Propagating Integrity Information in Multi-Tiered Database Systems

Arnon Rosenthal and Paul Dell
The MITRE Corporation, Bedford, MA
arnie@mitre.org, pdell@mitre.org

Abstract

Distributed heterogeneous databases involve information at several tiers: the source databases, perhaps intermediate results, and the derived database(s). Integrity information that originates at one tier will often be needed by users of the other tiers, but users and administrators may lack the documentation, skill, or time to reexpress it. Much research has been devoted to keeping source and derived databases consistent (i.e., view update and maintenance of materialized views). Analogous capabilities are needed for updates to metadata, and for events such as error messages and user corrections.

The difficulty is that while query transformation semantics are rigorous and well studied, the semantics for transforming this additional information are vague, requiring *ad hoc* customization. As a unifying metaphor, we propose that all interesting information items be appropriately *reflected* at all tiers, in a way that (perhaps informally) conforms to the effect of the data transformation processes in the multi-tier database. Such reflection is critical in managing data integrity efforts.

To specify and implement the desired reflection behaviors developers will need to implement many algorithms (hundreds or more), and administrators to select, for each attribute and annotation, which algorithm is appropriate for reflecting each chunk of information (potentially thousands). To keep all this organized and to factor out common features, we propose a framework for specifying reflection semantics, and for carrying out reflection requests. The framework is illustrated by showing how several kinds of quality metadata could be reflected across some derivation expressions.

1. Introduction

Metadata (e.g., quality annotations, constraints) and event-handling (e.g., of error messages and corrections) are essential to maintaining data quality. Researchers and vendors have produced quite a few techniques to manage this information in a single database. This work explores techniques for managing integrity-related metadata and events in a database that is organized into two or more tiers.

Multi-tiered architectures are being developed and utilized in a large number of organizations, both for entire distributed application systems and for distributed heterogeneous databases. A *multi-tier database* is one that provides several different virtual or physical databases, each one derived (approximately) from the one below. Typically, one tier represents the sources, while other tiers represent either intermediate results or different user communities (with a partial ordering). Integration of multiple sources over multiple tiers can yield very complex *multi-tier databases*. These include distributed databases that supply a virtual schema above multiple source schemas, and data warehouses where data is physically gathered, transformed, and stored in a separate server.

Figures 1a,b illustrate simple two-tier database systems, showing one or two databases at the source tier, a derivation step, and a derived database.

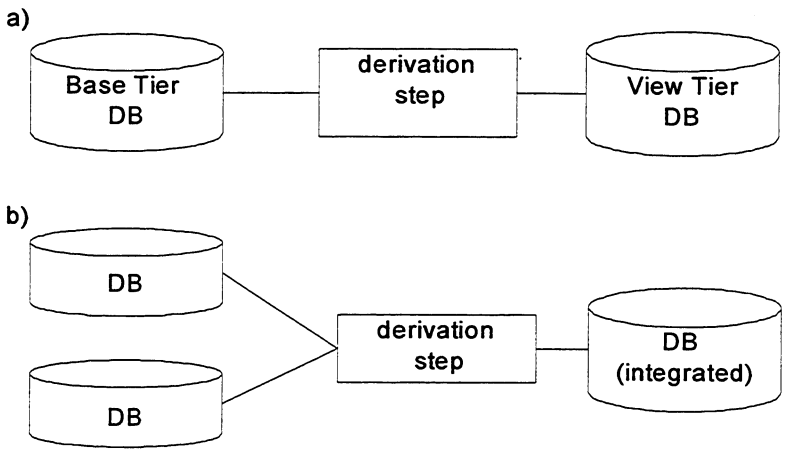


Figure 1a Simple 2-tiered database system with 1 source and 1 destination DB
Figure 1b 2-tiered database system with 2 sources and 1 integrated DB

Currently multiple tiered architectures do not support cross tier operations well. Instead the burden of mapping annotation data is left to the data administrators, and the burden of associating schema representations is left to the user. For example, suppose a view tier user complains "the accuracy of column Emp.TotalEarnings is terrible". This is information that the sources ought to hear, but they ought to hear it in terms of their own schema and quality measures, not in terms of some end user's view. Currently this inter-level schema mapping would be left to the user. We have two goals:

- *Automate the reflection* of most annotations (specifically integrity annotations) in a multi-tiered system. We expect that derivations of many attributes will be so simple that reflection can be automated.
- *Design a framework that componentizes* the metadata and services that customize and implement this automation. In particular, incremental extensions and changes should be supported.

To introduce this topic, section 1.1 explores the data integrity needs in the multi-tier database system. Section 1.2 discusses the notion of reflection and the types of information to be reflected. Section 2 introduces the framework concepts, vocabulary, and basic components. Section 3, provides some examples of reflection. The final section provides conclusions, plans for future work, and some open issues in this area.

1.1. The Requirement for Data Integrity

Larger, more complex databases tend to require greater attention to data integrity (and also security), beyond the mechanisms used in simpler systems. Multi-tier databases, especially those that integrate data from many sources, tend to fit this generalization. For example, the data warehouse literature often reports that data integration and "data scrubbing" is one of the major tasks consuming as much as 60-80% of the warehousing effort.[Inm96, Rob96]

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 (especially for updates) and limited the data returned (thereby 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.
- 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.

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. These improved data quality tests (consistency checks) may decrease the perceived quality.
- View tier end users are often less intimately familiar with the underlying source data, and hence less able to compensate for faults.
- View tier end users generally use summary data, which may hide the underlying errors.
- When data quality is uneven, it is difficult for users to use the information appropriately. 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.

1.2. The Need to Reflect Metadata and Events

Multi-tier databases are defined by derivation processes, both exact (e.g., database views) and approximate (due to time delays and incommensurable representations). The integrated tier is often not quite a view; instead, we say that the tiers *reflect* each other.

It is not only the ordinary data values that need to be reflected. Where integrity is an issue, a variety of *annotations* will be added. Some annotations hold data quality measures, both quantitative (accuracy, precision) and qualitative (identification of sources). Constraint predicates, error messages, and user corrections also need to be reflected across the tiers.

The annotations take several structural and storage forms. Some annotations may be persistent metadata (e.g., access permissions, constraint predicates, data quality measures), while others might be transient events that disappear after processing (e.g., urgent data corrections, queries with quality of service guarantees). Annotations may be attached to different sorts of data *granules*, such as tables, rows, attributes, individual cells, or arbitrary views (though some of these may be expensive to support). The derivation of a data granule determines (or at least strongly influences) what is an appropriate way to reflect metadata and event annotations on it.

2. System Concepts

This section describes the system model and elements of the solution. We first introduce the major components of the model. Some properties that the solution must have are then discussed. Finally we present the concepts and mechanisms used in organizing the solution.

2.1. System model

A multi-tiered database system is described in terms of databases, derivation function(s), data granules, and annotation(s) on data granules. These elements are shown in Figure 2, and defined subsequently. Each definition is followed by bulleted examples. To simplify the vocabulary, we assume there are only two tiers.

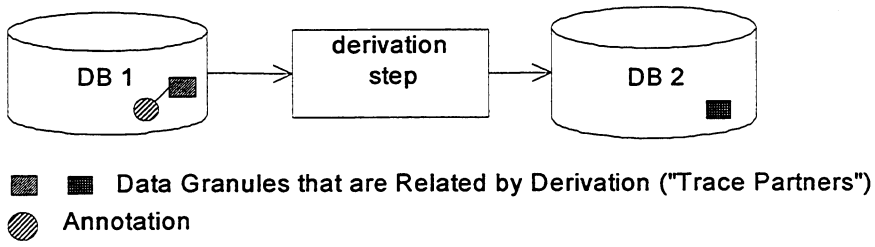


Figure 2 Basic Model for Multi-tier database system

Data granule: A chunk of data (e.g., table, attribute, row, cell).

- *EMP.Salary* - the salary *column* of the employee table
- *@emp123.Salary* - the *cell* holding the salary for the particular employee @emp123

Many familiar kinds of metadata can be treated as annotations on data granules.

Annotation: An annotation is additional information attached to a data granule, which is called the annotation's *root*. The annotation is typically used to help understand the granule's meaning and significance, or to describe an event that should be processed. Examples include credibility annotations, sensitivity annotations, integrity constraint predicates, and correction events. Typically, the functions that derive the view tier do not derive annotations. Annotations may be treated as data granules, and the dot notation for attributes extends straightforwardly.

- *EMP.Salary.*Credibility is an annotation on column EMP.Salary.

Metadata annotations are stored; event annotations are processed and discarded. Examples in each category include:

- Metadata: precision, accuracy, sensitivity, integrity_predicate, info_provider, credibility.
- Events: updates; corrections (i.e., updates with additional information to govern the behavior), error messages and warnings.

Annotations will typically be physically stored, so we assume that each tier has an annotation database that can store them. Often, to avoid interference with existing systems, the annotation database will be separated from the underlying data, i.e., kept in different tables or even different DBMSs. Users of the annotations are, of course, to be shielded from such distinctions.

Trace(Data granule, derivation, direction): This function traces through the derivation, to find related granules at the other tier. That is, tracing upward from a source granule g_s identifies view granules derived from it, via the indicated derivation. Similarly, tracing downward from view granule g_v identifies granules used in the computation that derives g_v. These are called *trace-partners* (for the indicated derivation and direction). We anticipate that the reflection framework vendor would deliver implementations of Trace for common derivation operators (e.g., relational algebra, attribute transformations).

Trace extends to cover annotations. That is, the trace of an annotation granule x.A is obtained by tracing the root granule x and then applying the qualifier ".A". That is, trace(x.A, Q, aDirection) = {y.*A* | y is in trace(x, Q, aDirection)}

2.2. Requirements that Guide the Framework Design

The goal of the framework is to allow the tasks of implementing and administering reflection to be componentized. That is, the framework should allow reflection to be customized, on whatever size data granule is appropriate. At the same time, each customization should be easy to specify and change. Pieces that can be applied repeatedly should be reused, to reduce implementation and administration work. As much as possible, tasks should be automated; for example, the system should semi-automatically compose the reflection semantics for primitive derivation operators to generate reflection semantics for a derivation expression. Finally, the degree to which existing databases manage the annotation information and reflection logic should be adjustable; this could be achieved by managing annotations separate from the underlying database thus minimizing the impact on existing systems.

2.3. Design Concepts

Our strategy for implementing the framework is to emphasize regularity, and to place each capability at the most general possible level. The concepts that implement this strategy are described in this section. For each concept, we give a definition and (bulleted) illustrations. We also briefly describe the role of the concept in the framework. Later sections bring the concepts together to illustrate how reflection can be specified and executed.

Annotation type: A general sort of annotation, which may be attached to different attributes in the database. (An annotation is then an instance of an annotation type, attached to a data granule and given a value.)

For each annotation type, we anticipate that the definer would provide one or several alternative semantics (i.e., functions) for each derivation operator and direction of reflection. For example, the Accuracy annotation type might define several functions for passing Accuracy through Total; administrators would choose among these options for DEPT.TotalSalary.Accuracy and TASK.Hours_spent.Accuracy. Each candidate is called a *reflection option.*

The framework can generate some reflection options automatically, independent of annotation type. For reflection upward from base to view, it might generate "move the annotation to every result attribute to which the source annotation's root contributes". And in any situation, given a list of options and a caption for each, it can generate the option "Ask the run-time user which of the {caption set} choices they want".

Reflectable Object: An annotation on a specific data granule, identified by the annotation type and the granule (instance). The system will attempt to reflect the value for that annotation.

- [*EMP.Salary*.Credibility = 0.7] - all values in the EMP.Salary column have a credibility of 0.7
- [*@task123.Budget*.Precision = 0.98] - the budget for Task123 is known with high precision

Decision Granule (DG): A set of values (annotation type, data granule set, derivation type, direction) for which a particular reflection function is chosen from the reflection options provided by the annotation type. Some or all of the values may be null. Members of that granule set (e.g., cells in a column) see the chosen reflection function.

- *TASK.Budget*.Precision uses reflection function f234 to reflect precision annotations on Budget, for all rows in the TASK table (note: derivation type and direction values are null)

The administrator of a DG will need an interface for choosing among the options, and perhaps for customizing them (e.g., by parameter substitution). That interface can be generated automatically, using captions and perhaps other information obtained from the annotation type. To simplify our examples and proposed implementation, we initially assume that annotations are on columns or individual cells, and decision granules are columns (e.g., *EMP.Salary*.Credibility).

Reflection choice: A selection of one of the candidate options for reflecting a particular reflectable object.

3. The Framework at Work

The goal of this section is to give the reader an understanding of how reflection may work for a variety of reflection tasks, for several kinds of integrity-related information. We first introduce the basic information and execution flow used in the framework. The examples then show how the framework supports default reflection behavior, administrator supplied reflection options, and derivation of new reflection options from the system information.

When transforming the information between tiers, the reflection choice depends on the Decision Granule information i.e. (annotation type, root data granule, the derivation expression(s), direction). The reflection system's basic execution flow is:

1. The framework is informed that a reflectable object needs to be reflected
2. The framework finds the decision granule for the reflectable object
3. The framework determines the trace partner for the given reflectable object
4. The framework retrieves the chosen reflection option and invokes it. This will involve:
    - invoked reflection code may perform some action (possibly null) at the current tier

- transform the annotation to a form understood by the other tier
- invoke action on other tier for each trace-partner

We now provide several examples to illustrate the scope of the problem and nature of the solution. Our hope is to convince the reader that the reflection framework can be a powerful tool for aiding or automating many annotation reflection tasks, and that the framework can make good use of existing system information in determining transformation rules for annotations. For simplicity, in the following examples the annotations are solely on column data granules.

We first describe the schemas, derivations, and annotations that will appear in the examples below

Explanation of attributes and notation:

_b, _v = Suffix used to designate attribute names at "Base" or "View" tiers

TASK, Name, Cost, and *T#* (Task number) attributes are self explanatory

Dur_hours, Dur_days = duration/length of task in hours or days

Crd: Credibility is a quality annotation

Prc: Precision is a quality annotation which is of type integer in the base tier and float in the view tier

Acc_b: Accuracy is a quality annotation with range 0-9 (base tier measure only)

Cor_v: Correctness is quality annotation with range 0-4 analogous to Accuracy (view tier measure only)

Derivation:

D1: Select all the attributes from the base table, changing the suffix; also, derive Dur_days = Dur_hours / 8

Schemas for Base and View Tables:

Base Tier Table: *TASK_b(T#_b, Name_b, Dur_hours, Cost_b)*

Base Tier Annotations: *Cost_b.Acc, Dur_hours.Crd and Dur_hours.Prc*

View Tier Table: *TASK_v(T#_v, Name_v, Dur_days, Cost_v)*

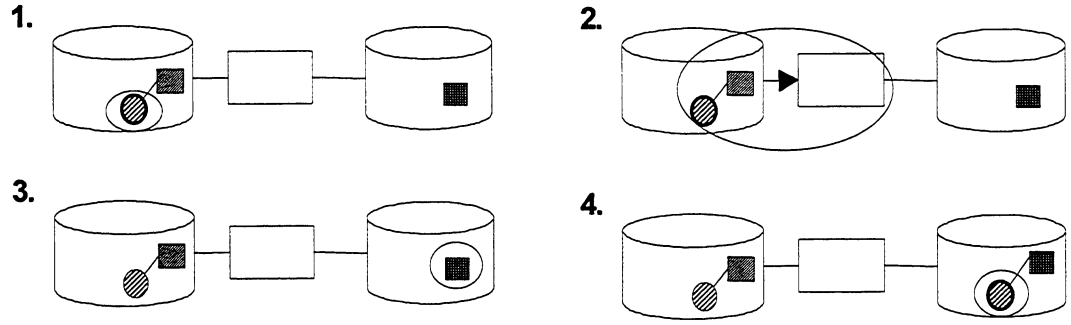
View Tier Annotations: *Cost_v.Cor_v*

3.1. Example: Reflecting Credibility from Base to View Tier

In this example, the annotation Crd_b on the *TASK.Duration* column is reflected to the view tier. The framework provides a default reflection option for each annotation type: Copy the annotation of that type to each trace partner of the root granule. This procedure is automated and does not need any intervention from the user or administrator. The processing of this reflection is described below. Each step is illustrated by the corresponding figure; the ellipse shows the part of the system that is active for that step.

Reflecting Crd_b to Crd_v will involve the following :

1. The framework is notified that *TASK_b.Dur_hours*.Crd_b needs to be reflected to the View tier
2. The framework determines the decision granule is (Credibility annotation type, *TASK_b.Dur_hours*, derivation = *D1*, direction = up)
3. The framework determines the trace partner. trace(*TASK_b.Dur_hours, D1*, up) = *TASK_v.Dur_days*
4. The framework finds and invokes the chosen (here the default) reflection option:
   a) Current Tier Action: null
   b) Translation: trace (*TASK_b.Dur_hours*, D1, up) = *TASK_v.Dur_days* and Crd_v = Crd_b
   c) Destination Tier Action: Store value *TASK_v.Dur_days*.Crd_v into the view tier's annotation database.



The framework determined the mapping between the data granule at the base with the granule at the view tier. Then it propagated the annotation on the base granule to be an annotation on the view tier granule. Without the framework's utilities, the user or administrator would have had to explicitly determined the transformation for each annotation type.

3.2. Example: Reflecting Precision Annotations when the Data Granule Changes

This example demonstrates that reflection options can be customized to handle reflection over different types of derivations. Given Prc_b metadata which annotates *TASK_b.Dur_hours* which represents the number of individual hours associated with the task. But in the view tier the corresponding column *Dur_days* represents the number of 8 hour work days associated with the task (e.g. 40 individual hours = 5 work days). To solve this mismatch a reflection option for (annotation type = Precision, data granule = *TASK_b.Dur_hours*, derivation type = (root value) / k , direction = up) is "Prc_v = Prc_b / k; then store as float in table Prc_v".

Reflecting *TASK.Dur_hours*.Prc to *TASK.Dur_days*.Prc will involve steps similar to 1-3 as given in Example #1, but step 4 will involve:

4. The framework finds and invokes the chosen reflection option:
   a) Current Tier Action: null
   b) Translation: trace (*TASK_b.Dur_hours, D1*, up) = *TASK_v.Dur_days* and Prc_v = (float) (Prc_b / 8)
   c) Destination Tier Action: Store value *TASK_v.Dur_days*.Prc into the view tier's annotation database.

The component supplier for Precision might supply a reflection rule that deals with conversions. That is, the derivation operator (derived_x = x / k) has the reflection rule (derived_x.Prc = x.Prc /k).

3.3. Example: Reflecting Accuracy Annotations from Base to View Tier

This example demonstrates reflecting annotations when the two tiers have different annotation representations. The reflection task is to reflect the Acc_b annotation on the *TASK_b.Cost_b* column from the Base tier to the View tier, but the View tier's accuracy annotation has a different value range. The administrator supplied reflection choice for Acc_b annotations on the *TASK_b.Cost_b* column is to halve and truncate the base tier value and store the new value as an integer in the view tier's column *Cost_v*.

In reflecting Cor_v from Acc_b step 4 will involve:

4. The framework finds and invokes the chosen reflection option:
   a) Current Tier Action: null
   b) Translation: trace (*TASK_b.Cost_b, D1*, up) = *TASK_v.Cost_v* and Cor_v = $\lfloor$(Acc_b /2)$\rfloor$
   c) Destination Tier Action: Store value *TASK_v.Cost_v*.Cor_v into the view tier's annotation database.

3.4. Example: Reflecting Accuracy Annotations from the View Tier Back to the Base Tier

Sometimes one of the tiers (here, the base tier) will not permit reflection to update its information directly. In this example, a base tier administrator intends to examine proposed updates (which must be phrased in terms of the base tier). Technically, this example introduces three new features: reflection downward , use of a non-default option on the Action step, and inverse reflection that has a set of possible options, from which one is chosen. (The forward reflection halved and truncated Acc_b to form Cor_v; a utility could generate the two downward options "*Acc_b* = (*Cor_v* *2)" or "*Acc_b* = (*Cor_v* *2+1)". The decision granule records the administrator's choice for

TASK_v.Cost_v.Cor_v, e.g., the second formula. So the initial execution of this reflection may consist of steps as in 1-3 above with step #4 being:

4. The framework finds and invokes the chosen reflection option:
    a) Current Tier Action: null
    b) Translation: trace (*TASK_v.Cost_v, D1*, down) = *TASK_b.Cost.b* and *Acc_b* = (*Cor_v* *2+1)
    c) Destination Tier Action: Notify administrator of base DB to the request "Insert value *TASK_b.Cost_b.Acc_b* into the database"

Derivations are generally not invertible. As a result, it is often ambiguous how an update on a view granule should be mapped to updates at the base. Not surprisingly, ambiguity in mapping changes to a root granule in the view can complicate the semantics of mapping annotations on that root.

4. Summary and Future Work

To manage data quality in a multi-tier database, it is necessary that quality annotations as well as base data values be reflected among the tiers. However, supporting reflection of annotations such as metadata and events complicates data administration. We have made several observations on providing reflection services:

1. There are a great number of metadata values and events to be reflected.
2. For each metadata or event type, one may want reflection options which are customized for particular databases, tables, columns, cell values or other groupings.
3. Reflection may involve actions outside the database (e.g. "...forward request via email...")
4. Supporting additional information and rules will require automated assistance and tools.

The reflection task for annotations is decomposed into small steps that directly correspond to operators used to derive the root data granule. 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 how to reflect precision and other quality information through views [Kon96] might lead to a component that was expert in transforms of precision metadata.

Researchers have a major role to play here. For many kinds of reflectable objects, there is no established technology. This is not surprising for little-studied issues like data quality, but it even applies to simple corrections. The first two research issues are extensions to the traditional "view update" problem [Kel86], motivated by the desire to accept corrections at the view tier.

- View updates with moving targets: If a correction is found and the source database changes its value in the interim, how should the correction be handled? Also will the correction even be able to trace the correction back to the original value? If a general solution is too difficult, at least the difficulties ought to be identified and isolated.

- Bulk corrections: How should one reflect a correction that is expressed as a query rather than as a set of tuples (e.g., Double all salaries for bearded employees in department G4F). Subproblems include: How should one view updates for bulk operation? Is it possible to determine an intensional operation to be supported at the other tier? How should one maintain a materialized view through a bulk operation?

- Specifying suitable reflection options for difficult operators: Each pair (annotation type, derivation operator) requires reflection logic for one or multiple options. Especially when an operation loses information (e.g., a Total) there may be no obvious semantics. Can one at least generating appropriate messages to the other tier, warning about problems in the proper attributes?

- Composition of reflection options and their selection: Reflection options will be provided and selected for a small set of basic operators; user queries are regarded as expressions composed of these operators. The reflection logic for such expressions will be composed of options selected for each component operator. Selecting the appropriate combination from the options for each operator may involve both algorithmic and human interface issues.

References

[Inm96] William Inmon. *Building the Data Warehouse*, John Wiley & Sons, (March 1996).

[Kel86] A. M. Keller. "Choosing a View Update Translator by Dialog at View Definition Time" *Proceedings 12th VLDB Conference, Kyoto, Japan,* (1986).

[Kon96] Henry Kon. "Data Quality Management: Foundations in Error Measurement and Propagation" *PhD Thesis, MIT Sloan School of Management*, (1996).

[Orf96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*, John Wiley, New York, (1996).

[Rob96] T. Robinson. "It all starts with good, clean data" *Software Magazine (supplement)*, (October 1996).

[Ros94] Arnon Rosentahal, and Leonard Seligman. "Data Integration in the Large: The Challenge of Reuse" *Proceedings of the 20th VLDB Conference Santiago, Chile*, (1994).

[Sil95] Avi Silberschatz, Mike Stonebraker, and Jeff Ullman editors. "Database Research: Achievements and Opportunities Into the 21st Century" *May, 1995 NSF Workshop, Stanford University* CS-TR-96-1563, (February 1996).

[Wan96] Yair Wand and Richard Wang. "Anchoring Data Quality Dimensions in Ontological Foundations" *Communications of the ACM* (November 1996).