# Propagating Integrity Information among Interrelated Databases

*A. Rosenthal*
*The MITRE Corporation*
*Bedford, MA, USA (781)-271-7577. (781)-271-2352 (fax)*
*arnie@mitre.org*

*E. Sciore*
*Boston College (also at the MITRE Corporation)*
*Chestnut Hill, MA, USA (617)-552-3928*
*sciore@bc.edu*

**Abstract**

Data integrity policies often require that quality and integrity metadata be generated and communicated to potential users. However, in data warehouses, federations, and other multi-tier databases, administrators at different tiers use different schemas. Data at the upper tier is derived from the lower, so we consider the upper tier's tables to be views, derived by SQL-like expressions. Unfortunately, an assertion about some granule in the sources (a table, column, or cell) is often meaningless to view users, and vice versa. An understanding of the SQL view gives intuitive guidance for propagating such metadata, but not explicit semantics.

It appears feasible to create a system that drastically reduces the skill and labor required for propagating metadata and events between the tiers. We show many examples where, based on the view query and the metadata on the relevant sources, one can automatically generate useful propagation rules. Propagation downward from views to sources is also handled. Our approach is to automate the easy cases (which we expect to be quite common), and to assist on harder cases. Knowledge of specific metadata types or query operators can be supplied incrementally. If the view's query expression is difficult, one may compose the propagation rules from its constituent operators.

**Keywords**

**derived data, view, metadata, multi-tier, integrity**

This paper appears in the IFIP Conference on Integrity and Internal Control in Information Systems. Ed. S. Jajodia., Published by Chapman & Hall, 1998.

## 1 INTRODUCTION

A data integrity strategy is likely to involve large amounts of metadata (e.g., quality measures, constraints) plus operations and events (e.g., corrections, error messages). This information helps users employ the data correctly, and helps managers plan data quality improvements. Our work explores techniques for coordinating and propagating such information in a multi-tier database.

A *multi-tier database* is one that provides several different virtual or physical databases, each one derived from the one below. The phenomenon takes many forms. For example, a set of view tables can be used to insulate applications from stored tables that have been partitioned or denormalized, and which change as the workload changes. A federated database provides a virtual schema above multiple sources. A data warehouse gathers and transforms data and stores it in a separate server; this can be seen as computing a materialized view (subject to delays in propagating source updates). Web-oriented distributed systems often have tiers of objects derived from each other.

Because the data in the tiers of a multi-tier database are related, it is important that the tiers maintain full and consistent integrity information. However, integrity metadata specified at a tier is usually local to that tier and is not propagated to other tiers. The reason is that users and administrators lack the time, motivation, the skill in understanding data derivations (e.g., SQL), or the business knowledge needed to propagate each item of new or changed metadata to all interested parties. One cannot ask administrators to write a separate piece of code for each metadata type on each attribute, let alone for each row or cell that has metadata attached.

The goal of our research is to extend the ability of database systems to support tiers as views. Users at any tier should have the illusion that their database is single-tier, though perhaps having multiple administrators. In such a system, all relevant metadata would be propagated to each tier, and made accessible in terms of the schema at the tier. A simplified picture of the system appears below in Figure 1.

The metadata-propagation problem is comparable to the well-known problem of view update semantics (Keller, 1986) that has daunted database theorists. Consequently, we acknowledge that a fully automatic solution is not possible in all cases. Our intention is therefore to automate the easy cases, and provide automated assistance for the hard ones.

There are numerous types of metadata, many of which are domain-specific. Consequently, the kinds of metadata (and the options for how they will be treated) cannot be provided as a turnkey system. Instead, the system should permit extensions by tool vendors, customer organizations, and even business-oriented data administrators. To this end, we propose a framework that allows semantic choices to be supplied as small chunks of knowledge, rather than modifications to a query processor.

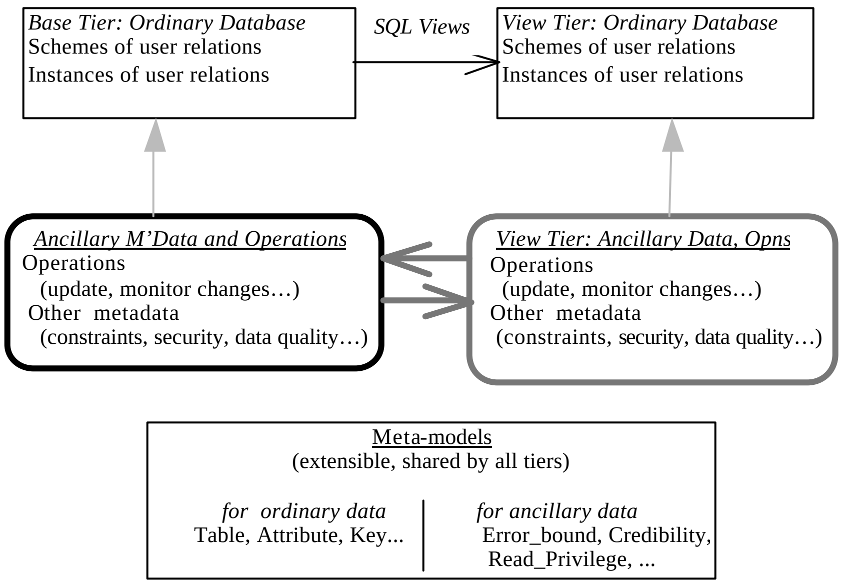
Base Tier: Ordinary Database
Schemes of user relations
Instances of user relations
SQL Views
View Tier: Ordinary Database
Schemes of user relations
Instances of user relations
Ancillary M'Data and Operations
Operations
(update, monitor changes…)
Other metadata
(constraints, security, data quality…)
View Tier: Ancillary Data, Opns
Operations
(update, monitor changes…)
Other metadata
(constraints, security, data quality…)
Meta-models
(extensible, shared by all tiers)
for ordinary data
Table, Attribute, Key...
for ancillary data
Error_bound, Credibility,
Read_Privilege, ...

Figure 1: System Overview

This paper is organized as follows. Section 2 discusses the need for integrity information in multi-tier databases, and how this information can be used effectively across tiers. Section 3 elaborates our proposed framework, and provides examples illustrating its components and its use. Section 4 provides conclusions, plans for future work, and some open issues in this area. Some preliminary results from this paper appeared in (Rosenthal, 1997).

## 2 DATA INTEGRITY AND METADATA

### 2.1 Data integrity requirements of large databases

Larger, more complex databases tend to require greater attention to data integrity and other controls, beyond the mechanisms used in simpler systems. Multi-tier databases, especially those that integrate data from many sources, are no exception. For example, the data warehouse literature reports that data integration and "data scrubbing" consume as much as 60-80% of the warehousing effort (Inman, 1996), (Robinson 1996).

Some of the reasons apply to any large system, regardless of architecture:

- Manual checking cannot handle the volume either of existing data or of new arrivals.
- Users' access to databases was traditionally mediated by applications, which often included integrity protections and limited the data returned (thereby also enhancing security). Now easy-to-use ad-hoc query interfaces make it feasible for many users to bypass this mediation.
- User bases are growing. A larger user base justifies greater expenditures.
- Finally, the data and tool sets are valuable, and should be made available to a wide span of users. Yet sensitive information will be withheld unless it can be protected. The larger scale of data and users (potentially thousands of data attributes and of users) makes security administration and enforcement serious problems.

Some other factors have been observed (by us and others) to apply especially to mechanisms that provide integrated views across multiple sources:

- Errors that went unnoticed when data was separate become painfully apparent when conflicting data is brought together. Improved data quality tests (e.g., consistency checks) may create a perception of decreased quality.
- Users at view tiers are often less intimately familiar with the underlying source data, and hence less able to compensate for faults.
- Warehouse users often use summary data (e.g., totals, averages), which may hide the underlying errors.
- Data that was appropriate for its original purpose may not suit new goals, e.g., matching against other data sources. The variations among the sources' attitudes, policies, and practices contribute to uneven quality.
- In an integrated database, the source that gathers certain data may not be a user of that data. So there is no natural internal feedback to ensure quality.

## 2.2 Uses of Propagated Metadata, Events, and Operations

Metadata is stored in the database, and can be queried using standard data query languages (such as SQL). The result of a query could include table references ("Which tables contain data supplied by Dow Jones?"), attribute references ("What attributes in the SUPPLIERS table have unreliable data?"), or data ("Which records in EMPLOYEE were last updated by user 'billgates'?"). This subsection discusses the ways in which users will want to access different kinds of integrity metadata.

### *Data Quality*

Data quality metadata might describe a granule's sources and processing history, its credibility, its error bounds (absolute or relative), and its availability (if connectivity is intermittent). Hundreds of other potentially useful types have been identified (Wand, 1996), so extensibility is essential.

End users and developers who work with view tables can use data quality metadata to select data to use (i.e., via query predicates that test quality information) and to interpret the data they do use. In both cases, they want descriptions in terms of their familiar views, not of source tables. Hence we need to propagate quality information upward.

Managers can use quality metadata to guide quality improvement, by comparing (in their native view) data quality with the business requirements for quality. When the existing quality falls short, the desired quality on view attributes needs to be propagated down to the source tables. That is, we want a "complaint" facility that propagates the complaints down to the correct source metadata. If other users should be made aware of the doubts (e.g., for scientific data or operational planning), shared complaints expressed against source data might then be propagated upward to other views.

## *Integrity Constraints*

Today's databases have many constraints defined on source tables. View users need to see them as constraints on view tables. Also, once a predicate is visible at both source and view tiers, it can be enforced at either place. Enforcing in clients permits faster notification of errors, and permits data entry when disconnected from the source databases.

A radical alternative is to say that constraint administration should be done mainly at the view level, not at the source level. After all, business domain experts are more knowledgeable about constraints than the technicians who design source tables. Each expert might define constraints for a portion of the database, expressed in terms of user views (e.g., the portion that they are primarily responsible for populating, e.g., transport or finance). Propagating these constraints down to source tables allows them to be enforced for all updates, not just updates through this view, and to use indexes maintained on the source tables. Also, constraints expressed on the source tables can propagate upward to other views.

For downward propagation, constraint predicates on view tables can be seen as queries, so query processors can re-express them on the source tables. However, this translation may not identify constructs (e.g., key constraints) that are meaningful to users or that have efficient enforcement techniques. Also, one needs to ask whether the constraint should apply to all updates, or only those received through this view.

Upward propagation is more complex. Ideally, one could devise a set of view constraints equivalent to those on the sources. In practice, while some source constraints will be expressible as constraints on data in the view tables, others will be irrelevant to updates provided through that view schema, and still others will be relevant but will require data not visible in the view schema (as discussed under the next item). A reasonable compromise would be to propagate source constraints upward to views wherever possible, and to indicate that further constraints are also enforced.

### *Error Messages*

Trouble ensues when a view user's update violates an integrity constraint that is enforced on source tier data. An error message should describe the violated constraint, but a message stated in foreign (source) terms may confuse and even anger view users. To the extent possible, the violation should be explained in terms of view tables, even if detected at the source.

This translation is easily automated for simple views, and when the constraint is expressible on information supplied with the user's update. However, some constraint violations involve data outside the user's view. This raises user interface issues; perhaps one should describe the error both in hybrid terms and purely in source terms. The system should also check whether the view user was authorized to read the additional relevant data from the source; if not, there are difficult tradeoffs between integrity and confidentiality (Jajodia, 1995).

## 3 A SYSTEM FRAMEWORK

The previous section demonstrated the value of propagating metadata between different tiers in a database system. This section describes a *framework* (that is, standards, services, and a repository of meta-metadata) that semi-automatically produces propagation rules, and enables new knowledge about propagation to be added incrementally and easily.

To understand the scope of this framework, suppose for a moment that a table at a view tier has just been defined in terms of some source tables. The view definition language (e.g. SQL) does not specify any metadata for the view. The framework therefore must do the following:

- Determine the types of metadata that should be in the view.
- Select "upward rules" (or provide the view creator with a choice of rules) that specify how view metadata values will be computed from source metadata, as well as rules that specify how the source metadata should change in response to changes in the view metadata (called "downward rules").
- Use these rules to answer user queries about the metadata, and to keep metadata values consistent between the two tiers.

Our intention is for the framework to be an aid to data administration. The rules chosen by the framework encode the semantics of the view, and thus ultimately require human assistance and verification. The data administrator should always be able to override suggested rules or add new rules. Although our discussion focuses on metadata, one also wants rules that propagate operations and events.

The following subsections flesh out this approach. Section 3.1 describes a 2-tier database for our running example. Section 3.2 examines how the framework determines the kinds of metadata in a view. Section 3.3 considers rules, and how they are used to compute metadata.

## 3.1 A Running Example

We shall illustrate the details of our framework using the following example. The source-tier schema contains information about aircraft, their positions, and their scheduled flights. Keys are underlined; FLIGHT.A_Id is a foreign key reference.

```
AIRCRAFT(A_Id, Type, Capacity)
POSITION(A_Id, LatLong, Speed_Mph, Height)
FLIGHT(F_Id, A_Id, StartAirfield, EndAirfield, FuelNeeded)
```

The view-tier schema contains three tables, defined as follows.

```
Define view POSITION_DE_AVION as
Select AvionNom=A_Id, JeSuisIci=LatLong, Vitesse_Kph=Speed_Mph*1.6
From POSITION

Define view AIRCR_FUEL as
Select A.*, F.F_Id, F.FuelNeeded
From AIRCRAFT A, FLIGHT F
Where A.A_Id = F.A_Id

Define view AIRFIELD_FUEL_NEEDS as
Select Airfield=StartAirfield, TotalFuel=SUM(FuelNeeded)
From FLIGHT
Group by StartAirfield
```

The view POSITION_DE_AVION renames POSITION attributes to French (or “franglais”) and converts Speed_Mph to kilometers. The view AIRCR_FUEL joins FLIGHT and AIRCRAFT, while AIRFIELD_FUEL_NEEDS computes the total amount of fuel needed for each airfield’s flights.

## 3.2 Granules and their Properties

### *Metadata on granules*

A *granule* is an identifiable subset of a database table to which metadata or methods can be attached, e.g., a table, column, row, cell value, or view. Each piece of metadata associated with a granule is given a name, called a *property*.

In our example, the granule POSITION might have a property Authorizations, describing the users authorized to access the table; the granule POSITION.Speed_Mph might in addition have the properties Credibility and AbsoluteErrorBound, denoting the fact that each data value in the column has the same credibility rating and error bound. Note that for the above examples, a value attached to a table or column granule describes each cell within it; other metadata types could describe properties more global to the granule. Values for specific granules override the wider-scope value.

We simplify our presentation by assuming that all property values are stored in one global *metadata table* that has three columns: Granule, Property, and Value. Thus one of the above properties might be represented in the metadata table as the row

(POSITION.Speed_Mph, Credibility, "Gary says 0.6")

In practice, this global table is likely to be a view that draws data from user data tables, CASE tool metadata tables, and system catalogs. However, a fuller treatment here would focus attention on irrelevant technical details (e.g., resolving inheritance and overriding, rules to propagate meta-metadata, and naming conventions for granules).

## *Derivation Queries and Derivation Trees*

To generate candidate metadata for a view granule, we first gather the metadata on relevant source granules. The task of identifying these granules is fairly straightforward. The basic idea is to exclude data and computation that are irrelevant to the view granule For a granule g of a view V, the *initial derivation query* (denoted idq(g,V)) is defined as follows:

- If g is an attribute A (i.e., column), use "Select V.A From V"
- If g is a cell, attribute A of row r, use "Select V.A From V where row_id = r"
- If g is the entire view, use "Select * from V"

Any query equivalent to idq(g,V) is called a *derivation query*, and denoted *dq(g,V)*. Query simplifications may be performed (e.g., replace V by its defining query, exploit integrity constraints). Algorithms developed for monitoring changes to views would seem to apply here, though we have not yet made specific connections. In particular, pushing projections down allows us to ignore metadata on attributes that are immediately projected away; selections allow us to ignore metadata on irrelevant cells.

For example, consider the views defined at the beginning of this section. Queries will be shown as trees, in relational algebra, and tree terminology will be used freely (e.g., calling the inputs of dq(g,V) the *leaves* of a *derivation tree*). The derivation tree for POSITION_DE_AVION.Vitesse_Kph appears in Figure 2a; the derivation tree for AIRCR_FUEL.A_Id appears in Figure 2b.

The derivation tree for a granule may be much simpler than the underlying view query. When AIRCR_FUEL is projected solely on FLIGHT attributes, the join with AIRCRAFT is irrelevant (since the foreign key constraint implies that each FLIGHT matches exactly one AIRCRAFT). Hence the derivation query for AIRCR_FUEL.FuelNeeded (see Figure 2c) contains only the node for FLIGHT.FuelNeeded. This sort of reasoning would require skill and care from a data administrator, but is easy to automate.

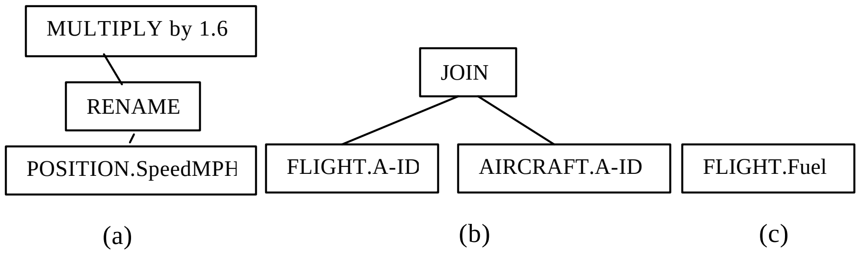

Figure 2: Example Derivation Trees

### *Suggested Granule Properties*

When the initial derivation query for a view granule g is simplified, what remains are the source granules necessary to compute g. Thus it is not unreasonable to assume that the properties of the source granules will "filter up" to g.

We express this intuition by defining the suggested set of properties for view granule g as {P| P is a property of a granule in a leaf of dq(g, V)}
The data administrator is free to omit some suggested properties and add others. In fact, the non-presence of a particular property in the view might be cause for a fruitful negotiation between the administrators of the two tiers.

## 3.3 Propagation Rules and Property Value Computation

Given a view granule g, the values of its properties will be calculated by assigning *propagation rules* (or just *rules*) to the operators in g's simplified derivation tree. Informally, each rule specifies a function that propagates metadata up (or down) the derivation tree. This section describes the structure and administration of these rules, as well as the mechanism for using rules to calculate property values of view granules.

### *Rules*

A rule has four components: a direction, a computation, a scope, and a strength. We discuss each component in turn, and then present examples.

The *direction* of a rule is either "upward" or "downward". An upward rule uses the values of the leaf granules to calculate a property value for the result. A downward rule uses the property value for a (single) view granule to calculate values for the leaf nodes. A rule specified as "both" is shorthand for two rules, one for each direction.

The *computation* of a rule is a function that determines how the output(s) will be calculated from the input(s).

The *scope* of a rule specifies when the rule is applicable to a granule. The scope specification makes it possible to define rules at the most general possible level, so as to promote sharing and better abstraction. In our scope, we can specify the tables, attributes, properties, and the atomic query operators to which the rule applies.

The *strength* of a rule specifies the extent to which the system should automatically use it. We currently have three possible values. A *definitive rule* is to be applied automatically, without user interaction. A *default rule* is to be preferred by the system, but is subject to user verification. And a *candidate rule* is one of perhaps several equal possibilities to be presented to the user. More sophistication is clearly possible (e.g., overriding, removing candidates, dependence on user, type-checking, etc.). However, we chose to leave such enhancements until we have better experience with the simpler scopes.

## *Examples of Rules*

The most generally applicable metadata-derivation rule is "do nothing" – that is, to pass the property value unchanged up (or down) the derivation query. This rule seems always applicable to the RENAME operator, and often applicable to MULTIPLY. (The property Credibility can pass through MULTIPLY unchanged, but AbsoluteErrorBound must scale proportionately.) The following rules capture these observations:

| | | |
|---|---|---|
| R1 | Direction: | both |
| | Computation: | output = input |
| | Scope: | operation=RENAME, |
| | | tables=ALL, atts=ALL, properties=ALL |
| | Strength: | definitive |
| R2 | Direction: | both |
| | Computation: | output = input |
| | Scope: | operation=MULTIPLY( c ) |
| | | Tables=ALL, atts=ALL, properties=Credibility |
| | Strength: | definitive |
| R3 | Direction: | upward |
| | Computation: | output = input * c |
| | Scope: | operation=MULTIPLY( c ) |
| | | Tables=ALL, atts=ALL, properties=AbsoluteErrorBound |
| | Strength: | definitive |
| R4 | Direction: | downward |
| | Computation: | output = input / c |
| | Scope: | operation=MULTIPLY( c ) |
| | | Tables=ALL, atts=ALL, properties=AbsoluteErrorBound |
| | Strength: | definitive |

There are many other rules of narrower scope. We would hope that vendors and even user organizations would incrementally add these rules to their systems. For example, for the SUM aggregation operator and the AbsoluteErrorBound property, a rule (with strength "default") might multiply the input by the number of values being aggregated.

Suppose we have probabilistic estimates of an attribute's correctness (here, defined as the probability of being exactly right) and availability (for fault tolerance, the probability of receiving a response from the server that stores the information). Then to calculate the correctness and availability properties, one might multiply the values from the input properties, with strength "candidate".

Other rules might perform more subtle analyses. We create a property type Pedigree to capture how each input to a granule's derivation affects the granule's value. Consider the view AIRCR_FUEL, obtained by joining AIRCRAFT and FLIGHT on foreign key A_ID. Because of the foreign key constraint, only FLIGHT.FuelNeeded influences AIRCR_FUEL.FuelNeeded. But the pedigree of AIRCR_FUEL.Capacity is more complex. AIRCRAFT.Capacity determines the value, but since an aircraft could have an arbitrary number of flights, both AIRCRAFT.A_Id and FLIGHT.F_Id influence which tuples are present, and the number of duplicates.

As a final example, when an operator combines two textual or Boolean fields, the result's Credibility might be set to the minimum (or product) of the input values (if purely numeric), or one might concatenate the textual discussions of credibility.

### *Invoking Propagation Rules*

Given property P of view granule g, its value is determined as follows. The derivation query dq(g, V) is calculated. For every operator in the tree, an applicable upward rule is chosen. The computations of the rules are then composed to compute the value of the root node, which becomes the value of P.

For example, consider the view granule POSITION_DE_AVION.Vitesse_Kph and its property Credibility. The derivation query tree for this granule was given in Figure 2a. We therefore need to choose an applicable rule for each of the two operators of the tree. Using the rules defined in Section 3.3, we see that R1 is the applicable rule for the RENAME operator, and R2 is the applicable rule for MULTIPLY. As both rules have the identity function as their computation, the result is that the value of Credibility is the same for this granule as for the granule POSITION.Speed_Mph.

Now consider the property AbsoluteErrorBound for the same view granule. We must choose rules for the same derivation tree applicable to this property. The applicable rules are R1 for RENAME, and R3 for MULTIPLY. The result is that the value for the property will be 1.6 times the value of AbsoluteErrorBound for POSITION.Speed_Mph.

In addition to selecting upward rules for each view property, downward rules can also be selected (either automatically, or with assistance from the view creator).

By doing so, the view creator links the metadata at both the source and the view, so that changes at one tier can be propagated to the other.

### *Administering Rules, within a Component Framework*

Managing the rule set includes creating and modifying rules, inspecting what rules apply, overriding or removing rules inherited from a wider scope, and selecting one of the candidate rules. The system should provide tools for performing all these tasks. Vendors, professional administrators, and power users need many of the same capabilities, so the tools should be part of the delivered system.

Rule choice may depend on the derivation query's logic, domain semantics, and organizational policy. Organizations could contribute domain-specific types and rules, and database administrators are able to add database-specific rules and override existing ones. Some specifications might be supplied when a new property or new derivation operator is defined. Others might be created when defining a view to serve a particular community or application. Simple tasks might be left to run-time users (e.g., confirming defaults, choosing among candidates).

It is impossible to provide appropriate rules for all properties, through all possible atomic query operators (both SQL and user-defined), for all organizations. A vendor of a propagation system could provide an initial set of useful rules. But as needs expand, both vendors and their customers will need to extend and customize the rule base. Thus, the system should be *componentized*, i.e., should allow simple, independent steps to extend the operators, properties, and rules.

An important aspect of our framework proposal is that it is a *component framework* for propagation rules, i.e., standards and services that enable separately-provided components to work together. The system framework would maintain a database of rules that is available to all tiers, and interfaces for inspecting, defining, modifying, and overriding rules. The framework also provides the facilities for rule invocation. Finally, to reduce semantic heterogeneity, a framework must define a set of fundamental properties (e.g., Credibility) and view-derivation operators (e.g., Select, Outerjoin), that all tiers would be encouraged to use.

## 4 DISCUSSION, SUMMARY AND FUTURE WORK

To manage integrity in a multi-tier database, we must propagate integrity metadata and events among the tiers. We have tried to illustrate several points:

- In multi-tier systems, it is essential to propagate ancillary metadata. For each metadata or event type, one may want propagation options that are customized for particular databases, tables, columns, cell values or other groupings. Since every attribute (and many other granules) may be associated with several pieces of ancillary information, automated assistance is essential.
- A framework can be constructed to help componentize propagation capabilities, enabling rules and knowledge to be supplied incrementally. The

framework would be employed at a variety of skill levels, e.g., to write new rules, to select appropriately from existing ones, or simply to execute a rule to see metadata from other tiers.

- While the general problem of "first class" views is notoriously hard, the goal of providing assistance is attainable. By offering multiple candidate rules, we help administrators handle cases where no single rule applies universally. A small collection of heuristics, plus knowledge of query operator semantics can handle many views.
- Propagation rules for complex queries can be composed from propagation rules of constituent operators, many of which will be simple. Propagating events may involve actions outside the database (e.g. "forward this request via email").

Our project (Managing Risk in the Data Warehouse) aims to provide the framework and simple components that handle some of the easy cases. More complex components (e.g., for complex derivation operators) would then be plugged in as researchers or vendors produced them. For example, research on data quality measures might lead to a component that was expert in transforms of precision metadata.

To illustrate the intended usage, desired capabilities, user roles, and technical feasibility, we have developed a demonstration vehicle (a series of screens, without real underlying code). The vehicle has helped us identify opportunities and difficulties. We also are using it to try to persuade tool vendors to add such capabilities to their products.

There are many challenges here for database researchers. There is no established propagation technology for most properties, operations and events. This is not surprising for little-studied issues like data quality, but it even applies to simple corrections. Potential research areas include *view updates after the source has changed* (e.g., for periodically refreshed materialized views), *bulk corrections* (i.e., translating SQL Update statements), *propagation options for additional query operators* (e.g., propagating error information through views (Kon, 1996)), *administration of expressions composed of multiple operators*, and *passing constraint information through views* (realizing that part of the constraint may not be expressible at the other tier, and few users can understand complex formulas).

Two broad challenges are critical to the success of this approach. First, vendors need to implement and perfect the framework specifications and services. Second, because multi-tier systems often span organizations, we need to borrow and use well-known ontologies for metadata and operation, both from consortia (World Wide Web consortium, Metadata Coalition) and from disciplinary bodies (e.g., Dublin Core, or geospatial metadata standards).

## 5 REFERENCES

Inmon, W. (1996) Building the Data Warehouse. John Wiley & Sons, New York.

Jajodia, S. (1995) Solutions to the polyinstantiation problem, in Information Security: An Integrated Collection of Essays (ed. M. Abrams et al.), IEEE Computer Society Press.

Keller, A. (1986) Choosing a View Update Translator by Dialog at View Definition Time. Very Large Data Base Conference, Kyoto, Japan.

Kon, H. (1996) Data Quality Management: Foundations in Error Measurement and Propagation. Ph.D. Thesis, MIT Sloan School of Management.

Robinson, T. (1996) It all starts with good, clean data. Software Magazine (supplement), October.

Rosenthal, A. and Dell, P. (1997) Propagating Integrity Information in Multi-Tiered Database Systems. Workshop on Information Quality, Cambridge, MA.

Wand, Y. and Wang, R. (1996) Anchoring Data Quality Dimensions in Ontological Foundations. Communications of the ACM, November.