First-Class Views:
A Key to User-Centered Computing

Arnon Rosenthal
The MITRE Corporation
arnie@mitre.org

Edward Sciore
Boston College and MITRE
sciore@bc.edu

ABSTRACT

Large database systems (e.g., federations, warehouses) are *multi-layer* – i.e., a combination of *base* databases and (virtual or physical) *view* databases[1]. Smaller systems use views for layers that hide detailed physical and conceptual structures. We argue that most databases would be more effective if they were more *user-centered* – i.e., if they allowed users, administrators, and application developers to work mostly within their native view. To do so, we need *first class views* – views that support most of the metadata and operations available on source tables.

First class views could also make multi-tier object architectures (based on objects in multiple tiers of servers) easier to build and maintain. The views modularize code for data services (e.g., query, security) and for coordinating changes with neighboring tiers. When data in each tier is derived declaratively, one can generate some of these methods semi-automatically.

Much of the functionality required to support first class views can be generated semi-automatically, if the derivations between layers are declarative (e.g., SQL, rather than Java). We present a framework where *propagation rules* can be defined, allowing the flexible and incremental specification of view semantics, even by non-programmers. Finally, we describe research areas opened up by this approach.

[1] The term "layer" was chosen for its naturalness in these familiar architectures. However, in the general case, derivation edges can form an arbitrary digraph.

1. WHAT NEEDS TO BE REMEDIED?

Views have enormous versatility, practicality, and importance [Rous98], but also significant limitations. Views give read access, but impose severe limits on other operations, such as updates and establishing change notification. Metadata defined on source tables (such as data quality, origin, and creation info) does not propagate up to views, and cannot be read or modified. Error messages from the system are presented to view users in terms of source tables. Schema changes to view tables (e.g., adding and deleting columns) are not allowed. Finally, there is no external mechanism for extending standard operations to wider classes of views (e.g., to define view update on views that include outerjoin or multiplication by a constant), or for defining new operations.

These limitations cause problems for both administrators and end-users. Administrators at a view layer are forced to perform their work in terms of tables belonging to other layers. Different layers may have large structural and vocabulary differences, making "schema-shifting" between them difficult and cumbersome. End-users at a view layer often cannot access their data directly, and must use applications to interface with the source layers. These applications can contain a lot of code, as they must manage the semantics of update operations, error handling, metadata propagation, etc. Layers can become "fat" with all of the code, which is costly to build and maintain.

In the remainder of this paper we try to give the reader a feeling for the tremendous potential of first-class views (Section 2), to outline a gradual approach to their implementation (Section 3), and show the numerous areas in which research is needed (Section 4).

2. HOW WOULD SYSTEMS LOOK, IF BUILT OVER FIRST CLASS VIEWS?

We say that a system supports *first class views* if:

- it offers view users (approximately) the same metadata and operations as source tables; and
- a view user who receives information from another layer has the pleasant illusion that the sender used the recipient's schema.[2]

In such systems, computing is more user-centered – users can work with tables natural to their viewpoint, regardless of the physical structure. Currently, views support this transparency for some data operations (e.g. queries and some updates), but rarely for other data operations, metadata operations, and events.

In this section we explore this idea further for two areas: multi-layer database systems (Section 2.1) and multi-tier distributed object architectures (Section 2.2). Although both areas involve metadata and application operations, section 2.1 focuses on the former and section 2.2 the latter.

2.1 Opportunity: Federations, Warehouses, Conceptual Views

Multi-layer databases, especially federations and warehouses, are gaining prominence. Figure 1 (next page) gives an impressionistic picture of military supply as a large, multi-layer federated database. Information in this database is collected from many sources, and transformed through several layers of organizations. Parts of the derivations will be simple (e.g., join on common identifier, units transforms, totals). Many attributes are passed upward with little change. But some derivations will also use complex functions that remain opaque to the DBMS, e.g., to normalize address representation or resolve conflicting values.

With current technology, to communicate information from sources to end users at the top, one needs to involve contractors at each layer. To enable this coordination, there is a strong effort to capture requirements in advance. However, the resulting systems tend to be too inflexible for our rapidly changing world. Fortunately, first class views have the potential to change this situation.

We look at three kinds of database usage: access to metadata, data operations, and schema administration.

Metadata: A source table's metadata covers many topics of interest to view users. Business users might define new metadata types, and provide values. (If we know how view metadata is derived, then an extension of view update theory guides handling of metadata updates). Important types of metadata include

- custodianship (ownership, points of contact)
- security specifications (e.g., access permissions, audit requirements)
- pedigree (e.g., what process produced it? who supplied the inputs? when?)
- data type
- quality (accuracy, completeness, precision)
- integrity constraint expressions
- descriptions of an integrity violation

One warehouse consultant estimated that his customers capture about eight such facts per attribute; more would be captured if the process were simpler. Metadata might also be captured at other granularities, i.e., table, row, cell, or view.

A user at a view layer must be able to manipulate the relevant metadata on the view. This metadata may have been specified explicitly at the view, or it may be implied by metadata specified at lower layers.

[2] Of course, the sender's information may refer to concepts outside the recipient's domain; in this case, the propagated result must reference the sender's schema. Negotiations between autonomous systems are discussed in Section 4.

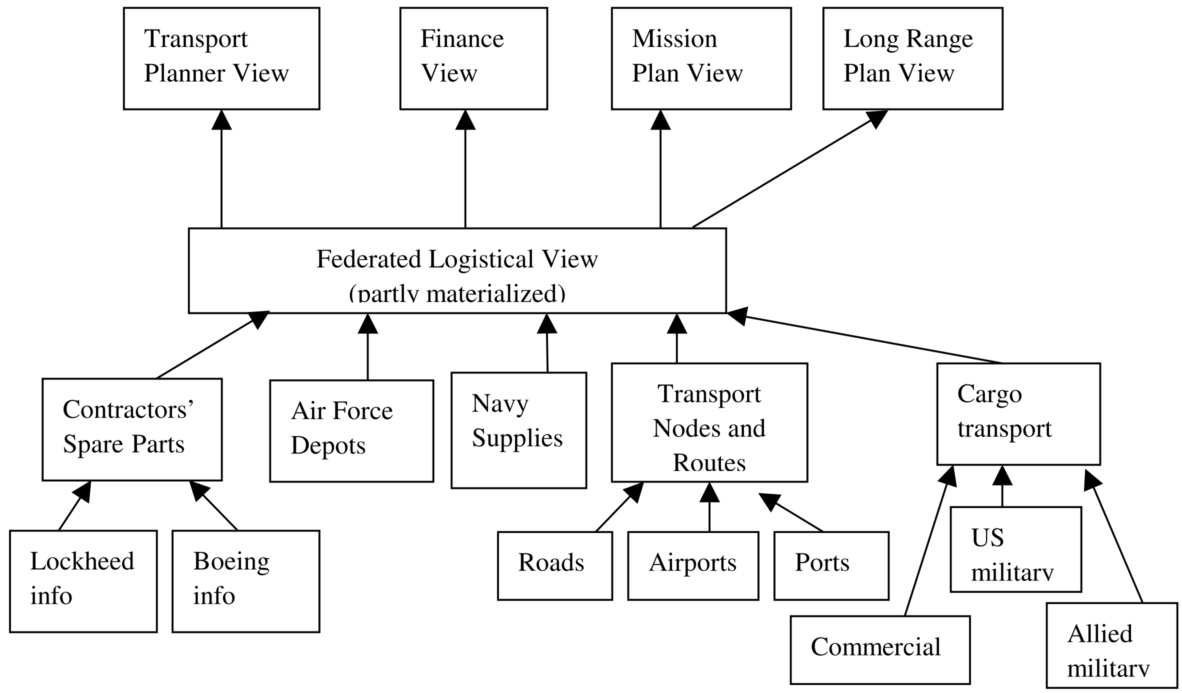

Figure 1: Military Supply (simplified)

For example, long-range planners may need to access the uncertainty metadata for contractors' delivery dates for F16 engines. Transport planners may need quality and access-control metadata for Albanian ports (How confident are we about Tirana's channel depth? Who may read the harbormaster's beeper number? Is it up to date?) In each case, the requested metadata must be translated by the system from its originating schema to the view's schema.

Operations: A user should be able to perform arbitrary operations (e.g. query, update, change notification requests) on view data. For example in Figure 1, transport planners may need to set up event notifications, so they are aware within 30 minutes of a major change to cargo capacity for any Airport, or to be informed when any contractor's supply of Tomahawks exceeds 20.

In federations, it is routine to translate view queries to source queries. For warehouses, queries normally execute against the warehouse. [Hel99] points out reasons to support querying through to original sources. For such queries, one needs declarative SQL view derivations. The same arguments apply to supporting other operations.

Today, when a federation user sees a need for an update or notification, she normally needs to express the update against a different schema. It would be simpler if she could issue these requests against their native schemas, perhaps extended by attributes needed for disambiguation. For example, federations often involve datatype and units conversions. To support these, the set of updatable views should be expanded, to allow unary operations whose inverses are available.

Schema Administration: Some schema administration tasks (e.g., constraint specification, adding and deleting columns) are best done by business experts, in terms of their native schema. Where possible, such administration should be insulated from details of sources' physical tables.

In addition, one can consider metadata derivations as an augmentation to views' defining queries. There is no uniform rule for metadata translation; sometimes it depends only on metadata type, sometimes also on the query operation, or the application semantics of a particular table. (For example, in some cases, "Select * from T where Date>1995" may bias average accuracy, and in other cases not. Administrators must have a way to manage the enormous number of meta-attribute translations easily.

Fortunately, derivations of view metadata can often be generated from general patterns (see Section 3). When a view is defined, the system generates metadata values automatically, wherever possible. Where the inferred values are unreliable, the system indicates the uncertainty or offers a menu of choices. In this way, business users see a relatively full set of metadata for the data in their view. Also, this view metadata is regenerated automatically upon changes to the source metadata or the view query.

These capabilities are extensible by user organizations, not just by DBMS vendors. For example, to handle unit conversions, power users can describe how "multiplication by a constant" affects access permissions (no change), error bounds (multiply absolute bounds by the constant), and view update (divide, as the inverse of multiply).

An added advantage of first class views is that they promote decentralized administration. Business experts can work more efficiently using their native views, as they need no longer be concerned with how their schema relates to others. Foreign concepts, unnatural organization, and irrelevant detail, all necessary at the lower layers, are hidden.

2.3 Opportunity: Enterprise Object Architectures

First class views are also relevant to enterprise architectures that consist of objects residing in tiers of servers, e.g., desktops, web servers, middleware, and databases. A tier can have several layers of objects (each implemented above the others), and each tier may implement over lower tiers.

Recent proposals for enterprise architectures use data management merely as a back-end store at the lowest tier. We advocate taking a greatly expanded role [Ros99a]. Data used only by one object at one tier requires no special coordination. But for shared data, our goal is: *Support the illusion that there is a giant shared database, of which each object's data constitutes a first class view.*

Currently, applications face problems similar to the multi-layer database systems described in Section 1. The programmer must think in terms of multiple schemas – her own layer, the supporting one, and ones to which she provides a customized interface.

As in multi-layer databases, little metadata is available outside the layer where it is provided. However, a more pressing problem is that method code is often "fat" – costly to build and change. A main cause is that generic data management tasks like change notification and attribute "set" methods must be explicitly coded (and maintained). The resulting code entwines business tasks and mapping tasks.

Today, a designer cannot afford to give *every* community a set of objects to work with. There is excessive cost in program maintenance (too much method code is needed for coordinating the data) and also in execution time. To see the latter, note that a method is implemented by requests to the next layer down, and it by requests to the next layer down, and so forth. This imposes a substantial performance penalty. For data manipulations over virtual data, the compiler composes the multiple layers of view definitions and determines an efficient strategy against the sources.

In a software system based on first class views, each layer sees the data it needs as attributes local to its objects, instead of as data to be obtained from or coordinated with other layers' objects. This locality simplifies methods at that layer. The task of coordinating between layers shifts from ordinary application developers to view definers.

The proposed future scenario is as follows: With the assistance of tools, modelers specify:

- *the query that derives the view's data* (declaratively, e.g., in SQL, so mappings other than query can be inferred).
- *process requirements from the application domain* (e.g., for persistence, for freshness of data values, for timely callbacks for interesting events, and for transaction consistency between layers).
- *functions that derive view metadata and operations* from the corresponding source information, and functions that infer source metadata from view metadata.

Next, implementation techniques (e.g., caching) are chosen by humans or automated wizards. At this point, the necessary code for coordinating across layers is generated, preferably automatically. ([Goy96] applied this elegant approach for GUI programming. We anticipate that it can scale up.)

With this regime, application methods are simpler. Their developers no longer require skill in data modeling and management. Caching and persistence strategies can be built largely from reusable generic code. Their implementation can change without affecting applications, and will eventually be chosen automatically based on usage profiles.

3. IMPLEMENTING FIRST-CLASS VIEWS

When modelers define an ordinary view, they provide only an SQL query that derives the contents of a view table. A first class view involves considerably more knowledge, i.e., a derivation mapping for each kind of metadata or propagatable operation, *on each attribute* (or other database granule), upward and downward, through each query operator [Ros99c]. The approach is feasible only with extensive code reuse and automated generation.

This section discusses an approach to:

- Semi-automatically generating mappings for metadata and common operations.
- Supporting incremental extensions, by administrators and tool vendors.

3.1. Why the Problem Is (Mostly) Tractable

The semantic difficulties of propagating view operations to source tables are well known, and we expect no silver bullet. But programmers can often determine candidate metadata for a view table, given metadata on the input data. And since some metadata types (e.g., owner, access control list) recur on almost every attribute or table, mapping functions can be reused extensively. Finally, even "impossible" general problems like view update can be handled by augmenting each view with stored procedures (or Instead triggers) that accept additional inputs to resolve ambiguities.

Thus, instead of seeking a complete solution, we reframe the problem as ***"Provide automated assistance to the humans who solve the individual cases."*** We would consider the automated assistance successful if we could provide a substantial fraction of the first class view functionality with little administrative effort. To do so, we hope to:

- *Handle the easy cases automatically.* Since many attributes in views are pulled with little or no change from a source table, many kinds of metadata require little mapping.
- *Disambiguate by point-and-click decisions, not by coding.* Tools already do this for updating a (vendor-determined) set of select-project-join views. It should be provided for other operations and metadata.
- *Focus human attention only on the difficult operations within view expressions.* For example, in Median(Project(Select(T))), only Median might need special handling.

3.2 Propagation Rules and Frameworks

To make views first class, we want a framework for specifying and applying broadly applicable *propagation rules*. To keep human effort low, the rules are general patterns that provide derivation mappings for many situations. This subsection gives a quick overview. The system has not been implemented, but an interface demonstration (pure html) is available at [Ros98]. Theory details can be found in [Ros99c].

For simplicity, we only consider here upward propagation (from source to view) of a meta-attribute, such as *ErrorBound* on *Port.Capacity* or *ErrorBound* in general. Similar techniques seem appropriate for metadata at other granularities (e.g., on tables and cells), for operations and events, and for downward propagation.

A *propagation rule* specifies a *scope,* a *strength,* and a *translation function.* The *scope* of a rule identifies the meta-attributes it applies to; it does so via a predicate that depends on the meta-attribute being mapped, the attributes involved, and the query operator. For example, a rule might apply to the *Credibility* meta-attribute on all attributes in tables Sue owns, for propagation through any invertable unary function.

The translation function takes value(s) for the sender's meta-attributes and produces a suggested value for the recipient's. The rule's *strength* describes its reliability; the framework uses it to determine priority of applicable rules and to estimate reliability of the value (e.g., to guide whether the result requires human confirmation).

By defining a taxonomy on each aspect of scope, we enable the framework to suggest ways an administrator might make a rule more general. For example, a rule that applied to *multiplyByConstant* might generalize to the parent category, {invertable operators}.

The rules are part of an extensible framework, which makes it possible to add rules and query operators incrementally. The framework provides a rule-execution environment; traces data flows between source and view granules (sometimes called lineage or pedigree); and defines "well known" types of metadata for use by all layers.

Tool vendors and power users can write rules for propagating through individual query operators, e.g., *Select*. The framework determines propagation through query expressions, by composing the functions for the individual operators. Sometimes a view can be derived from alternative sets of inputs, e.g., from sources or data marts. In such cases, the framework is responsible for inferring metadata values along each path, and combining (where possible).

4. RESEARCH ISSUES

The development of a facility for first-class views is a large, multi-faceted problem, involving both engineering issues and more formal questions.

4.1 Engineering Issues

The challenges here are to specify the framework, to provide good tools for administrators, and to coordinate with metadata managed by administration tools and DBMSs. One hopes to incrementally provide interfaces and wizards that simplify administrators' tasks, e.g., to identify overlaps and select rule strengths.

Much of our work assumes that metadata is (logically) attached to the various granules [Sci94]. There are many other situations where one needs to attach additional data to a granule, so a very general "annotation" facility would seem appropriate. Issues include crafting an interface for manipulating and displaying the attachment, and providing for a range of efficient implementations (with both eager and lazy evaluation of virtual fields).

Surveys of practice are also needed. We require a declarative description of the mapping from source to view. For a data warehouse's extract/scrub/integrate process, can such derivations be described using SQL plus embedded functions? Do such descriptions make it easy to determine the right propagation rule?

4.2 Formal Issues

The framework makes each bit of propagation knowledge valuable. This opens opportunities for small contributions (e.g., how to pass update permissions through outerjoin), perhaps from novice researchers and practitioners.

View queries that involve multiple operators require composing rules over the operators in an algebraic expression. But how does composition work, especially if multiple rules apply, or if user interaction is included? And how should we handle expressions where some operators have applicable rules, but others do not?

Better facilities for tracing lineage are needed. Current research aims largely at explaining data in the view, or at guiding a change notification strategy. Administration of first class views requires more. First, we want a uniform treatment that works at different granularities (e.g., column, cell, row). Second, to avoid prohibitive implementation costs, lineage-tracing should be a small change to algorithms that vendors already implement, e.g., query rewrite. Finally, when the lineage is complex, one needs to decompose in a way that matches administrators' concerns. (For example, for security officers, the lineage could indicate that a view depends on a join predicate's results, but not on actual joinfield values [Ros99b].)

There are special cases where rule inference can be formalized. For example, we have been working on a theory of metadata where all values represent lower bounds; this allows combining values inferred from different derivations [Ros99b]. One might also exploit invertability and monotonicity in propagating metadata. For example, suppose a view attribute's value is f(source attribute), and f^{-1} is known (e.g., constant multiplication and division). Since no information is lost, this mini-view can be added as a solvable case of both view update (using f^{-1}) and security (the view and source attributes are equally sensitive). Similarly, some metadata (e.g., maximum error) increases monotonically with the set of rows in a table. This knowledge admits conservative functions for propagating across Select or Union.

Research is needed on propagating constraints and error messages between layers. The theory of logical inference must be adapted in several ways: to handle differences in systems' abilities to enforce constraints; to express results in a form users will understand (e.g., "Emp# must be unique" rather than the corresponding predicate calculus); and to split constraint (or security) enforcement among a DBMS, middleware, and GUIs.

Security in multi-layer databases raises many good research issues: How do autonomous administrators at different layers negotiate and agree on appropriate access privileges, and ensure consistent enforcement? Should one protect information independent of location? A discussion of the issues appears in [Ros99b].

Where layers are administered independently, there is need for negotiation support. Upper layer administrators may wish to obtain certain data qualities and privileges for their users, i.e., to describe requests; they also describe enforcements performed at their layers. Data sources describe provisions, i.e., what they currently have, and perhaps could have. Propagation rules can translate requests and provisions so they are expressed in terms of a single schema, and hence are comparable. But there will be need to control proposals, counterproposals, hypothetical explorations, commands, and other forms of interaction.

Finally, multi-layer databases rely on replication. Multi-tier object systems rely on caching, often with imperfect consistency. The fundamental tasks of replication and caching products are the same – maintaining data consistency. A facility that unified these capabilities would be more generally useful, and avoid a great deal of redundant theory, specification language, and perhaps implementation.

5. SUMMARY

Our community has debated expanding its scope to additional data by "expanding the box" versus "getting out of the box". We believe there is another major opportunity that seems more easily grasped: to provide full database functionality to derived data that already resides in databases.

Views are one of the jewels in relational theory, and have wide applicability. This paper has argued that first-class views would make multi-layer databases more convenient for application builders. Even partial success would provide real help, so difficult cases can be solved later. Much of the time, the necessary specifications can be created semi-automatically.

We also described how such views simplify the methodology and implementation of multi-layer object architectures. Most software architects give DBMSs a limited role, providing just persistent storage at the bottom layer. Our proposed direction gives the database community a wider role in creating tomorrow's enterprise software systems.

In software, as in circuitry, the cost of active components may eventually be exceeded by the cost of maintaining the interconnections. First class views help slim and coordinate the layers. A suitable framework can greatly simplify the management of the inter-layer connections.

6. REFERENCES

[Goyal96] N. Goyal, C. Hoch, R. Krishnamurthy, B. Meckler, M. Suckow, "Is GUI Programming a Database Research Problem?". *Proc. ACM SIGMOD Conference, June 1996*.

[Hel99] J. Hellerstein, M. Stonebraker, R. Caccia, "Independent, Open Enterprise Data Integration", *IEEE Data Engineering Bulletin*, March, 1999.

[Kel86] A. Keller, "The Role of Semantics in Translating View Updates". *IEEE Computer* (19:1), January 1986, pp. 63-74.

[Ros98] A. Rosenthal and G. Gengo, "An HTML Demo of a Multi-Tier Administration Tool". www.cs.bc.edu/~sciore/papers/demo.zip

[Ros99a] A. Rosenthal, E. Hughes, "Coordinating All the Data: A Big Role for Data Management in Enterprise Architectures". www.mitre.org/technology/managing_risk/pubs/oomtier_doa.ps

[Ros99b] A. Rosenthal, V. Doshi E. Sciore, "Security Administration for Federations, Warehouses, and other Derived Data", IFIP 11.3 Working Conference on Database Security, 1999. www.cs.bc.edu/~sciore/papers/secadmin.doc

[Ros99c] A. Rosenthal, E. Sciore, "Metadata Propagation in Large, Multi-tier Database Systems" Submitted to ICDE 2000. www.cs.bc.edu/~sciore/papers/ metadataProp.doc

[Rous98] N. Roussopoulos, "Materialized Views and Data Warehouses". *SIGMOD Record,* March 1998, pp. 21-26.

[Sci94] E. Sciore, M. Siegel, and A. Rosenthal, "Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems". *ACM Transactions on Database Systems* (19:2), June 1994, pp. 254-290. file://rombutan.mit.edu/pub/papers/TODS94.ps

Acknowledgement: The authors thank Sandra Heiler, Frank Manola, and Scott Renner for their insightful suggestions.