A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches*

Richard Hull
Computer Science Department
University of Colorado
Boulder, CO 80309-0430
hull@cs.colorado.edu

Gang Zhou†
Computer Science Department
University of Colorado
Boulder, CO 80309-0430
gzhou@cs.colorado.edu

Abstract

This paper presents a framework for data integration currently under development in the Squirrel project. The framework is based on a special class of mediators, called *Squirrel integration mediators*. These mediators can support the traditional virtual and materialized approaches, and also hybrids of them.

In the Squirrel mediators, a relation in the integrated view can be supported as (a) fully materialized, (b) fully virtual, or (c) partially materialized (i.e., with some attributes materialized and other attributes virtual). In general, (partially) materialized relations of the integrated view are maintained by incremental updates from the source databases. Squirrel mediators provide two approaches for doing this: (1) materialize all needed auxiliary data, so that data sources do not have to be queried when processing the incremental updates; or (2) leave some or all of the auxiliary data virtual, and query selected source databases when processing incremental updates.

The paper presents formal notions of consistency and "freshness" for integrated views defined over multiple autonomous source databases. It is shown that Squirrel mediators satisfy these properties.

1 Introduction

The advent of the Information Superhighway has dramatically increased the need for efficient and flexible mechanisms to provide integrated views over multiple information sources. The traditional approach to this problem is to represent the view in a *virtual* fashion; queries against the view are decomposed and sent to the remote sources [SBG+81, DH84, LMR90, T+90, ADD+91, ACHK93]. A complementary approach has emerged recently, that is based on storing the view in *materialized* form [WHW89, ZHKF95, ZHK95, ZGHW95]. In that approach, queries can be answered without accessing the source databases, and the materialized view is typically maintained via propagation of incremental updates. Speaking broadly, the virtual approach may be better if the information sources are changing frequently, whereas the materialized approach may be better if the information sources change infrequently and very fast query response time is needed. The virtual and materialized approaches represent two ends of a vast spectrum of possibilities. This paper develops a general and flexible framework for supporting integrated views using a hybrid of these two approaches.

In the framework developed here, a relation in the integrated view can be supported as (a) fully materialized, (b) fully virtual, or (c) partially materialized (i.e., with some attributes materialized and other attributes virtual). In general, (partially) materialized relations of the integrated view are maintained by incremental updates from the source databases. Two approaches for doing this are provided in our framework: (1) "fully materialized support" [ZHKF95, ZHK95]: materialize all needed auxiliary data, so that data sources do not have to be queried when processing the incremental updates; and (2) leave some or all of the auxiliary data virtual, and query selected source databases when processing the incremental update. A view with fully materialized support has been termed "self-maintainable" in [GJM96].

The framework presented in this paper forms one aspect of the Squirrel project currently under way at the University of Colorado [ZHKF95, ZHK95]. The framework is based on a special class of mediators [Wie92], called *Squirrel integration mediators*. Squirrel is a tool that can be used to generate these mediators from high-level specifications. References [ZHKF95, ZHK95] describe a restricted form of Squirrel mediators, that support only fully materialized integrated views with full materialized support; they do not address the issue of virtual or hybrid integrated views.

*This research was supported in part by NSF grant IRI-931832, and ARPA grants BAA-92-1092 and 33825-RT-AAS.

†A student at the University of Southern California, in residence at the University of Colorado.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

The present paper substantially generalizes and extends the work of [ZHKF95, ZHK95] by developing a framework for supporting hybrid materialized/virtual integrated views. In particular, we describe a generalized architecture for Squirrel mediators that supports hybrid integrated views, and we present the key algorithms used by these mediators. The discussion here is presented in terms of the relational model, and the general spirit of our techniques can be applied in the context of the object-oriented database model.

The Squirrel mediators implement a synthesis of several technologies, including query processing against virtual integrated views [LMR90], algorithms for updating materialized views [BLT86, GMS93, GL95, ZGHW95], the use of the active paradigm to implement those algorithms [CW91, Cha94], and the use of "active modules" to provide light-weight, customizable activeness [Dal95, BDD+95].

A central construct used by Squirrel mediators to support integrated views is the notion of "View Decomposition Plan" (VDP). A VDP provides a systematic framework for synthesizing the above technologies in order to support materialized, virtual, and hybrid relations. In particular, this includes support for (i) retrieval from source databases of virtual data needed to answer queries, and (ii) incremental update of materialized data (including querying of source databases as necessary). VDPs provide a framework for optimizing support for integrated views in a manner reminiscent of query execution plans.

This paper presents formal notions of consistency and "freshness" for integrated views defined over multiple autonomous source databases. Consistency guarantees that the state of the view at any time corresponds to a family of states of the source databases (although not necessarily states that existed simultaneously). Freshness ensures that updates to source databases are reflected in the view within a bounded time. It is shown that if the source databases and network satisfy certain natural conditions, then Squirrel mediators satisfy the consistency and freshness properties.

Section 2 uses a simple example to illustrate different kinds of hybrid integrated views and how Squirrel mediators can support them. Section 3 presents the formal notions of consistency and freshness. Section 4 presents the architecture of Squirrel mediators. Section 5 describes VDPs. Section 6 describes the three central algorithms used by Squirrel mediators to support hybrid integrated views. Section 7 shows that Squirrel mediators satisfy the consistency and freshness properties. Some future research directions are offered in Section 8. Due to space limitations, many details are omitted in this paper; see [HZ96].

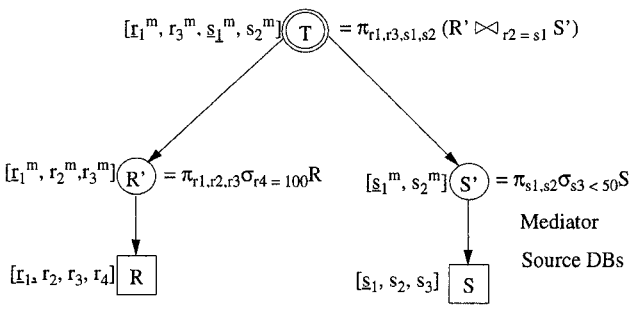

Figure 1: An annotated VDP for an integrated view $T = \pi_{r_1,r_3,s_1,s_2}(\sigma_{r_4=100}R \bowtie_{r_2=s_1} \sigma_{s_3<50}S)$

2 Motivating Examples and Intuitive Remarks

This section presents three related examples to give a progressive overview of several key aspects of Squirrel mediators. Example 2.1 illustrates how a Squirrel mediator maintains a materialized relation T that has fully materialized support, and introduces the central notion of annotated VDP. Example 2.2 modifies the first example by allowing some of the auxiliary supporting data to be virtual. Providing support for T as a hybrid (partially materialized) relation is described in Example 2.3.

Example 2.1: Let $R(\underline{r_1}, r_2, r_3, r_4)$ with key r_1 and $S(\underline{s_1}, s_2, s_3)$ with key s_1 be two relations from distinct databases. Suppose that the integrated view supported by a mediator has the single export relation $T = \pi_{r_1,r_3,s_1,s_2}(\sigma_{r_4=100}R \bowtie_{r_2=s_1} \sigma_{s_3<50}S)$. A VDP for T is shown in Figure 1. Each non-leaf node in the VDP corresponds to a relation maintained by the mediator, and each leaf node corresponds to a relation in a source database. The attributes of the relations are shown to the left of the nodes. The relationship between a node and its children nodes indicates that the relation of the parent node is derived directly from relations of the children nodes.

An attribute of a relation of a non-leaf node is annotated as either materialized or virtual. For example, the attribute notation $[r_1^m, r_3^m, s_1^m, s_2^m]$ for the node T indicates that all four attributes are materialized, i.e., the relation T is fully materialized. In fact, all the relations maintained by this mediator are fully materialized.

Supporting queries against relation T is trivial. The incremental maintenance of T is supported by the auxiliary relations R' and S' that are fully materialized. As will be discussed in Subsection 5.2, the rules responsible for propagating updates from R' and S' to T are:

rule #1: on changes to R', $\Delta T = \Delta R' \bowtie S'$;
rule #2: on changes to S', $\Delta T = R' \bowtie \Delta S'$;
where ΔQ denotes the net change to a relation Q. In this context, T can be maintained using incremental updates from the source databases and data local to the mediator, without querying the source databases. □

The next example modifies the previous example by selectively materializing the auxiliary data for T.

Example 2.2: Assume that updates to relation R are frequent, but updates to relation S are infrequent. To reduce the overhead of continually maintaining R' and to conserve space in the mediator, we change the annotation of R' to be $[r_1^v, r_2^v, r_3^v]$, where the superscript v stands for virtual. So R' is fully virtual. The annotations for S' and T are unchanged. Rule #1 introduced in Example 2.1 shows that in response to a change $\Delta R'$ to R', ΔT is computed solely from $\Delta R'$ and S'. So letting R' be virtual does not delay the propagation of updates against R, i.e., the majority of updates. In the rare case when updates to relation S occur, the mediator must incur the expense of sending queries to relation R to compute ΔT. □

The next example will not only have some auxiliary data virtual, but also keep some infrequently accessed attributes of the export relation T virtual. We describe how queries are answered, and how the values of virtual attributes are obtained when needed.

Example 2.3: Assume that queries against relation T mainly refer to attributes r_1 and s_1, i.e., to $\pi_{r_1,s_1}T$. For this example, we choose the annotation for the VDP to be: $T[r_1^m, r_3^v, s_1^m, s_2^v]$, $R'[r_1^v, r_2^v, r_3^v]$, $S'[s_1^v, s_2^v]$. The response time to the queries that only refer to r_1 and s_1 is not affected by the fact that r_3 and s_2 are virtual. With this annotation, answering queries involving attributes r_1 and s_1 is straightforward.

How does the mediator answer a query involving one or more virtual attributes, e.g., the query $q = \pi_{r_3,s_1}\sigma_{r_3<100}T$? The mediator constructs a temporary relation that is equivalent to the answer of q. For this example, the temporary relation will be $T_{tmp} = \pi_{r_3,s_1}\sigma_{r_3<100}T$. This can in turn be constructed from relations of the children nodes of T, namely R' and S', as: $T_{tmp} = \pi_{r_3,s_1}(\sigma_{r_3<100}R' \bowtie_{r_2=s_1} S')$
In this manner, the VDP serves as a guide concerning what temporary relations need to be materialized in order to answer a query against an export relation.

Sometimes, there are more efficient ways to construct T_{tmp} by using integrity constraints inherited from the source databases, but that is not the focus of this paper. □

The preceding examples showed how a single VDP can be used to support a variety of different combination of materialized and virtual support for an integrated view. VDPs are quite similar to query execution plans, in that both data structures represent a decomposition of one or more queries. VDPs are used to support queries against an integrated view, to hold materialized portions of the view, and to organize the incremental maintenance of those materialized portions. As a result, the VDP of a Squirrel mediator is relatively static, although one VDP may be migrated to another VDP as a result of optimization. The annotation of a VDP might also be modified. In contrast, query execution plans are typically developed on a query by query basis.

3 Formal Notions of Correctness

The kind of consistency that should be supported for a view depends on the context of the view definition. If source data and a (materialized or virtual) view is within a single database system, then the view can be designed to reflect the current database state. In the materialized case this is accomplished by combining within a single transaction both updates and update propagation to the view. Typically, a virtual or materialized view defined over remote databases will reflect *some* state of the sources, but not necessarily their current state.

This section develops formal notions of correctness for the context where an integrated view over multiple source databases is supported in a separate database system. We assume that the source databases are relatively autonomous, and do not assume that they participate in global transactions. As a result, there is generally no global state of the multiple autonomous databases. The formal definitions given below capture natural intuitions about consistency and "freshness" in this context. The integration mediators described in this paper satisfy these properties (see Section 7).

An *integration environment* consists of a sequence $\vec{DB} = \langle DB_1, \ldots, DB_n \rangle$ of source databases, a view definition ν that defines an integrated view of portions of the source databases, and a "database" V, which is intended to hold that view. The environment is presumed to include software that supports V in some manner. (The view may be either materialized or virtual, or a hybrid of these.)

We shall use the following notation.

- $\vec{t}$: a time vector in the form of $\langle t_1, ..., t_n \rangle$. We write $\vec{t} \leq \vec{t'}$, if $t_i \leq t'_i$ for each $i \in [1, n]$. Also, $\vec{t} < \vec{t'}$, if $\vec{t} \leq \vec{t'}$ and $\vec{t} \neq \vec{t'}$. If t is a time and $\vec{t'}$ a time vector, then $t \leq \vec{t'}$ means that $\langle t, \ldots, t \rangle \leq \vec{t'}$; $t < \vec{t'}$, etc., are defined analogously.

- $state(d, t)$: the state of d at time t, where d ranges over DB_i or V.

- $state(\vec{DB}, \vec{t})$: the state vector of databases $\vec{DB}$ at time $\vec{t}$, i.e., $\langle state(DB_1, t_1), \ldots, state(DB_n, t_n)\rangle$.
- $\nu(s)$: a view defined on a state (vector) s of source database(s).

We now define the notions of consistency and freshness for integration environments. In these definitions we let t_{view_init} denote the time at which the view is initialized.

Definition: [Consistency]
An integration environment is *consistent* after time t_{view_init} if there exists a "reflects" function $\vec{ref}$: $Time \rightarrow (Time)^n$ such that:

(a) [Validity] For each time $t \geq t_{view\_init}$,
$state(V, t) = \nu(state(\vec{DB}, \vec{ref}(t))$.
(Intuitively, this means that the state of the view at time t should correspond to some state vector of the source databases. We do not insist that the view corresponds to the set of source database states at a single time t', because updates from and accesses to the source databases will typically be asynchronous.)

(b) [Chronology] For each time $t \geq t_{view\_init}$ and each $i \in [1, n]$,
$t \geq \vec{ref}(t)_i$. (Intuitively, this insists that the state of the view at time t corresponds to the source databases at a times $\leq t$, i.e., the view does not "forecast the future".)

(c) [Order preserving] For each pair t_1, t_2 of times satisfying $t_{view\_init} \leq t_1 \leq t_2$,
$\vec{ref}(t_1) \leq \vec{ref}(t_2)$.
(Intuitively, this insists that successive states of the view correspond to successive (vectors of) states of the source databases.)

We now give a definition that captures an intuitive property of freshness:

Definition: [Guaranteed freshness]
An integrated view is *guaranteed fresh* within time vector $\vec{f}$ after time t_{view_init} if for each $t \geq t_{view\_init}$ there is a $\vec{t'}$ such that $state(V, t) = \nu(state(\vec{DB}, \vec{t'}))$ and $t - t'_i \leq f_i$ for $i \in [1, n]$. (Intuitively, this states that for each time $t \geq t_{view\_init}$, the contents of the view at time t correspond to "recent" states of the source databases. The definition is based on a time vector $\vec{f}$ rather than a single time, to accommodate integration environments where some databases announce updates very quickly, while others announce them only periodically, e.g., once every 24 hours).

4 Overview of Squirrel Mediators

Squirrel mediators support integrated views derived from multiple remote source databases (see Figure 2). A mediator consists of five components: a local store, a query processor (QP), a virtual attributes processor (VAP), an update-queue, and an incremental update processor (IUP). The local store contains the VDP that represents the schema and the derivation relationship between relations in the local store (more details in Section 5), the materialized portion of the view, other supporting materialized data, and a rulebase that specifies the incremental maintenance of the materialized data in a declarative fashion. The QP provides the interface for querying the view. Upon receiving a query against the view, the QP determines first whether the query can be answered solely based on the materialized portion of the view. In case virtual data is needed to answer the query, the QP requests the VAP to construct temporary relations containing the relevant data (see Subsection 6.3). The update-queue holds incremental updates from remote information sources, and the IUP is responsible for propagating the updates accumulated in the queue into the materialized data according to the rules in the rulebase (see Subsection 6.4).

There are three kinds of information flow within an integration mediator. One involves incremental updates against the source databases, which flow into the update-queue; they are then propagated into the integrated view under the control of the IUP. The second kind of information flow is generated by the VAP that sends (receives) queries (answers) to (from) the source databases. The third kind of information flow involves queries posed against the integrated view, and answers made in response to them. Importantly, humans and processes that query the Squirrel mediator need only be aware of the query processor.

There are three ways a source database can be associated with the mediator. First, a source database is called a *materialized-contributor*, if all its contribution to the mediator is in the materialized data portion of the mediator. Second, a source database is called a *hybrid-contributor*, if part of its contribution to the mediator is to the materialized portion and part to the virtual portion. Finally, a database is called a *virtual-contributor*, if it contributes only to the virtual portion.

We now mention several assumptions that are made about the source databases and the network. It is assumed that source databases in the first two categories above actively send messages to the mediator specifying incremental updates that have occurred to the database. In order to guarantee that the integrated view satisfies the correctness conditions defined in Section 3, we assume that the messages (including both incremental updates and query answers) transferred from each source database to the mediator are in order. Also, each message sent from the source database corresponds to a family of one or more consecutive transactions

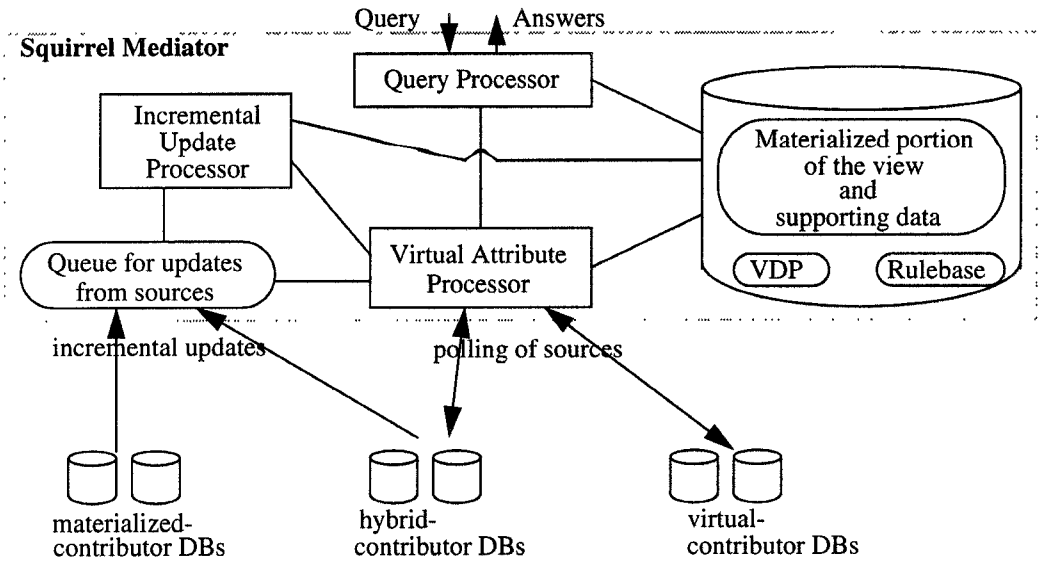

Figure 2: A Squirrel mediator connected with multiple source DBs

that occurred in the database. Databases in the last two categories must be able to answer queries from the virtual data processor, so that the relevant virtual data portion in the mediator can be evaluated when necessary. Since a virtual-contributor database only needs to be able to answer queries, its role can be played by all kinds of DBMSs, including legacy systems that do not have active database capabilities.

5 Annotated View Decomposition Plan

The skeleton of a Squirrel-generated integration mediator is provided by its View Decomposition Plan (VDP). A VDP specifies the relations that the integration mediator will maintain (either materialized, virtual, or hybrid), and provides the basic structure for supporting incremental maintenance of materialized data and evaluation of virtual data. This subsection presents the definition of VDP and gives an example.

As will be defined formally below, the VDP of an integration mediator is a directed acyclic graph (dag) that represents a decomposition of the integrated view supported by that mediator (see Figure 1). The leaf nodes correspond to relations in the source databases, and the other nodes correspond to derived materialized or virtual relations which are maintained by the mediator. Analogous to query execution plans, different VDPs for the same integrated view specification may be appropriate under different query and update characteristics of the application.

The language currently supported by Squirrel for specifying integrated views includes the relational algebra. We use an attribute-based form of the algebra. In the interest of clarity, in the discussion below, we do not consider the use of attribute-renaming (see [HZ96]). Although the view definition language is based on set semantics, some of the relations stored inside an integration mediator may be bags, in order to support our incremental maintenance algorithms; this occurs if the integrated view involves projection or union. Another part of the language specifies "object matching". See [ZHKF95, ZHK95].

5.1 Definition

We now present the definitions of VDP and annotations for them. Formally, a VDP is a labeled dag $\mathcal{V} = (V, E, relation, source, def, Export)$ where V is a set of nodes, E is a set of edges over V, and such that:

1. The function *relation* maps each node $v \in V$ into a specification of a distinct relation, which includes the name of the relation and its attributes. We often refer to a node v by using the name of $relation(v)$.

2. *source* is a function that maps the leaves of V into the set of source databases of the mediator. The leaves of V correspond to relations in the source databases, and are depicted using the $\square$ symbol. Other nodes are depicted using a circle.

3. An edge $(a, b) \in E$ indicates that $relation(a)$ is directly derived from $relation(b)$ (and possibly other relations). If b is a leaf node, a is called *leaf-parent*.

4. For each non-leaf $v \in V$, $def(v)$ is an expression in the view definition language that refers to $\{relation(u) \mid (v, u) \in E\}$. Intuitively speaking, $def(v)$ defines the population of $relation(v)$ in terms of the relations corresponding to the immediate descendants of v. Similar to query execution plans, some combinations of operators are permitted in connection with a single node

of the tree and its immediate descendants, while others are not. The restrictions are follows: (a) the immediate parents of leaf nodes can involve only projection and selection on those leaf nodes. Otherwise for node v, (b) *def*(v) can be arbitrary combination of selects, projects and joins; or (c) *def*(v) can have the form of a union or a difference, with arbitrary selects and projects underneath that. Non-leaf nodes involving difference are called *set nodes*, and all other non-leaf nodes are called *bag nodes*. The relations associated with set nodes are stored as sets, while the relations associated with bag nodes are stored as bags.

5. $Export \subset V$ denotes the set of export relations. Each maximal node (i.e., node with no in-edges) is in $Export$; other non-source nodes may also be in $Export$. Elements of $Export$ are depicted using a double circle.

If $\mathcal{V}$ is a VDP and v is a node and $relation(v)$ has name R and attributes $a_1, ..., a_n$, then an *annotation* for R is a function from $\{a_1, ..., a_n\}$ into $\{m, v\}$, which indicates which attributes are materialized and which are virtual (see Section 2). Given a VDP $\mathcal{V}$, an *annotation* for $\mathcal{V}$ is a function *ann* defined on the set of non-leaf nodes, such that $ann(v)$ is an annotation of the relation of v, for each non-leaf node $v \in V$.

5.2 Rules for update propagation

Every edge (a, b) in a VDP is associated with an update propagation rule which computes an incremental update ($\Delta relation(a)$) to the relation $relation(a)$ based on an update $\Delta relation(b)$. The rules follow the general spirit of previous works, mainly [GMS93, GL95]. Due to the space limitation, we present only one sample rule for relations defined with select/project/join (SPJ). The complete set of rules is presented in [ZHK95].

SPJ: Suppose a relation T in a VDP is defined with SPJ operators: $T = \pi_p \sigma_f (\pi_{p_1} \sigma_{f_1} R_1 \bowtie_{g_1} \ldots \bowtie_{g_{n-1}} \pi_{p_n} \sigma_{f_n} R_n)$. The sub-VDP for T is analogous to the one in Example 2.1, with $V = \{T, R_1, ..., R_n\}$ and $E = \{(T, R_1), ..., (T, R_n)\}$. The rule for the edge (T, R_i) is (using the bag semantics):

rule for SPJ: edge (T, R_i)
on new ΔR_i,
$\Delta T = \pi_p \sigma_f (\pi_{p_1} \sigma_{f_1} R_1 \bowtie_{g_1} \ldots \bowtie_{g_{i-1}} \pi_{p_i} \sigma_{f_i} \Delta R_i \bowtie_{g_i} \ldots \bowtie_{g_{n-1}} \pi_{p_n} \sigma_{f_n} R_n)$;

6 Accessing Virtual Data and Maintaining Materialized Data

This section describes the two primary algorithms used in a Squirrel mediator to support hybrid integrated views. The first algorithm (Subsection 6.3) is performed by the Virtual Attribute Processor (VAP), and materializes temporary relations that hold the "current" value of (projections of) virtual or hybrid relations. These temporary relations might be needed to answer a query, or to incrementally update materialized data. The second algorithm (Subsection 6.4) is performed by the Incremental Update Processor (IUP), and incrementally updates materialized data to reflect updates received from the source databases.

Before the algorithms are presented, Subsection 6.1 gives some technical background used in the description of the algorithms. Also, Subsection 6.2 develops formalism for describing the behavior of the algorithms. During the presentation of the algorithms this formalism will be used to verify certain properties of the algorithms. Section 7 also uses the formalism.

6.1 Tools to manage deltas

The update-queue of a Squirrel mediator holds incremental updates received from the source databases. A Squirrel mediator processes these incremental updates and internal ones for a variety of purposes. This subsection introduces the notation and tools used by the mediator to manipulate such incremental updates.

We use the Heraclitus paradigm [HJ91, GHJ96], which elevates "deltas", or the differences between database states, to be first-class citizens in database programming languages. Speaking loosely, in the relational case, a *delta* (*value*) is simply a set of *insertion atoms* of the form '$+R(\vec{t})$' and *deletion atoms* of the form '$-R(\vec{t})$', subject to the consistency condition: two *conflicting atoms* (i.e., two atoms $+\alpha$ and $-\alpha$) cannot both occur in the delta. A delta can simultaneously contain atoms that refer to more than relation. Deltas have also been generalized to bags [DHR96]. Incremental updates in the update-queue and incremental updates computed during update propagation are represented as deltas.

Two important operators are *apply* and *smash*. Given delta Δ and database state *db*, $apply(db, \Delta)$ denotes the result of applying the atoms in Δ to *db*. If Δ refers only to relation R, we also write $apply(R, \Delta)$. It turns out that *apply* commutes with select and project, e.g., if Δ refers only to R, then $\pi_C \sigma_f apply(R, \Delta) = apply(\pi_C \sigma_f R, \pi_C \sigma_f \Delta)$.

Smash, denoted '!', is a kind of compose operator. In particular, for any state and deltas, $apply(db, \Delta_1 ! \Delta_2) = apply(apply(db, \Delta_1), \Delta_2)$. For the relational case, the smash $\Delta_1 ! \Delta_2$ can be computed by forming the union of Δ_1 and Δ_2, and then deleting any element of Δ_1 that conflicts with an element of Δ_2 [HJ91]. Smash is also relatively easy to compute for bag deltas.

An insertion atom $+R(\vec{t})$ in Δ is *redundant* for state *db* if $\vec{t}$ is in R under *db*; and similarly for deletion atoms. In the context of Squirrel mediators, no atom of any delta that is used is redundant. As a result, the natural *inverse* operator $^{-1}$, that reverses the sign of all atoms in a delta, has the property (for the states and deltas

that arise) $apply(apply(db, \Delta), \Delta^{-1}) = db$. Note also that $(\Delta_1!\Delta_2)^{-1} = \Delta_2^{-1}!\Delta_1^{-1}$

In the discussion below, we assume that an incremental update from a source database is a delta expressed in terms of the relations of the source database. Because each leaf-parent holds a relation which is a project-select of a source database relation, it is easy to "filter" the deltas in the update queue so that they are applicable to the leaf-parent nodes. (A straightforward optimization that can be applied in some cases is to "filter" the incremental updates at the source databases.)

6.2 Constructing the reflect function

This section establishes notation for describing how a Squirrel mediator operates through time, and describes how a "reflect" function $r\vec{e}f$ as in the definition of consistency can be constructed for Squirrel mediators.

We assume that the mediator receives queries asynchronously from the user/applications and incremental updates from the source databases. Internally the mediator performs a series of sequential transactions of two kinds:

Query transaction: Compute the answer to a query

Update transaction: Empty the update-queue and propagate those updates to all affected materialized data in the mediator

In general, transactions of these two kinds will be interleaved, to form a sequence such as

$$t_{view\_init}, t_1^u, t_1^q, t_2^u, t_2^q, t_3^q, t_4^q, t_3^u, t_4^u, t_5^q, \ldots$$

where t_{view_init} is the time when the view is initiated; $t_1^q, t_2^q, t_3^q, t_4^q, t_5^q, \ldots$ are *query transaction times*, i.e., the commit times of the query transactions; and $t_1^u, t_2^u, t_3^u, t_4^u, \ldots$ are *update transaction times.* While the mediator may perform parts of these transactions concurrently, we assume that the implicit serial order of the concurrent execution matches the order of the transaction commit times.

The algorithms of Squirrel mediators are carefully devised to ensure that the mediator is consistent, in the sense defined in Section 3. We now describe how a function $r\vec{e}f$ satisfying the definition of consistency can be constructed for the mediator, based on the timing of the update and query transactions. Let $DB_1, \ldots, DB_n$ be the source databases, and let ν define the view of the mediator. We focus here on the first property of $r\vec{e}f$ in the definition of consistency, namely that for each time t we should have $state(V, t) = \nu(\vec{DB}, r\vec{e}f(t))$.

Let $DB_1, \ldots, DB_m$ be the materialized-contributor and hybrid-contributor databases (see Section 4), and $DB_{m+1}, \ldots, DB_n$ be the virtual-contributor databases. Let V' be the relational database schema that includes all materialized portions of relations in the annotated VDP of the mediator, and let ν' be the function derived from ν and the VDP that specifies how V' should be populated from the source databases $DB_1, \ldots, DB_m$. To describe $r\vec{e}f$, we define first the function $r\vec{e}f' : Time \rightarrow (Time)^m$, with the property that for each update transaction time t_i^u we have $state(V', t_i^u) = \nu'(\vec{DB}[1, m], r\vec{e}f'(t_i^u))$ (where $\vec{DB}[1, m] = \langle DB_1, \ldots, DB_m \rangle$).

Consider an update transaction time t_i^u. As detailed in Subsection 6.4 below, with each execution the IUP processes the full set of deltas "currently" in the queue. Thus, for the execution of the IUP that ends at time t_i^u, there is some earlier time $empty\_queue(t_i^u)$ when the IUP "flushes" the update-queue, and combines the individual updates using smash into a delta value Δ. Time $empty\_queue(t_i^u) > t_{i-1}^u$, because the mediator is assumed to perform the update transactions serially. Furthermore, Δ will hold all incremental updates received by the mediator from the source databases between $empty\_queue(t_{i-1}^u)$ and $empty\_queue(t_i^u)$.

Continuing with our focus on time t_i^u, for each $k \in [1, m]$, let t_k be the time that DB_k sent the last update to the mediator that was received before time $empty\_queue(t_i^u)$. Set $r\vec{e}f'(t_i^u) = (t_1, \ldots, t_m)$. The IUP will guarantee that for each i, $state(V', t_i^u) = \nu'(\vec{DB}[1, m], r\vec{e}f'(t_i^u))$ This will follow from two basic facts: (a) the IUP correctly incorporates the impact of Δ into V', and (b) if virtual data is needed to determine the appropriate incremental update to some relation in V', then an algorithm (Subsection 6.3) is used to guarantee that virtual data from database DB_k corresponds to $state(DB_k, t_k) = state(DB_k, r\vec{e}f'(t_i^u).k)$.

As just defined, the domain of function $r\vec{e}f'$ is $\{t_1^u, t_2^u, \ldots\}$. This is extended to all times $\geq t_{view\_init}$ in the natural fashion: if $t_i^u \leq t < t_{i+1}^u$, then $r\vec{e}f'(t) = r\vec{e}f'(t_i^u)$.

The definition of consistency gives conditions about the state of the integrated view that are supposed to hold at all times. It is sufficient to ensure that these conditions hold at all query execution times. We now describe how the function $r\vec{e}f$ is defined for each query execution time t_j^q.

Suppose first that the query q processed at time t_j^q requires access only to the materialized data in the VDP. This query is answered by simply accessing data in the local store of the mediator. This data will correspond to $\nu'(\vec{DB}[1, m], r\vec{e}f'(t_j^q))$. In this case we define

$$r\vec{e}f(t_j^q).k = \begin{cases} r\vec{e}f'(t_j^q).k & \text{if } k \in [1, m] \\ t_j^q & \text{if } k \in [m+1, n] \end{cases}$$

(Because DB_k is not involved for $k \in [m+1, n]$, the query output does reflect the "current" state of DB_k, i.e., the state at time t_j^q.)

Suppose now that the query q associated with query execution time t_j^q involves virtual data. Under the

algorithm of the VAP, virtual data coming from a hybrid-contributor DB_k will come from the state of DB_k at time $r\vec{e}f'(t_j^q).k$. Virtual data from virtual-contributors is obtained by the VAP through direct querying of the relevant source databases. Suppose that DB_k is a relevant virtual-contributor. The VAP packages all queries against DB_k into a single transaction against DB_k, so that the answers received from DB_k all correspond to a single time, say t_k. In this case we define

$$r\vec{e}f(t_j^q).k = \begin{cases} r\vec{e}f'(t_j^q).k & \text{if } k \in [1,m] \\ t_k & \text{if } k \in [m+1,n] \text{ and } DB_k \text{ contributes to } q \\ t_j^q & \text{if } k \in [m+1,n] \text{ and } DB_k \text{ does not contribute to } q \end{cases}$$

(As before, if DB_k does not contribute to the query, then we use the state of DB_k at the "current" time.)

6.3 Accessing virtual data

This subsection presents how the virtual attribute processor (VAP) of the mediator supports accesses to the virtual data by the query processor (QP) and the incremental update processor (IUP). When QP or IUP needs to evaluate a query $q = \pi_A\sigma_f R$, where A or f contains at least one virtual attribute, the VAP constructs a temporary relation $T = \pi_A\sigma_f R$ as indicated in Section 2. The QP or IUP transmits to the VAP the set I of needed queries involving virtual attributes. The set I is in the form of $\{(R_1, A_1, f_1), ..., (R_m, A_m, f_m)\}$, where (R_i, A_i, f_i) corresponds to the query $T_i = \pi_{A_i}\sigma_{f_i}R_i$. In order to construct the temporary relation T_i, the VAP determines the relations from which T_i is derived. Let S be one of those relations. If at least part of the data in S used to construct T_i is virtual, a temporary relation S' must be constructed before the construction of T_i. S' will correspond to the projection and selection of the portion of S that is referred to in the construction of T_i.

We now introduce a function *derived_from* on a relation T, that returns a set of projections and selections from which T can be derived.

Definition: Let v be a non-leaf node in a VDP, $R = relation(v)$, $attr(R)$ be the whole attribute set of R, $A \subseteq attr(R)$, and f is a selection condition. We define the function $derived\_from(R, A, f) = \{(S_1, B_1, g_1), ..., (S_n, B_n, g_n)\}$, where $A \subseteq attr(R)$ and S_i is a relation in the VDP, g_i is a selection condition, and $B_i \subseteq attr(S_i)$ is minimal such that $\pi_A\sigma_f R$ can be derived from $\pi_{B_i}\sigma_{g_i}S_i$ (possibly together with other relations). More specifically, $\{(S_1, B_1, g_1), ..., (S_n, B_n, g_n)\}$ are determined as follows:[1]

(1) If $def(v) = \pi_C\sigma_h S$ and $D \subseteq attr(S)$ is a set of attributes involved in the select condition h, $derived\_from(R, A, f) = \{(S, A \cup D, h)\}$.

(2) If[2] $def(v) = \pi_C\sigma_h(\pi_{C_1}\sigma_{h_1}S_{i_1} \bowtie_{j_1} \ldots \bowtie_{j_{m-1}} \pi_{C_m}\sigma_{h_m} S_{i_m})$, where $\{i_1, ..., i_m\} \subseteq \{1, ..., n\}$ and D_i is the set of attributes in $attr(S_i)$ that are used in the select and join conditions, $derived\_from(R, A, f) = \{(S_1, B_1, h_1), ..., (S_n, B_n, h_n)\}$, where $B_i = (A \cap attr(S_i)) \cup D_i$.

(3) If $def(v) = (\pi_C\sigma_{h_1}S_1) \cup (\pi_C\sigma_{h_2}S_2)$, where D_i is the set of attributes in h_i, $derived\_from(R, A, f) = \{(S_1, B_1, h_1), (S_2, B_2, h_2)\}$, where $B_i = (A \cap attr(S_i)) \cup D_i$.

(4) If $def(v) = (\pi_C\sigma_{h_1}S_1) - (\pi_C\sigma_{h_2}S_2)$, where D_i is defined analogously as in (3), $derived\_from(R, A, f) = \{(S_1, B_1, h_1), (S_2, B_2, h_2)\}$, where $B_i = (A \cap attr(S_i)) \cup D_i \cup C$.

The execution of the VAP has two phases. During the first phase, all the temporary relations to be constructed are identified. Then they are instantiated in a bottom-up fashion through the VDP during the second phase.

The first phase is specified by the following algorithm:

(1) Initialization: Let *Unprocessed* hold the input set $\{(R_1, A_1, f_1), ..., (R_n, A_n, f_n)\}$, The nodes in the VDP are topologically sorted such that a parent node is considered before its children nodes. The output set *Processed* is set to $\emptyset$.

(2) Processing the *Unprocessed* set: While *Unprocessed* is not empty, for the first $(R, A, f) \in$ *Unprocessed* according to the topological order do:

(2a) Identifying further temporary relations to be constructed: For each $(R', A', f') \in derived\_from(R, A, f)$, if A' contains one or more virtual attributes, then a temporary relation $T = \pi_{A'}\sigma_{f'}R'$ needs to be constructed as well.

(2b) Merging temporary relations: If a triple (R', B, g) is in *Unprocessed*, replace it with $(R', A' \cup B, f' \vee g)$. Otherwise, add (R', A', f') to *Unprocessed*.

(2b) Move (R, A, f) from *Unprocessed* to *Processed*.

We now turn to the second phase of the VAP execution. The output set *Processed* of last phase is the input of this phase. For each triple $(R, A, f) \in$ *Processed*, a temporary relation T is constructed as $\pi_A\sigma_f def(v)$, where $R = relation(v)$. They can be easily constructed in a bottom-up fashion, except for the case where v is a leaf-parent node. The rest of this subsection focuses on the leaf-parent node case.

Suppose there exists $(R, A, f) \in$ *Processed* such that R is associated with a leaf-parent node and the corresponding temporary relation is T. Since R is a virtual or hybrid relation, T can only be derived from a

[1] In the interest of clarity, we present these definitions informally. Also, as indicated in Section 5, we ignore the possibility that renaming of attributes might be in the $def(v)$.

[2] Note the same relation may be referred multiple times in a n-way join formula.

relation S of a hybrid-contributor or virtual-contributor source DB_k. In order to construct T, the VAP queries relation S.

We consider first the case that DB_k is a hybrid-contributor. As pointed out in Subsection 6.2, the data polled from S must be consistent with the materialized data also contributed from the same source where S resides. To guarantee this consistency, we have developed the following mechanism: Using notation defined in Subsection 6.2, suppose t_i^u is the time of the last update of the materialized data in the mediator. The portion of materialized data derived from DB_k corresponds to the $state(DB_k, \vec{ref}(t_i^u).k)$ of DB_k. In this case we desire that the result from querying S also corresponds to this state of DB_k. Note that the source relation S may have been updated since time $\vec{ref}(t_i^u).k$. The "correct" value for S, i.e., the value for S as of time $\vec{ref}(t_i^u).k$ is obtained by "compensating" the current value for S by the inverse of the smash of the updates for S that are in the update-queue up to the time when the result of querying S is received.

The mechanism described above overlaps to certain extent with mechanisms developed in [ZGHW95] and [CGL+96]. Reference [CGL+96] establishes invariants between a source database, a view of that database which may be maintained using deferred updates, and deltas on the source database and view. The algorithm described above is compatible with these invariants. Reference [ZGHW95] considers warehoused views derived from a single source database. The algorithm described in [ZGHW95] has to send compensating queries to the source, while our algorithm performs the compensation locally in the mediator. Unlike our algorithm, the algorithm in [ZGHW95] may have to repeatedly send out compensating queries, if the updates in the source occurs very frequently. In fact, the algorithm may need to execute an arbitrary number of compensating queries without producing an answer to the initial query.

If DB_k is a virtual-contributor, the only restriction to the querying of S' is: for constructing temporary relations for a given view state, each virtual-contributor can only be accessed at most once, so that no more than one state of the same source can contribute to the view state. For our specific case, this means: the VAP finds all the leaf-parent temporary relations specified in the set *Processed* and packages all queries of DB_k into a single transaction (see Subsection 6.2).

6.4 Maintenance of materialized data

The IUP is based primarily on the use of active database techniques to propagate incremental updates from the update-queue into the materialized portions of the VDP. However, because some of the intermediate relations might be virtual or hybrid, in the general case the IUP may have to temporarily materialize some of the virtual data. To describe the behavior of the IUP, we begin with some preliminary comments. We then describe the "IUP Kernel Algorithm". This can be used directly in the simplified context where the updates in the update-queue affect only fully materialized relations that have fully materialized support. We then describe the general IUP algorithm, which generalizes the IUP Kernel Algorithm

As indicated in Subsection 6.2, the IUP process is executed at a series of times $t_1^u, t_2^u, \ldots$ called "update transaction times". In each such transaction, the entire contents of the update-queue are combined into a single delta Δ, which is propagated upwards through the VDP.

Assume now that a VDP $\mathcal{V} = (V, E, class, source, def, Export)$ is fixed. As indicated in Subsection 5.2, based on the definitions of non-leaf nodes, a rule specifying how updates should be propagated is associated with each edge of the VDP. Let *edge_rule* be the function that maps each edge in E to a rule. (This will be independent of annotations for $\mathcal{V}$.) Finally, let $\mathcal{A}$ be an annotation for $\mathcal{V}$.

For the IUP algorithm, two repositories are associated with each non-leaf node v of $\mathcal{V}$. Suppose that $relation(v) = R$. The first repository is denoted simply as 'R'. During execution of the IUP it holds the "current" population of relation R. The second repository is denoted by 'ΔR', and holds the smash of zero or more incremental changes for R that result from the incremental propagation of updates during a single execution of the IUP.

Before continuing, it is convenient to extend the function *edge_rule* to apply to nodes as follows:

$$edge\_rule(v) = \{edge\_rule(v', v) \mid (v', v) \in E\}$$

Intuitively, $edge\_rule(v)$ holds all rules of in-edges to v, i.e., all rules that propagate updates out of v to its parents.

Some care must be taken if incremental updates affect two or more children of a node, as illustrated next.

Example 6.1: Recall Example 2.1, where T, R' and S' are materialized and $T = R' \bowtie S'$. Suppose that updates $\Delta R'$ and $\Delta S'$ have been computed. The rules of Example 2.1 must be applied. However, it would be incorrect to compute

$$\Delta T = (R' \bowtie \Delta S') \cup (\Delta R' \bowtie S')$$

because this will "miss" the contribution of $\Delta R' \bowtie \Delta S'$. One correct solution is to compute

$$\Delta T = (R' \bowtie \Delta S') \cup (\Delta R' \bowtie apply(S', \Delta S'))$$

This captures all of $(R' \bowtie \Delta S')$, $(\Delta R' \bowtie S')$, and $(\Delta R' \bowtie \Delta S')$. □

To avoid the problem of "missing" contributions of deltas illustrated in the previous example, we develop a

systematic approach to firing rules and applying deltas to relations. Let v be a node with $relation(v) = R$. By the phrase *"process node v"* we mean to fire all eligible rules in $edge\_rule(v)$ (in any order), and then to execute the following steps:

$$R := apply(R, \Delta R);$$
$$\Delta R := \emptyset;$$

During IUP processing, a node v will not be processed until all of its children have been processed. As a result, all incremental changes to a node v are accumulated before any of these changes are propagated to parents of v.

The IUP Kernel Algorithm: We now describe the IUP Kernel Algorithm, which can be used directly for the simplified case that all relations affected by the update Δ are fully materialized and have fully materialized support. In this case, the IUP acts as a specialized active database execution model, that applies rules from the rulebase in an order determined by the VDP.

We assume that the queue holding incremental updates from the source databases is nonempty. Let $t = t_i^u$ be the time that this cycle of processing begins. The algorithm proceeds as follows:

(1) Initialization: Let Δ hold the smash of all incremental updates held in the queue at time t. Δ can be broken into a set $\Delta R_1, ..., \Delta R_k$ of subdeltas that refer to some set $R_1, ..., R_k$ of source database relations that are associated with leaf nodes $v_1, ..., v_k$, (respectively) of $\mathcal{V}$. During this phase, two things occur:

(1a) All eligible rules in $\cup\{edge\_rule(v_i) \mid i \in [i, k]\}$ are fired in any order.

(1b) All entries in the queue that contributed to Δ are deleted. (It may be that during the execution of step (1a) additional deltas were added to the queue. These will remain in the queue until the next cycle of rule firing is initiated.)

(2) "Upward" traversal of (V, E). During this phase each non-leaf node is processed according to some topological sort of (V, E), i.e., in an order that satisfies the following restriction: A node v cannot be processed until all of its children have been processed.

Suppose that the above algorithm is executed at time t_i^u, and that the most recent execution of the algorithm that occurred before t_i^u took place at time t_{i-1}^u. The first step of the algorithm propagates the updates in Δ to the nodes that are directly above the leaf nodes of the VDP (although it does not "process" those nodes). It is straightforward to verify that

(A) If a non-leaf node v with $relation(v) = R$ has been processed, then

(i) the repository R will hold the population for R that corresponds the state of the source databases at time $\vec{ref}(t_i^u)$, and

(ii) the associated repository ΔR is empty.

(B) If a non-leaf node v with $relation(v) = R$ has not been processed, then

(i) the repository R will hold the population for R that reflects the state of the source databases at time $\vec{ref}(t_{i-1}^u)$.

(ii) the associated repository ΔR may hold information corresponding to some or all of the incremental updates implied for relation R by the incremental updates to the source databases reported between times t_{i-1}^u and t_i^u.

(C) A node v with $relation(v) = R$ is not processed until all contributions to ΔR have been computed.

In particular, then, after execution terminates, all of the affected nodes will hold the state reflecting all updates that reached the queue up until the time of flushing the queue, i.e., $empty\_queue(t_i^u)$ in the notation of Subsection 6.2. As a result, we have $state(V, t_i^u) = \nu'(\vec{DB}[1, m], \vec{ref}'(t_i^u))$ as promised in Subsection 6.2.

The general IUP algorithm: We now describe the general IUP algorithm,, that supports incremental update to arbitrary materialized and hybrid relations. We again assume that this update transaction has time t_i^u. The general execution of IUP has three phases, namely (a) determine needed temporary relations; (b) populate needed temporary relations (using the VAP); and (c) propagate updates.

Example 6.2: To understand why temporary relations are needed, recall Example 2.2, where T and S' are materialized but R' is virtual. If during processing $\Delta S'$ becomes non-empty, then the value of R' must be obtained, in order to apply rule #1 of Example 2.1. □

To perform phase (a), i.e., to determine the needed temporary relations, we apply a variant of the IUP Kernel Algorithm, called the IUP Preparation Algorithm. In this algorithm, the IUP Kernel Algorithm is simulated, to see what rules would be fired based on the delta Δ obtained from the update-queue. This yields a set $I = \{(R_1, A_1, f_1), ..., (R_n, A_n, f_n)\}$ of projections of selections of hybrid and virtual relations that will be needed. (If two temporary relations (R, A, f) and (R, A', f') are needed for the same node, then these are merged.)

The second phase (b) is simply a call to the VAP to obtain temporary relations for all of I. Note that all source databases that are accessed are hybrid contributors. [Materialized data in the VDP depends on them because they were identified by phase (a), and virtual data in the VDP depends on them because they are being accessed to populate relations in I with virtual attributes.]

Because the input queue holds all incremental updates received after t^u_{i-1}, the VAP populates I to the state corresponding to $\vec{ref}'(t^u_{i-1})$. This is appropriate, because some of the updates in Δ might imply updates that are to be applied to the materialized portions of hybrid relations that participate in I.

Phase (c) of the general IUP algorithm is to apply the IUP Kernel Algorithm, except using the temporary relations in place of their associated hybrid or virtual relations. When the IUP Kernel Algorithm is finished, update the materialized part of any hybrid relation according to the value of the associated temporary relation. By the properties of the IUP Kernel Algorithm discussed above, after completion of these steps the materialized portion of the VDP will correspond to $state(\vec{DB}[1,m], \vec{ref}'(t^u_i))$.

7 Correctness of Squirrel Mediators

This section briefly sketches the proofs that Squirrel mediators are both consistent and (under natural assumptions) guarantee freshness.

Theorem 7.1: Squirrel mediators are consistent.

Proof: (sketch) To prove consistency, we begin with the function $\vec{ref}$ constructed in Subsection 6.2. As verified during the presentations of the IUP and VAP algorithms, $state(V, t^q_j) = \nu(\vec{DB}, \vec{ref}(t^q_j))$ and so Condition 1 of the definition of consistency is satisfied. Also, it is straightforward to verify that Conditions 2 and 3 of the definition consistency. □

To show that mediators guarantee freshness, we first identify some natural properties that the overall environment can be assumed to satisfy. The following lengths of time are important:

- *announcement delay*$_i$ (ann_delay_i): the period between the time when updates are committed in database DB_i and the updates are "announced" to the mediator
- *communication delay*$_i$ ($comm\_delay_i$): the time it takes for messages from DB_i to reach the mediator, and for messages from the mediator to reach DB_i.
- *update holding delay*$_{med}$ ($u\_hold\_delay_{med}$): the length of time in the worst case between when an update might arrive at the mediator and when the mediator starts the next update transaction.
- *update processing delay*$_{med}$ ($u\_proc\_delay_{med}$): the length of time in the worst case between when the mediator starts an update transaction and completes it, excluding the time taken to query the source databases.
- *query processing delay*$_i$ ($q\_proc\_delay_i$): the length of time it takes in the worst case for DB_i to answer a query posed against it. (It is 0 if queries are not made against DB_i.)
- *query processing delay*$_{med}$ ($q\_proc\_delay_{med}$): the length of time it takes in the worst case for the mediator to do processing associated with the QP and VAP, excluding the time taken to query the source databases.

Both ann_delay_i and $u\_proc\_delay_{med}$ are dictated primarily by policies, e.g., on how often source databases should transmit updates to the mediator, and how frequently the mediator should flush its incremental update queue.

We now state:

Theorem 7.2: Suppose that in an integration environment involving source databases $DB_1, \ldots, DB_n$ and Squirrel mediator, the delays are bounded by ann_delay_i, $comm\_delay_i$, $u\_hold\_delay_{med}$, $u\_proc\_delay_{med}$, $q\_proc\_delay_i$, and $q\_proc\_delay_{med}$. Define vector $\vec{f}$ so that for materialized- and hybrid-contributors DB_i and virtual-contributors DB_j we have:

$$\begin{aligned} f_i &= ann\_delay_i + comm\_delay_i \\ &\quad + u\_hold\_delay_{med} + u\_proc\_delay_{med} \\ &\quad + \Sigma_{k=1}^{n}(q\_proc\_delay_k + 2 * comm\_delay_k) \\ &\quad + q\_proc\_delay_{med} \end{aligned}$$

$$\begin{aligned} f_j &= \Sigma_{k=1}^{n}(q\_proc\_delay_k + 2 * comm\_delay_k) \\ &\quad + q\_proc\_delay_{med} \end{aligned}$$

Then the integration environment is guaranteed fresh within $\vec{f}$ after time t_{view_init}.

Proof: (sketch) From the proof of Theorem 7.1 we know that for each query transaction time t^q_j, $state(V, t^q_j) = \nu(\vec{DB}, \vec{ref}(t^q_j))$. Also, it is easy to verify that for each source database DB_i we have $t^q_j - \vec{ref}(t^q_j).i \leq f_i$. □

The choice of $\vec{f}$ in the above theorem reflects the worst case, in which a query against the mediator must access all hybrid- and virtual-contributor source databases. Importantly, if a query against fully materialized data is asked, then the "freshness" for each materialized-contributor database DB_i is

$$\begin{aligned} f_i &= ann\_delay_i + comm\_delay_i \\ &\quad + u\_hold\_delay_{med} + u\_proc\_delay_{med} \end{aligned}$$

8 Conclusions

This paper presents Squirrel mediators, which support views that integrate data from multiple data sources, and support the materialized and virtual approaches, and hybrids of them. An important topic for future research is to develop guidelines for choosing optimal VDPs and annotations for Squirrel mediators. One step in this direction will be to benchmark various alternatives under different environmental conditions.

Another important topic is to develop algorithms for migrating to new VDPs and/or annotations in a Squirrel mediator, as the underlying environment changes. These changes might include the addition/deletion of data sources and/or export relations, network conditions, and query and update frequencies.

References

[ACHK93] Y. Arens, C.Y. Chee, C.N. Hsu, and C.A. Knoblock. Retrieving and integrating data from multiple information sources. *Intl. Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[ADD+91] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, and M. C. Shan. Pegasus heterogeneous multidatabase system. *IEEE Computer*, December 1991.

[BDD+95] O. Boucelma, J. Dalrymple, M. Doherty, J. C. Franchitti, R. Hull, R. King, and G. Zhou. Incorporating Active and Multi-database-state Services into an OSA-Compliant Interoperability Framework. In *The Collected Arcadia Papers, Second Edition.* University of California, Irvine, May 1995.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 61–71, 1986.

[CGL+96] L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, and H. Trickery. Algorithms for deferred view maintenance. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 469–480, 1996.

[Cha94] T.-P. Chang. *On Incremental Update Propagation Between Object-Based Databases.* PhD thesis, University of Southern California, Los Angeles, CA, 1994.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[Dal95] J. Dalrymple. *Extending Rule Mechanisms for the Construction of Interoperable Systems.* PhD thesis, University of Colorado, Boulder, 1995.

[DH84] U. Dayal and H.Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. on Software Engineering*, SE-10(6):628–644, 1984.

[DHR96] M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 306–317, 1996.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Trans. on Database Systems*, 1996. To appear. Available via anonymous ftp at ftp://ftp.cs.colorado.edu//users/hull/heratech94-revised.ps.

[GJM96] A. Gupta, H.V. Jagadish, and I.S. Mumick. Data integration using self-maintainable views. In *Proc. of Intl. Conf. on Extending Data Base Technology*, 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 328–339, 1995.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 157–166, 1993.

[HJ91] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1991.

[HZ96] R. Hull and G. Zhou. A framework for optimizing data integration using the materialized and virtual approaches. Technical report, Computer Science Department, University of Colorado, 1996. in preparation.

[LMR90] W. Litwin, L. Mark, and N. Roussopolos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[SBG+81] J. M. Smith, P. A. Bernstein, N. Goodman, U. Dayal, T. Landers, K.W.T. Lin, and E. Wong. Multibase – Integrating heterogenous distributed database systems. In *National Computer Conference*, pages 487–499, 1981.

[T+90] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22(3):237–266, September 1990.

[WHW89] S. Widjojo, R. Hull, and D. Wile. Distributed Information Sharing using WorldBase. *IEEE Office Knowledge Engineering*, 3(2):17–26, August 1989.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 316–327, San Jose, California, May 1995.

[ZHK95] G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization, 1995. To appear, *Journal of Intelligent Information Systems.*

[ZHKF95] G. Zhou, R. Hull, R. King, and J-C. Franchitti. Using object matching and materialization to integrate heterogeneous databases. In *Proc. of Third Intl. Conf. on Cooperative Information Systems (CoopIS-95)*, Vienna, Austria, May 1995.