbf{Cols}(T^L) \cup \mathbf{Cols}(T^R)$ $\mathbf{Tables_{out}} = \mathbf{Tables_{in}} - \{T^R\}$	$inst_{out}(T^L) = inst_{in}(T^L) LOJ_{T^L.K=T^R.K \forall K \in \mathbf{KS}} inst_{in}(T^R)$
HPartition $(T, C, \mathbf{Ts}^{result})$	$\mathbf{Tables_{out}} = (\mathbf{Tables_{in}} - T) \cup \mathbf{Ts}^{result}$ $\forall T_r \in \mathbf{Ts}^{result}$, $\mathbf{Cols_{out}}(T_r) = \mathbf{Cols_{in}}(T) - \{C\}$	$\forall T_r \in \mathbf{Ts^{result}} inst_{out}(T_r) = \pi_{\mathbf{Cols_{in}}(\mathbf{T})-\{\mathbf{C}\}}(\sigma_{T.C=Name(T_r)}(inst_{in}(T)))$
HMerge $(\mathbf{Ts}, T^{result}, C)$	$\mathbf{Cols_{out}}(T^{result}) = \mathbf{Cols_{in}}(T) \cup \{Name(C)\}$, for any $T \in \mathbf{Ts}$ $\mathbf{Tables_{out}} = \mathbf{Tables_{in}} - \mathbf{Ts} \cup \{T^{result}\}$	$inst_{out}(T^{result}) = \cup_{T \in \mathbf{Ts}}(inst_{\text{in}}(T) \times \{(Name(T))\})$
Pivot $(T, \mathbf{Ks}, A, V)$	$\mathbf{Cols_{out}}(T) = \mathbf{Ks} \cup \mathbf{Ds}$ where $\mathbf{Ds}$ = the set of values in $inst_{\text{in}}(T.A)$ New key for T is $\mathbf{Ks} - \{A\}$	$inst_{out}(T) = \pi_{\mathbf{Ks}-\{A\}}(inst_{in}(T)) \bowtie \rho_{V \to D_1} \pi_{(\mathbf{Ks}-\{A\}) \cup \{V\}}(\sigma_{T.A=D_1}(inst_{in}(T))) \bowtie \ldots \bowtie \rho_{V \to D_n} \pi_{(\mathbf{Ks}-\{A\}) \cup \{V\}}(\sigma_{T.A=D_n}(inst_{in}(T)))$ for $\mathbf{Ds} = \{D_1, D_2, \ldots, D_n\}$
Unpivot $(T, \mathbf{Ks}, A, V)$	$\mathbf{Cols}_{out}(T) = \mathbf{Ks} \cup \{V, A\}$ New key for T is $Ks \cup \{A\}$	$inst_{out}(T) = \bigcup_{C \in (\mathbf{Cols_{in}}(T)-\mathbf{Ks})}(\pi_{Ks \cup \{C\}}(inst_{in}(T)) \times (Name(C)))$

Any table in the input schema or instance that is not explicitly referenced by the operator simply passes to the output unaffected.

Table 4
Defining the inverse of the eight transformation operators

Operator	Inverse
Apply (T, $\mathbf{C}_1$, $\mathbf{C}_2$, f)	Apply (T, $\mathbf{C}_2$, $\mathbf{C}_1$, f^{-1})
Rename (f)	Rename (f^{-1})
VPartition (T, **Ks**, **Cs**, T^{new})	VMerge (T, T^{new}, **Ks**)
VMerge (T^L, T^R, **Ks**)	VPartition (T^L, **Ks**, $\mathbf{Cols}_{\text{in}}(T^L) - \mathbf{Ks}$, T^R)
HPartition (T, C, $\mathbf{Ts}^{result}$)	HMerge ($\mathbf{Ts}^{result}$, T, C)
HMerge (**Ts**, T^{result}, C)	HPartition (T^{result}, C, **Ts**)
Pivot (T, **Ks**, A, V)	Unpivot (T, $\mathbf{Ks} - \{A\}$, A, V)
Unpivot (T, **Ks**, A, V)	Pivot (T, $\mathbf{Ks} \cup \{A\}$, A,V)

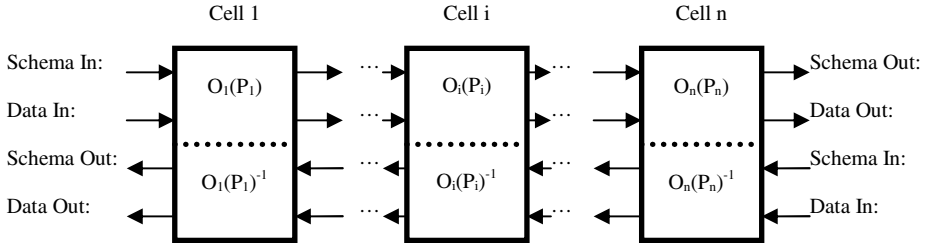


Fig. 6. Description of a channel, as used in our formal definition.

6. Extending the Guava channel

The formal definition of a channel as described in the previous section assumes that there is a full database instance – whose schema is the natural schema – materialized somewhere, and that communication to and from the physical database is done an entire database instance at a time. More than that, each channel operator materializes the entire database instance as well. This view of the channel works well for defining the semantics of each operator, but has serious practical limitations. For instance, it does not make sense to make a small update to the natural schema instance, then have that update propagate to the physical database by re-transforming the entire instance through the channel. Also, the operators listed in Tables 1–3 do not represent every possible transformation that one may want to perform between the UI and the physical database. We describe in this section the directions we are taking in implementing a channel in a prototype implementation. In the first section, we describe how to implement the channel in a more efficient way; in the second section, we add new capabilities to the channel by defining new operators.

6.1. New channel pathways

If one looks at a typical forms-based application, communication between its user interface and its physical database generally consists of the following:

- Retrieval of a small collection of data, for display in a single form – thus corresponding to a single row in one table in the natural schema.
- Retrieval of a larger collection of data to match a search criteria, such as searching for patients by last name.
- Inserts, updates, or deletes to information from a single form.

The first two of these actions take a query against the physical database, then take the data that is returned and transform it into a format for display in a form – in essence, putting the data into the natural schema. The third action is really like a single DML statement (e.g., INSERT) against the natural schema that the application middleware translates into (possibly multiple) DML statements against the physical schema.

With these use cases as motivation, we define a new schematic for a channel with three additional pathways (Fig. 7). Unlike the previous version of the channel as shown in Fig. 6, this version of the channel does not contain an inverse of each operator in each cell. Instead, the three new pathways (Update, Query, and Results) handle all communication between the natural schema and the physical database in both directions, so the inverse operators are not necessary. This version treats the natural schema as an updatable view of the data in the physical database. There may be some query results stored locally in the application while it is being viewed or updated, but for the most part, there is never a fully-materialized natural schema instance, nor a fully-materialized intermediate instance.

Our intent is to support queries, DDL and DML statements addressing the natural schema by modifying the query, DDL and DML statements in each cell. More than that, our goal is to do so on the inbound path (from the natural schema) as much as possible, requiring that the outbound results be transformed minimally in the cell if at all.

One new pathway through the five-pathway channel (marked as "Update" in Fig. 7) takes updates in the form of DML or even DDL statements against the natural schema and translates them into DML or DDL against the physical database. In formal terms, if C_{data} is the action of a channel on data, C_{update} is the action of a channel on an update statement, D is an instance of a database in the natural schema, and U is an update issued against D then we intend to prove:

$$C_{data}(U(D)) = C_{update}(U)(C_{data}(D)) \tag{1}$$

The remaining pair of new pathways (marked as "Query" and "Results" in Fig. 7) works on queries and their results. The query pathway takes the relational algebra output from algorithm 2 as input, then transforms the query into a query over the physical schema. The result pathway then takes the query answer and transforms it into a result that is equivalent to having run the original query against the natural schema. Again, in formal terms, if C_{data} is the action of a channel on data, C_{query} is the action of a channel on a query, C_{result} is the action of a channel on a query result, D is an instance of a database in the natural schema, and Q is a query against the natural schema, then we intend to show that:

$$C_{result}(C_{query}(Q)(C_{data}(D))) = Q(D) \tag{2}$$

These two properties ensure that the user sees the natural schema as if it were the real database, when it is in fact merely a view of the physical database. Property (1) says that an update against the natural schema will appear to the user as if the update were being performed on a materialized instance of the natural schema, when in fact Guava translates the update and runs it against the physical database. Property (2) guarantees that when the user issues a query against the natural schema, the answer appears to have been run against a materialize natural schema instance. Fig. 8 shows these two properties graphically.

For some database transformation operators, we are able to modify queries so that they deliver the query answer in the proper form without any additional processing of the results. For instance, the HMerge operator, no matter what its parameters, is able to take any conjunctive query and rewrite it so that it is equivalent to the original query when over the schema and data of the operator's output (Fig. 9a). In this case, there is no additional processing necessary to make the query answer consistent with the original query, so the results pathway of HMerge does nothing.

However, other operators require some additional processing in the results pathway. The unpivot operator is able to transform the query into an equivalent form using a number of left outer joins, but the

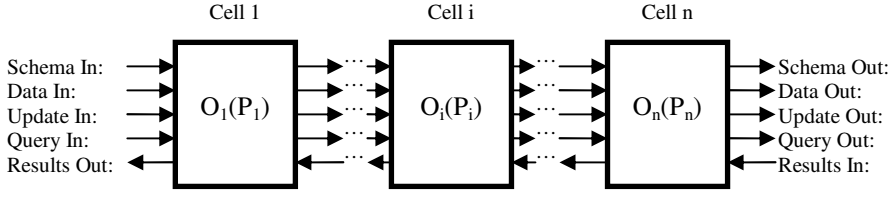


Fig. 7. Description of a channel, version 2, with three additional pathways.

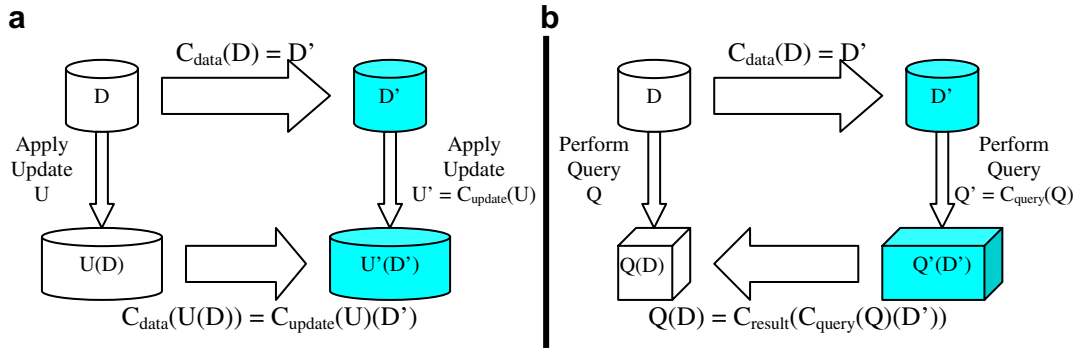


Fig. 8. Demonstrating channel properties 1 and 2.

resulting query can be very inefficient (Fig. 9b). There is an algorithm, however, that can "undo" the effects of the unpivot operator in just one pass of the data, assuming the data is sorted; even with the sort procedure, this algorithm can be orders of magnitude faster than the outer joins. So, in the case of unpivot, it is more efficient overall to produce a simpler query and allow the result pathway to process the result efficiently.

The Apply operator may or may not require results processing, depending on the function it uses (Fig. 9c). For such functions, the only way to produce an equivalent query result is to rely on the results pathway to transform the query answer by applying the inverse function on the data to produce the data that the user expects.

Our prototype of Guava implements the five-pathway channel. We are currently working on extending the definitions of the eight transformation operators to the update, query, and result pathways.

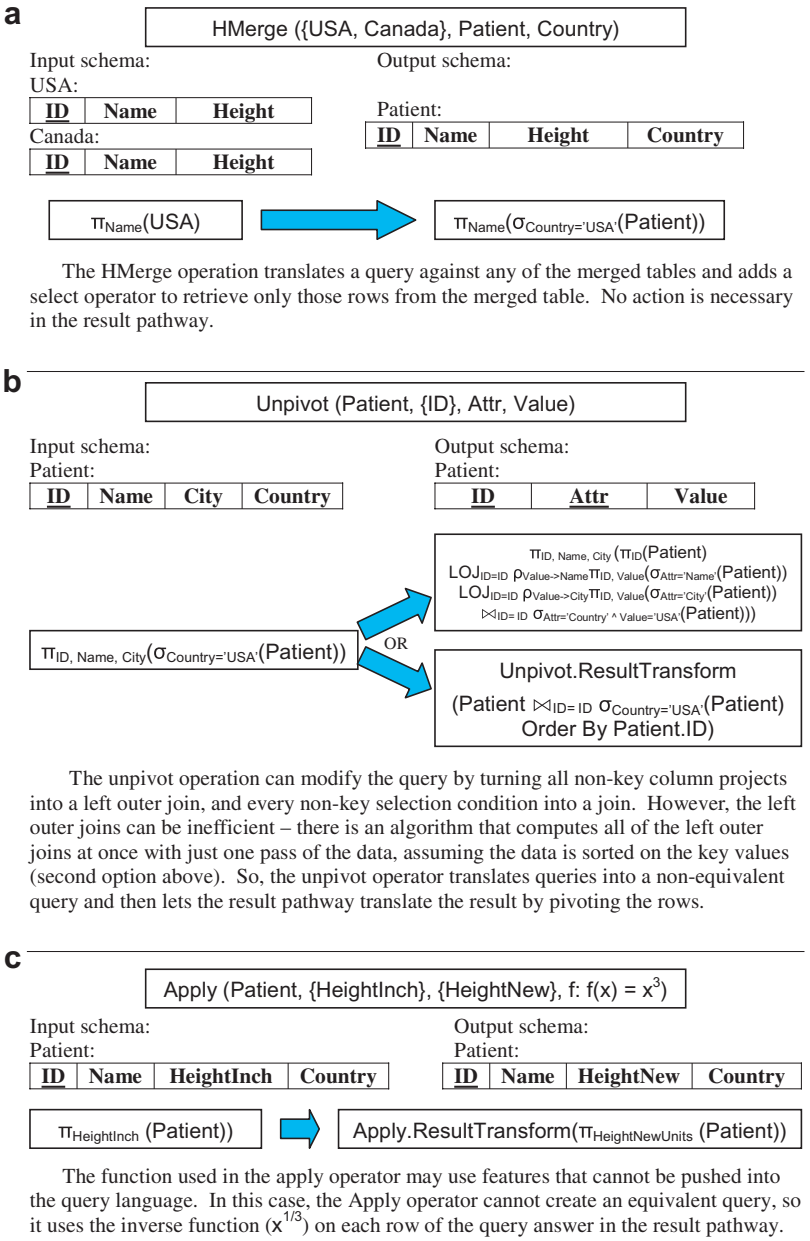
6.2. Semantic operators and the split channel

Consider the following actions:

- Adorn: Add additional columns containing environment information, such as system time or user name, and specify whether these values are to be updated either any time the row is updated or only when the row is inserted.
- Audit: Add temporal capabilities to the database, so that no information is ever deleted – rather than update or delete any rows, deprecate old rows and add new ones to reflect inserts and updates.
- ColumnEquate: Assert that two different columns, possibly from different tables, always hold the same value, implying that one of the columns may be dropped without losing information.
- TableAssert: Assert that the objects stored in one table are a subset of the objects in another table.
- CheckConstraint: Define a condition that must be true before an instance of a form can be accepted/committed.

These actions are considered part of the realm of *business logic*, which developers often consider a separate activity to user interface design, and thus possibly outside of the reach of Guava. However, it makes sense to model these actions as part of the channel because each of these actions alters schema, DML/DDL, queries and results just like the transformation operators. None of these actions can be represented using only the eight transformation operators in Table 1. Thus, each of these actions can be considered as a new channel operator in its own right, with actions defined for all five pathways. We call these new actions *semantic operators*.

The semantic operators are also similar to the transformation operators in that they are invertible, though not in the same way. The transformation operators are closed under inverse according to Table 4, meaning that the inverse of a transformation operator is also a transformation operator. The semantic operators are



The HMerge operation translates a query against any of the merged tables and adds a select operator to retrieve only those rows from the merged table. No action is necessary in the result pathway.

The unpivot operation can modify the query by turning all non-key column projects into a left outer join, and every non-key selection condition into a join. However, the left outer joins can be inefficient – there is an algorithm that computes all of the left outer joins at once with just one pass of the data, assuming the data is sorted on the key values (second option above). So, the unpivot operator translates queries into a non-equivalent query and then lets the result pathway translate the result by pivoting the rows.

The function used in the apply operator may use features that cannot be pushed into the query language. In this case, the Apply operator cannot create an equivalent query, so it uses the inverse function ($x^{1/3}$) on each row of the query answer in the result pathway.

Fig. 9. Explaining why translated queries may require processing of the results, and why we need a results pathway.

"invertible" in the sense that they can be undone; for instance, Adorn can be undone by projecting away extra columns, and ColumnEquate can be undone by re-duplicating the column data in question. It is unlikely that the inverses of semantic operators should be allowed as channel operators. The "inverse" of Adorn is, effectively, the relational Project operator, which is lossy and not invertible on its own.

There is another major difference that separates the semantic from the transformation operators. The transformation operators are intended to be invisible to the users of both the user interface and the query interface. Said another way, if two applications have the same user interface but different transformation operators in their channels, the user would not know the difference. The semantic operators, on the other hand, represent actions that create data that may be of interest to anyone constructing queries. Table 5 illustrates the beneficial

Table 5
Semantic operators, their parameters, restrictions, and effects on the query interface

Operator	Input parameters	Query interface effect
Adorn	T: The table to adorn C: The name of the new column f: A function with zero inputs and one output, representing the value of an environment variable b: A boolean value that, if true, allows updates on the value in column C only when a row is created	Adds a new element on the query interface for the related form(s) that lets the user query using elements in the new column
Audit	T: The table to which it will add temporal features C_b: The name of a column, representing the "begin" time C_e: The name of a column, representing the "end" time	Adds two new elements to the query interface for the related form(s) that let the user query using begin and end timestamps for tuples
ColumnEquate	T_1: The name of a table C_1: The name of a column in table T_1 T_2: The name of a table such that, if g_n is the g-tree entity node that generated T_n, one of the following relationships is true: g_2 is a descendant of g_1 g_2 and g_1 are both descendants of an entity node g_0 such that the path from g_1 to g_0 to g_2 traverses no *Multiple-launch* edges g_2 has a descendant attribute node a with no entity nodes on the path from g_2 to a and $Domain(a) = EntitySet(g_1)$ C_2: The name of a column in table T_2, which will be deleted	Let the user see, for any attribute in the interface, all equivalent attributes
TableAssert	T_1: The name of a table T_2: The name of a table whose objects are a subset of those in T_1	Let the user see, for any form in the interface, all subset/superset forms
CheckConstraint	f: The name of a function with boolean output **Cs**: The set of columns to be used as input to f	Warn the user if a query will necessarily return no data because of the constraint

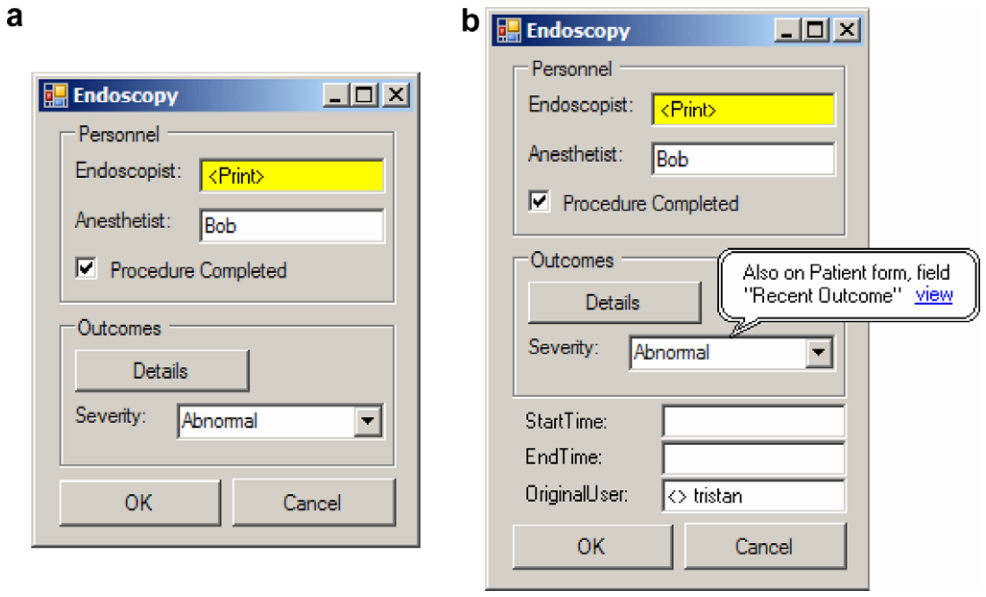


Fig. 10. A query interface for one form with no semantic operators (a), and the same interface after adding Adorn, Audit, and ColumnEquate operators (b).

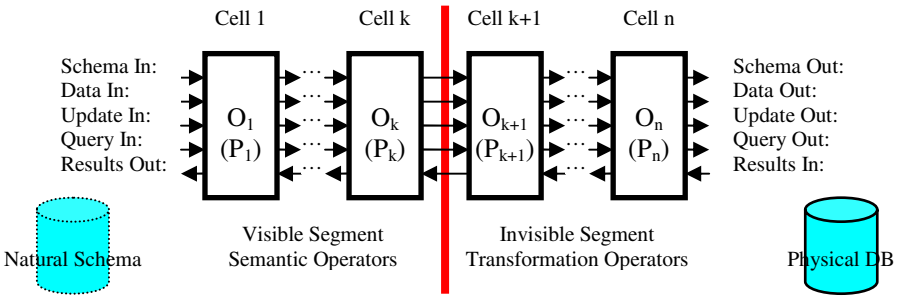


Fig. 11. The version of the channel as implemented in the current prototype of Guava.

effect each of the semantic operators can have on the query interface. Because the semantic operators are visible in the query interface, their effects should not be undone in the same way as the transformation operators – another reason why we do not worry about inverting the operators at this time.

Fig. 10 gives an example where the Endoscopy table in the natural schema has been affected by three semantic operators – an Audit operator, an Adorn operator adding the value of the user name that created the entry, and a ColumnEquate on the Severity attribute stating that it holds the same value as the Recent Outcome field in the Patient table. These three operators manifest themselves in the query interface, with the Audit and Adorn adding extra query fields and the ColumnEquate appearing as a help bubble that can be clicked to show the referenced form.

To allow for the semantic operators to affect the query interface while leaving the transformation operators hidden, we plan to split the channel into two segments: a *visible* segment that contains the semantic operators, and an *invisible* segment that contains the transformation operators. We place the visible segment closest to the natural schema to reduce the depth into the channel that one must search to find all such operators. The final version of the channel, with all five pathways and split into two segments, is shown in Fig. 11.

7. Related work

Our Pivot, Unpivot, HMerge, and HPartition operators are adapted from Lakshmanan, et al. [6]. Our work differs in its approach. Whereas SchemaSQL allows the user (i.e., the query writer) to express queries in the presence of schematic heterogeneity, Guava presents the user with a simple query language over a simple view of his or her data that mirrors a user interface. Guava then uses the transformations chosen during database design to transform the query and the query answer, as appropriate, including transformations that introduce schematic heterogeneity.

There are several approaches that model a user interface as a tree structure and view the associated data as an XML document, including XAML [16], XUL [18], and XForms [17]. These XML-based approaches are similar in spirit to Guava but they are limited to describing a single form at a time; there is no automated support for describing the relationship among forms other than by using a programming language.

The Guava channel allows the database designer to describe how to transform a natural relational schema (that arises from the hierarchical *g*-tree) to other relational schemas. So Guava offers an approach different from the typical XML shredding approaches [4] to choosing a relational database schema to store XML data.

Rather than starting with the UI and generating the back end as Guava does, a Ruby on Rails scaffold [12] starts with the database and generates a UI that performs queries and updates. That is, Ruby on Rails makes it easy to start with what we call the natural schema and then generate a user interface. The Rails framework provides no support for schematic heterogeneity. One approach to application development would be to start with the natural schema and use Ruby on Rails (or a similar tool) to generate a user interface and then use Guava to generate the back-end.

Several projects have studied the relationship between forms and data. For instance, Rollinson and Roberts [11] describe how to represent the semantics of a forms-based interface in a conceptual modeling language. The applications they consider are limited to ones where the UI and the database are closely related, perhaps even where the UI is semi-automatically generated from the database.

The Natural Forms Query Language [5] allows users to write forms that serve as the interface to an underlying database, both for updates and for queries. The relationship between NFQL forms and the underlying database is established using name matching, vocabulary analysis and user-guided heuristics. With Guava, one can have far more difference between the structure of the forms and the database than NFQL; for instance, Guava can support a generic schema (using the unpivot operator) and a temporal database (using the audit operator). Also, Guava artifacts are generated from a UI implemented in various languages, allowing the developer the freedom to choose the UI implementation language.

In general, our Guava framework offers one approach to information transformation and is thus related to some aspects of the decades of work in information integration. We comment on a few specific efforts here. Miller et al. [9] have considered ways to help the user identify mappings between data sources in the presence of schematic heterogeneity, e.g., by using data in various ways. In Guava, we are working in the context of a single database and we ask the database designer to generate the mappings by describing a channel. If two or more database systems had been designed using Guava and the user were able to indicate correspondences between two user interfaces, then perhaps we could exploit the definitions of the respective channels to construct the mapping between the two data sources.

Because the natural schema operates like a view over a physical database, Guava channels offer a new solution to the view update problem [3]. Existing solutions, including new approaches [2], usually focus on updates through views defined by SPJ queries. Guava does not have a channel operator that can do an arbitrary join. However, some of the Guava operators, including pivot and unpivot, are not covered by the existing view update literature.

Larson et al. [7] considered the problem of attribute and values equivalences extensively. We introduce the Apply operator to handle attribute transformations. The COIN project [13] focuses on describing attributes with context elements, such as units. Guava captures context information about each user interface control, such as the label that appears on the screen. We do not focus specifically on the kind of context elements considered in the COIN project, such as units for a value, but such context elements could be easily incorporated in Guava.

Guava offers an alternative to the classical extract-transform-load processing associated with a data warehouse [15]. In fact, Guava was motivated by the difficulties experienced by CORI analysts when trying to understand the data warehouse schema well enough to extract data for their queries. The contribution of Guava is it allows users to express queries against the natural schema, with context elements from the user interface, rather than against the data warehouse schema.

Finally, our work is not addressing the problem of automatic schema matching [10] but perhaps suggests the possibility of trying to match user interfaces directly and then rely on the corresponding channels to determine the mappings between schemas. Also, if we succeed in extending Guava to propagate user interface changes to the stored database schema, we will likely have a detailed description of how one schema (from an earlier version of the software) matches new versions.

8. Discussion

Guava hides the data transformation process from analysts, while simultaneously giving them powerful querying tools with an easy to use interface to the data. If the analyst is at all familiar with the user interface of the software, as the CORI analysts are, the query interface to the data will already be familiar. As a result, we expect that the time it takes to create and execute studies over data from a Guava application to be greatly reduced. By eliminating the developer from the study process, analysts can be sure that the data they see in a query result is exactly the data that was originally entered. We expect that this assurance, combined with being able to see the context of the data as it was entered, will translate into analysts being able to provide study results that more closely match the necessary semantic and quality requirements of the studies in less time, which could then translate into results such as better patient care.

In addition to the time saved due to not being involved in the analysts' querying processes, there are several additional ways that Guava reduces the burden on software developers. Some products, such as Ruby on Rails [12], save development time by creating an in-memory data access layer and semi-automatically creating links to the underlying database. Guava goes two steps further, first by automatically generating the data

access layer (the natural schema) and second by giving much more flexibility in the relationship between that layer and the database. That flexibility comes in the form of pivots, application of functions, and semantic and temporal features offered by the operators. Thus Guava provides more of the plumbing that is necessary to create a data-centric application, and allows for more options for constructing the physical database.

9. Conclusions and future work

This paper defines the initial, formal framework for supporting Guava where we use the user interface, directly, as a conceptual model. The query interface, automatically created from the specification of the user interface, allows a user to create queries in a familiar environment similar to that of the original application. Queries from the interface are translated into a subset of select-project-join relational algebra. The database operators, instantiated in a channel, precisely describe the transformations that take place between the natural schema and the desired, stored database schema. Using the channel, Guava allows users of both the original user interface and the query interface to interact with data without ever needing to know the structure of the physical data, providing logical data independence.

Our Guava implementation extends a subset of the forms controls in the Microsoft Visual Studio development environment to automatically generate the *g*-tree and corresponding natural schema. We are currently implementing the operators shown in Tables 3 and 5 in the channel. We are considering additional operators as necessary. We are also currently implementing a graphical channel builder tool.

We have defined the Guava query language, and are developing query optimization strategies. We are looking into ways that we can extend the expressive power of the query language to include arbitrary joins and set operators. We plan to develop a user-friendly interface that allows the user to express queries against a visual display that looks like the original user interface.

Although our work was prompted by the need to describe the relationship between the user interface and the stored, generic database typically used by forms-based reporting tools, we believe that the database operators introduced here can be used more generally. As one part of that work, we will define the conditions under which a channel is valid for a particular schema and instance. We will formally define the operators defined in Table 5 as part of that work as well. Finally, we will consider how to modify the Guava framework to accommodate modifications to the UI. We hope to propagate the desired changes to the *g*-tree and its natural schema through the channel to the physical database.

Acknowledgements

This work is supported in part by Collins Medical Trust, by DHHS NIH National Institute of Diabetes Digestive and Kidney Diseases No. 5-R33-DK061778-03 awarded to Oregon Health & Science University (OHSU), and by NSF grant No. 0534762.

References

[1] Clinical Outcomes Research Initiative. <http://www.cori.org/>. Last accessed on April 10, 2006.
[2] A. Bohannon, B.C. Pierce, J.A. Vaughan, Relational lenses: a language for updatable views, in: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '06), Chicago, IL, USA, 2006, pp. 338–347.
[3] U. Dayal, P. Bernstein, On the correct translation of update operations on relational views, ACM Transactions on Database Systems 8 (3) (1982) 381–416, September.
[4] F. Du, S. Amir-Yahia, J. Freire, A comprehensive solution to the XML-to-relational mapping problem, in: Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management, Washington DC, November 12–13, 2004, pp. 31–38.
[5] D.W. Embley, NFQL: the natural forms query language, ACM Transactions on Database Systems 14 (2) (1989) 168–211, June.
[6] L.V.S. Lakshmanan, F. Sadri, S.N. Subramanian, On efficiently implementing SchemaSQL on a SQL database system, in: Proceedings of the International Conference on Very Large Databases (VLDB 99), Edinburg, Scotland, September 1999, pp. 471–482.
[7] J.A. Larson, S.B. Navathe, R. Elmasri, A theory of attribute equivalence in databases with application to schema integration, IEEE Transactions on Software Engineering 15 (4) (1989) 449–463. April.

[8] R.J. Miller, Using schematically heterogeneous structures, in: Proceedings of ACM SIGMOD, Seattle, WA, June 1998, 27(2), pp. 189–200.
[9] R.J. Miller, M.A. Hernandez, L.M. Haas, L.-L. Yan, C.T.H. Ho, R. Fagin, L. Popa, The clio project: managing heterogeneity, SIGMOD Record 30 (1) (2001) 78–83.
[10] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, in: Proceedings of the 27th International Conferences on Very Large Databases, 10(4) 2001, pp. 334–350.
[11] S.R. Rollinson, S.A. Roberts, Formalizing the informational content of database user interfaces, in: Proceedings of the 17th International Conference on Conceptual Modeling (ER98), Singapore, November 16–19, 1998, pp. 65–77.
[12] Ruby on Rails. <http://www.rubyonrails.org/>. Last accessed on April 10, 2006.
[13] E. Sciore, M. Siegel, A. Rosenthal, Using semantic values to facilitate interoperability among heterogeneous information systems, ACM Transactions on Database Systems 19 (2) (1994) 254–290, June.
[14] J.F. Terwilliger, L.M.L. Delcambre, J. Logan, Context-sensitive data integration, in: Proceedings of the EDBT 2006 Workshop on Information Integration in Healthcare Applications (IIHA), Munich, Germany, March 26, 2006, pp. 20–31.
[15] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, S. Skiadopoulos, A generic and customizable framework for the design of ETL scenarios, Information Systems 30 (7) (2005) 492–525, November.
[16] XAML. <http://www.xaml.net/>. Last accessed on January 14, 2007.
[17] XForms. <http://www.w3.org/TR/xforms/>. Last accessed on January 14, 2007.
[18] XUL. <http://www.xulplanet.com/>. Last accessed on January 14, 2007.
[19] M.M. Zloof, QBE/OBE: A language for office and business automation, IEEE Computer 14 (5) (1981) 13–22.

James Terwilliger graduated with Phi Beta Kappa honor with a BA in Mathematics from Reed College, and earned an MS in Computer Science and Engineering from Oregon Health and Science University. He is currently a Ph.D. candidate in Computer Science at Portland State University. His research focuses on developing domain-specific query interfaces for non-technical users, as well as extending the capabilities of updatable views.

Lois M.L. Delcambre received her BS in Mathematics from the University of Louisiana, Lafayette (previously the University of Southwestern Louisiana), her MS in Mathematics from Clemson University, and her PhD in Computer Science from the University of Louisiana, Lafayette. She is currently Professor of Computer Science at Portland State University. Her research interests include information integration; data models, conceptual models, and knowledge representation models; digital libraries; and superimposed information – where the *superimposed* information includes *marks* (i.e., encapsulated addresses) to small-grained information in other (*base*) information sources. She co-developed this notion of superimposed information with David Maier, also a Professor of Computer Science at Portland State University.

Judith R. Logan received her BS in Chemistry from Stanford University, her MD from University of Southern California and an MS in Medical Informatics from Oregon Health & Science University. She is currently an Associate Professor of Medical Informatics and Clinical Epidemiology at Oregon Health & Science University where she teaches and directs development and technology for the Clinical Outcomes Research Initiative and the Informatics Shared Resource. Her research interests center on the uses of clinically collected health care data for research and quality improvement.