, , 1–24 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Generating Data Integration Mediators that Use Materialization[*]

GANG ZHOU[**], RICHARD HULL, AND ROGER KING {gzhou, hull, king}@cs.colorado.edu
Computer Science Department, University of Colorado, Boulder, CO 80309-0430

Abstract. This paper presents a framework for data integration that is based on using "Squirrel integration mediators" that use materialization to support integrated views over multiple databases. These mediators generalize techniques from active databases to provide incremental propagation of updates to the materialized views. A framework based on "View Decomposition Plans" for optimizing the support of materialized integrated views is introduced. The paper describes the Squirrel mediator generator currently under development, which can generate the mediators based on high-level specifications.

The integration of information by Squirrel mediators is expressed primarily through an extended version of a standard query language, that can refer to data from multiple information sources. In addition to materializing an integrated view of data, these mediators can monitor conditions that span multiple sources. The Squirrel framework also provides efficient support for the problem of "object matching", that is, determining when object representations (e.g., OIDs) in different databases correspond to the same object-in-the-world, even if a universal key is not available.

To establish a context for the research, the paper presents a taxonomy that surveys a broad variety of approaches to supporting and maintaining integrated views.

Keywords: materialized integrated view, integration mediator, activeness, view decomposition plan

1. Introduction

Given the advent of the information super-highway, an increasingly important computer science problem today is to develop flexible mechanisms for effectively integrating information from heterogeneous and geographically distributed information sources. The traditional approach is to support a *virtual* integrated view, and to support queries against the view by query decomposition, query shipping, and integration of query results [3], [22], [25]. More recently, the use of *materialization* has been gaining increasing attention in connection with supporting both single-source and integrated views [2], [26], [23], [29]. There are a variety of situations under which materialization is preferable to the virtual approach, e.g., cases where network connectivity is unreliable, where response-time to queries is critical, or where it is cheaper to materialize and incrementally maintain intricate relationships rather than re-compute them each time they are needed for a particular query answer. The primary contribution of the research presented in this paper is the description

[*] This research was supported in part by NSF grant IRI-931832, and ARPA grants BAA-92-1092 and 33825-RT-AAS. A preliminary version of some of the material presented here appears in [28].

[**] A student at the University of Southern California, in residence at the University of Colorado.

of a prototype tool that can generate systems that support data integration using materialized integrated views.

A central component of our framework is the notion of "Squirrel integration mediator" (or Squirrel mediator). As detailed below, these provide a variety of mechanisms for supporting and incrementally maintaining materialized integrated views. Squirrel mediators are implemented as special purpose "active modules" [6], [11]; these are software components whose behavior is specified largely by rules, in the spirit of active databases. The rules permit a relatively declarative style of programming, thus increasing reusability and maintainability. The primary components of a Squirrel mediator are a local store for the materialized integrated view and auxiliary information, rules for incremental maintenance of the view, and an execution model for applying these rules.

Squirrel mediators extend existing techniques [5], [8], [9], [18], [19] for the incremental maintenance of materialized views defined over a single database in two fundamental ways. First, Squirrel mediators can support materialized integrated views over multiple databases. These mediators materialize both the classes for export and also auxiliary classes, so that maintenance can be performed using exclusively incremental updates from the source databases. (This contrasts with the approach of [29], where only export classes would be materialized. Under that approach, export classes is maintained using incremental updates from the source databases and polling of the source databases.) Second, Squirrel mediators are based on "View Decomposition Plans" (VDPs), which serve as the skeletons for supporting materialized integrated views, providing both data structures for holding the required auxiliary information, and serving as the basis for the rulebase. VDPs provide a broad framework for optimizing support for integrated views, in a manner reminiscent of query execution plans used in traditional query optimization (as described in, e.g., [1]).

In the Squirrel project at the University of Colorado we are currently developing a prototype generator that can be used to generate Squirrel mediators as described above. The current Squirrel generator takes as input the specification of the integrated view to be constructed, expressed in a high-level Integration Specification Language (ISL). The specification includes primarily how the data from various sources is to be integrated. For this purpose, a generalization of a standard query language is used. As output, the generator produces a Squirrel mediator. When invoked, the mediator first initializes the integrated view and sends to the source databases specifications of the incremental update information that they are to transmit back to the mediator. Then the mediator maintains the integrated view and answers queries against it. In order to construct the Squirrel generator, we have developed a systematic approach to building Squirrel mediators, that is based largely on the use of VDPs.

A novel feature of the mediators generated by Squirrel generator is that they can provide efficient support for monitoring conditions based on information from multiple sources. This is accomplished by materializing and incrementally maintaining information relevant to these conditions. In this manner, a mediator can send an alert as soon as updates received from the source databases indicate that a condition has been violated.

A second novel feature of Squirrel mediators is the support they can provide for "object matching", that is, determining when two object representations (e.g., keys in the relational model or object identifiers in an object-oriented model) from two different databases refer to the same object-in-the-world. In this regard, Squirrel mediators build on previous systems that support full [26] or partial [2], [21] materialization for supporting integrated views. In particular, Squirrel mediators can accommodate a variety of complex criteria for matching objects, including "look-up tables", user-defined functions, boolean conditions, historical conditions, and intricate heuristics.

The current Squirrel prototype is focused on a small portion of the full space of possible approaches to data integration. Indeed, modern data integration applications involve a broad array of issues, including the kinds of data, the capabilities of data repositories, the resources available at the site of the mediator (e.g., storage capacity), and the requirements on the integrated view (e.g., query response time and up-to-dateness). No single approach to supporting data integration can be universally applied. To better understand the impact of those issues on data integration, and provide a larger context within which to understand the Squirrel framework, we include in this paper a survey of issues and techniques that arise in data integration, with an emphasis on those issues that affect approaches based on materialization. This survey is presented in the form of a taxonomy based on several spectra, including for example a spectrum about the degree of materialization, which ranges from fully materialized to fully virtual, and spectra concerning different ways to keep materialized data up-to-date. This taxonomy will be used in the future development of Squirrel, both guiding the choice of extensions, and in permitting modular support for different kinds of features.

The rest of the paper is organized as follows: Section 2 describes related work that this research is based upon. Section 3 gives a motivating example that illustrates several aspects of our approach. Section 4 gives a high level description of the Squirrel framework, including descriptions of the ISL, View Decomposition Plans, and the generation of Squirrel mediators from ISL specifications. Section 5 presents the taxonomy of the space of approaches to data integration. Brief conclusions are given in Section 6.

Due to space limitations, the presentation here is rather terse; further details may be found in [27].

2. Preliminaries

This section briefly surveys two of the technologies that are used by Squirrel.

The Heraclitus Paradigm: Squirrel mediators use incremental updates to maintain materialized integrated views. The notation and tools used to manipulate such incremental updates are introduced now.

We use the Heraclitus paradigm [20] which elevates "deltas", or the differences between database states, to be first-class citizens in database programming languages. This paradigm has been developed for relations [17], [16], for bags [13], and for the object-oriented database model [6], [12]. We illustrate key elements of the paradigm here in the context of the relational model. Speaking loosely, 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 one relation.

Three important operators for deltas are *apply*, *smash*, and `when`. Given delta Δ and database state db, $apply(db, \Delta)$ denotes the result of applying the atoms in Δ to db.

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 [20]. Smash is also relatively easy to compute for bag and object-oriented deltas.

Finally, the operator `when` permits efficient access to hypothetical states of a database, without modifying the current database state. In particular, the expression 'q `when` Δ' yields the value of query q on the state that *would* arise if Δ were applied to the current state. Squirrel mediators are implemented in the language Heraclitus[Alg,C] [16], which extends C to include persistent relations and deltas.

Immutable OIDs for export: One subtlety concerning object identifiers (OIDs) is that from a formal perspective, only the relationship of the OID to values and other OIDs in a database state is important [4]; the particular value of an OID is irrelevant. As a result, a DBMS is free to change the specific values of OIDs, as long as its internal state remains "OID-isomorphic" [1] to the original state. This may create a problem if OIDs from a source database are used to represent information in the local store of a Squirrel mediator.

To overcome this problem, we generally assume that the relevant physical OIDs in a source database are immutable. If a source database does not use immutable OIDs, then we follow the technique of [14], and assume that these source databases have been wrapped to support immutable OIDs for export.

3. A Motivating Example and Intuitive Remarks

This section gives an informal overview, based on a very simple example, of several key aspects of the Squirrel framework for data integration using Squirrel mediators. Section 4 describes the Squirrel framework in more detail.

In the example there are two databases, `StudentDB` and `EmployeeDB`, that hold information about students at a university and employees in a large nearby corporation, respectively. A Squirrel mediator, called here `S_E_Mediator`, will maintain an integrated view about persons who are both students and employees, providing

```
Source-DB: StudentDB
  interface Student {
    extent      students;
    string      studName;
    integer[7] studID;
    string      major;
    string      local_address;
    string      perm_address;
    };
  key: studID

Source-DB: EmployeeDB
  interface Employee {
    extent      employees;
    string      empName;
    integer[9] SSN;
    string      divName;
    string      address;
    };
  key: SSN
```

```
Correspondence S_E_match:
  Match classes:
    s IN StudentDB:Student,
    e IN EmployeeDB:Employee
  Match predicates:
    close_names(s.studName, e.empName)
    AND (e.address = s.local_address
    OR e.address = s.perm_address)
  Match object files:
    $home/demo/close_names.o

Export classes:
  DEFINE VIEW Student_Employee
  SELECT s.studName,s.major,e.divName
  FROM s IN StudentDB:Student,
       e IN EmployeeDB:Employee
  WHERE S_E_match(s,e);

Conditions:
  Condition:
    count(Student_Employee) =< 100
  Action:
    send_warning('count exceeded')
```

Figure 1. The ISL specification of the example problem

their names, majors, and names of the divisions where they work. The mediator will also monitor the condition that no more than 100 students are employees.

Figure 1 gives a high level specification (in our ISL language, see Section 4.2) of the data integration problem. This ISL specification includes primarily the relevant subschemas of the two source databases (in the `Source-DB` parts), and the definition of the integrated view (in the `Export classes` part). In this example the view consists of only one class; in general the view might include several classes. In the example, there is not a universal key between students and employees. However, the ISL specification includes a description of how students and employees can be matched, in the `Correspondence` part (see below). Note that the function `S_E_match` defined by that correspondence is used in the specification of the view. Finally, the `Conditions` part of the ISL specification includes the condition to be continuously monitored.

We now consider in a little more detail how `S_E_Mediator` (a) provides support for object matching, (b) uses rules to support incremental maintenance of materialized data, and (c) monitors the condition.

With regards to issue (a), the `Match predicate` in the `Correspondence` part of the ISL specification incidates that a student object *s matches* an employee object e if (1) either s.`local_address` = e.`address` or s.`perm_address` = e.`address`, and (2) their names are "close" to each other according to some metric, for instance, where different conventions about middle names and nick names might be permitted. The "closeness" of names is determined by a user-defined function, called here `close_names()`, that takes two names as arguments and returns a boolean value. (More intricate match criteria can also be supported.)

Following the default approach used by Squirrel, object matching between students and employees is supported in `S_E_Mediator` by having the local store hold a "match" class, in this case called `match_Stud_Emp`, that essentially holds the "outer join" of the `Student` and `Employee` classes. For each person who is both student and employee there will be one "surrogate" object in `match_Stud_Emp` that represents this person; for each person who is a student but not an employee there will be one "surrogate" object in `match_Stud_Emp`, several of whose attributes will be `nil`; and likewise for employees who are not students. This match class is used by the Squirrel mediator to support the derived boolean relation `S_E_match` referred to in the definition of the view class `Student_Employee`.

The class `match_Stud_Emp` illustrates one kind of intricate relationship between data from multiple sources which is expensive to compute. By using materialization, this relationship can be computed when `S_E_mediator` is initialized, and then maintained incrementally as relevant data in the source databases changes. In general, the query response time obtained by using this materialized approach to data integration will be faster than when using the virtual approach, where the potentially expensive step of identifying matching pairs of objects may be incurred with each query against the view. Also, we expect that if the update-to-query ratio is sufficiently small, then the materialized approach will also be more efficient on average than the virtual approach.

In this simple example, the export view class `Student_Employee` is a simple projection and selection of the class `match_Stud_Emp`. Thus, `S_E_Mediator` can support this class in a virtual fashion, translating queries against the view into queries against `match_Stud_Emp`. In general, a Squirrel mediator may materialize some export view classes, and support others as selections and projections of other materialized classes.

We now turn to issue (b), that of incrementally maintaining materialized data in the Squirrel mediator. Two basic issues arise: (i) importing information from the source databases and (ii) correctly maintaining the materialized data to reflect changes to the source databases. For this example, with regards to (i) we assume that both source databases can actively send messages containing the net effects of updates (i.e., insertions, deletions, and modifications) to `S_E_Mediator`. A rulebase in the Squirrel mediator is used to perform (ii). To illustrate briefly, we informally describe two representative rules involved in supporting the class `match_Stud_Emp`. The two rules correspond to the creation of new `Student` objects in the source database `StudentDB`.

Rule `R1`: If an object of class `Student` is created, insert a corresponding new object into class `match_Stud_Emp` whose Employee-attributes are `nil`.

Rule `R2`: Upon the insertion of a `match_Stud_Emp` object x with `nil` Employee-attributes, if there is a corresponding object y in `match_Stud_Emp` with `nil`

Student-attributes that matches x, then delete x and modify y by replacing its `nil` attributes with values from x.

The complete rulebase would include rules dealing with creation, deletion, and modification of objects in both source databases (see Subsections 4.5 and 4.6).

Finally, we indicate (c) how the condition `count(Student_Employee) =< 100` is monitored. This is a particularly simple case, because the only class mentioned in the condition is one of the export view classes. In this case, the condition is monitored by rules that incrementally maintain the count of tuples in `match_Stud_Emp` that have no `nil` values. More generally, a condition may refer to data that is not represented by any of the export view classes. In that case, the classes holding data relevant to the condition are materialized, and rules are used to incrementally maintain these classes and monitor the truth-value of the condition on them.

Importantly, the Squirrel mediator can alert a user that the condition has been violated as soon as the relevant updates to the source databases are transmitted to the mediator. If a virtual approach to supporting the integrated view were used, then the condition could be monitored only by periodically asking a query that called for the count of `Student_Employee`. This would involve repeated accesses to the two source databases, and might not alert the user of violation of the condition as quickly as the materialized approach.

4. The Squirrel Integration Mediator Generator

We are currently developing a prototype tool called Squirrel mediator generator. The Squirrel generator takes as input a high-level specification of an integrated view to be supported, and produces as output a mediator that supports it. One of the challenges in designing the Squirrel generator was to develop a systematic and uniform methodology for constructing such mediators from high-level specifications. In this section we describe both the methodology and the Squirrel mediators that are produced by it.

The section begins with a high-level description of how Squirrel mediators are generated. (Subsection 4.1). Next, the high-level Integration Specification Language (ISL) (Subsection 4.2) is described. The skeleton of a Squirrel mediator is provided by its View Decomposition Plan (VDP); this is described in Subsection 4.3. The next two subsections (4.4 and 4.5) describe the execution model used by Squirrel mediators, and also indicate how incremental updates are propagated through the various materialized classes stored by these mediators. Subsection 4.6 describes how VDPs and rule-bases are constructed. A final component of our solution is the automatic generation of rules to be incorporated into the rulebases of the source databases, so that relevant updates will be propagated to the mediator. We do not address the generation of those rules here.

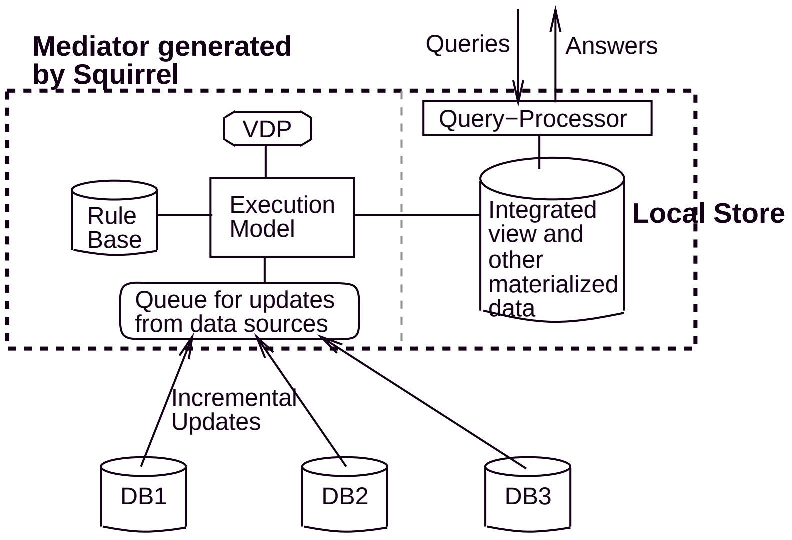
Mediator generated by Squirrel
Queries
Answers
Query−Processor
VDP
Execution Model
Rule Base
Integrated view and other materialized data
Local Store
Queue for updates from data sources
Incremental Updates
DB1
DB2
DB3

Figure 2. Configuration of a Squirrel mediator connected with three source DBs

4.1. An overview of the automatic generation of Squirrel mediators

This subsection gives a brief overview of how Squirrel mediators are generated. Further detail is given in the subsequent subsections, where the various components of Squirrel mediators are described.

The overall architecture of a Squirrel mediator is shown in Figure 2. A Squirrel mediator consists of six components – an update-queue that holds incremental updates from remote information sources, a VDP, a rulebase, an execution model, a local persistent store, and a query processor that accepts queries against the view.

There are two kinds of information flow within a Squirrel mediator. One involves incremental updates against the source databases, which flow into the queue; as a result of the execution model (applied to the rulebase and VDP) these incremental updates then propagate into the integrated view. The other 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 and the local store.

The process of generating Squirrel mediators from an ISL specification is illustrated in Figure 3. The software modules corresponding to the components of a Squirrel mediator can be divided into two groups with regards to the construction of the mediators. The first group includes three modules, namely the execution model, query processor, and update-queue handler. These modules are independent from any particular ISL specification and are kept in the *Squirrel library.* The second group of modules includes the VDP, the rulebase, and the initialization

Figure 3. The process of automatically generating Squirrel mediator from an ISL specification

module. The latter initializes the local store and (possibly) creates rules for the remote source databases. Those modules must be tailored to particular ISL specifications, and are generated dynamically by Squirrel's *ISL compiler* from the ISL specification. More specifically, the ISL compiler reads in an ISL specification and outputs the three modules in Heraclitus[Alg,C] code. As mentioned in Section 2, Heraclitus[Alg,C] is a database programming language that provides notation and constructs that are convenient for implementing various software modules of the Squirrel mediator. Since these modules are in Heraclitus[Alg,C] code which is relatively high-level, the user has the freedom to modify them, e.g., by adding new rules or modifying the VDP. The final executable Squirrel mediator is created by pushing the three generated modules through the Heraclitus[Alg,C] compiler, and linking the result with the modules from the Squirrel library. In the remainder of this section we discuss the ISL and the three most important components of Squirrel mediators, namely, the VDP, the execution model, and the rulebase.

4.2. Squirrel Integration Specification Language (ISL)

The Integration Specification Language (ISL) allows users to specify their data integration applications in a largely declarative fashion. The language currently supported by Squirrel for specifying integrated views includes rich object matching criteria and a subset of ODMG's OQL [7] that corresponds to the relational algebraic operators selection, projection, join, union, and set difference, where both imported and exported classes may be sets or bags. In the discussion here we focus on the case where the imported and exported classes are sets; the extension to bags is straightforward. Importantly, even in the case where imported and exported classes are restricted to be sets, some of the classes stored inside a Squirrel mediator may be bags; this occurs if the integrated view involves projection or union. The primary focus of ISL is on the specification of matching predicates, of integrated views, and of conditions to be monitored. An ISL specification for the Student/Employee example is shown in Figure 1 in Section 3. We now briefly describe the four parts of the ISL (see [27] for more details).

(1) *Source DB subschemas* describe relevant subschemas of the source databases. A key may optionally be specified for each class.

(2) *Correspondence specifications* describe match criteria between objects of families of corresponding classes. A correspondence specification for a given family of classes has three parts: (a) *Match classes* part lists the classes that are matched in this specification, and indicates the ranges of variables used in the match predicates. (b) *Match predicate* part is a binary matching predicate specifying correspondences between objects from two classes. The predicates can be based on, among other things, boolean relations or user-defined functions (that may in turn refer to "look-up tables" or intricate heuristics). In the case of n-ary matching, the full correspondence is expressed using a set of binary match predicates. We are developing extensions to incorporate historical conditions and heuristics expressed as rules. (c) *Match object files (optional)* part specifies the path(s) of the object file(s) containing the implementation of user-defined comparison function(s).

(3) *Distinguished classes* include all export and possibly some internal classes. The definition of a distinguished class may refer both to source database classes and to distinguished classes that are already defined.

(4) *Conditions* include rules for monitoring conditions. The conditions may refer to source classes, distinguished classes, and/or user-defined functions.

4.3. View decomposition plans (VDPs)

The skeleton of a Squirrel mediator is provided by its View Decomposition Plan (VDP). A VDP specifies the classes (both distinguished and other) that the mediator will maintain, and provides the basic structure for supporting incremental maintenance. As noted in the Introduction, VDPs are analogous to query execution plans as used in query optimization. This subsection presents a definition of VDP and gives several examples.

As will be defined formally below, the VDP of a Squirrel mediator is a directed acyclic graph (dag) that represents a decomposition of the integrated view supported by that mediator. The leaf nodes correspond to classes in the source databases, and the other nodes correspond to derived classes which are materialized and maintained by the Squirrel mediator. Some non-leaf nodes, including all maximal nodes of the VDP, correspond to the distinguished (i.e., export and internal) classes in the ISL specification for the Squirrel mediator. An edge from node u to node v in a VDP indicates that the class of v is used directly in the derivation of the class of u. In general, the propagation of incremental updates will proceed along the edges, from the leaves to the top of a VDP. Analogous to query execution plans, different VDPs for the same ISL specification may be appropriate under different query and update characteristics of the application.

The framework developed here can be used with both the object-oriented and relational data models. For the sake of conciseness, we describe the framework by using the relational algebra syntax, which can be mapped to the OQL syntax.

Formally, a VDP is a labeled dag $\mathcal{V} = (V, E, \mathit{class}, \mathit{Source}, \mathit{def}, \mathit{Dist})$ such that:

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

2. *Source* is a possibly empty subset of V that contains some or all of the leaves of the dag. Nodes in *Source* correspond to classes in the source databases, and are depicted using the $\Box$ symbol. Other nodes are depicted using a circle. In a "complete" VDP each leaf is a source database class; VDPs whose leaves are not source database classes are used in Subsection 4.6 to describe the construction of "complete" VDPs.

3. An edge $(a, b) \in E$ indicates that $class(a)$ is directly derived from $class(b)$ (and possibly other classes).

4. For each non-leaf $v \in V$, $def(v)$ is an expression in the view definition language that refers to $\{class(u) \mid (v, u) \in E\}$. Intuitively speaking, $def(v)$ defines the population of $class(v)$ in terms of the classes corresponding to the immediate descendants of v. The expressions used to define a class in terms of other classes are restricted. The restrictions are follows: (a) the immediate parents of leaf nodes can involve only projection, selection, and object matching 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. *Dist* $\subset V$ denotes the set of distinguished classes. These correspond to the internal and export classes specified in the ISL. Each maximal node (i.e., node with no in-edges) is in *Dist*; other non-source nodes may also be in *Dist*. Elements of *Dist* are depicted using a double circle.

Example 1: Let $R(r_1, r_2, r_3, r_4)$ and $S(s_1, s_2, s_3)$ be two classes from distinct databases. Suppose that the integrated view for a Squirrel mediator has the single class $T = \pi_{r_1,s_2}(\sigma_f R \bowtie_{r_2=s_1} \sigma_g S \bowtie_{s_2=r_3} \sigma_h R)$. A VDP $\mathcal{V}_1$ for T is shown in Figure 4. The dotted line separates the mediator classes from the source classes. There are four non-leaf classes in the VDP, namely T, R'_1, S', and R'_2. The attributes of the classes are shown next to the non-leaf nodes. R'_1, R'_2, and S' serve as auxiliary data, so that T can be maintained using incremental updates from the sources and information local to the mediator.

An alternative to $\mathcal{V}_1$ is VDP $\mathcal{V}_2$, where nodes R'_1 and R'_2 are merged into node R'. The join in $\mathcal{V}_1$ can use R'_1 and R'_2 directly, while the join in $\mathcal{V}_2$ must use selections of R'. The combination of R'_1 and R'_2 will generally take less space than R'. For example, if an object (a_1, a_2, a_3) satisfies only the selection condition f, the whole object is in the class R', but only the projection (a_1, a_2) would be in R'_1. On the other hand, incremental maintenance of R' may be more efficient than that of R'_1 and R'_2, because an update to the class R needs to be processed only once in the former case. Each of the non-leaf nodes in both VDPs here are bag nodes. $\Box$

Figure 5. $T = (\pi\sigma(R_1\ match_\gamma\ R_2) - \pi\sigma(R_3)) - \pi\sigma(R_4 \bowtie R_5)$

In the previous examples, the integrated view involves only one export class. In the next example we give a complex VDP involving several distinguished nodes.

Example 2: Among other export classes of some Squirrel mediator, let class T be defined as $(\pi\sigma(R_1\ match_\gamma\ R_2) - \pi\sigma(R_3)) - \pi\sigma(R_4 \bowtie R_5)$. Here $M = (R_1\ match_\gamma\ R_2)$ is a match class based on matching predicate γ (as illustrated in the Student/Employee example in Section 3), which contains object correspondence information for objects in classes R_1 and R_2. To simplify the exposition, selection and join conditions and projection attributes are omitted in the view definition. A VDP supporting this class is shown in Figure 5. The VDP includes three distinguished classes, W, R_4', and M, in addition to T. The grey edges coming from W indicates that W also relies on other classes not shown in the figure. In this VDP, M, Q, and T are set nodes, and P, R_3', R_4', R_5' and S are bag nodes. $\Box$

4.4. An execution model for Squirrel mediators

Together this and the next subsections present the execution model used by Squirrel mediators generated by the current Squirrel generator. The execution model is called the "bottom-up VDP-based execution model" (BV execution model). This is just one of several possible and reasonably efficient execution models; a topic of continuing research is the comparison of alternative execution models. This subsection focuses on how the execution model supports the traditional query operators (selection, projection, join, union, difference), and the following subsection describes how support for object matching is incorporated.

The approach to maintaining integrated views presented here (based on the execution model, VDPs, and rulebases) provides a systematic and comprehensive implementation of an algorithm that follows the spirit of and generalizes the algorithms of [5], [19], [18] for maintaining materialized views over a single database, using the active paradigm as in [8], [9].

As with all active databases, the BV execution model permits a separation of the logic of a Squirrel mediator from the control. As with other active databases, the control aspect of the Squirrel mediator is performed by the execution model. Unlike most other active databases, the logic of a Squirrel mediator is found in two components: the rulebase and the VDP. In essence, the execution model uses the VDP to guide the order of rule application. Furthermore, the rulebase is closely related to the VDP. In general, each rule specifies how updates are propagated along a specific edge of the VDP.

The BV execution model ensures that incremental updates of source data are correctly reflected in the integrated view. The model offers some freedom with regards to the specific order in which rules are fired, thereby offering opportunities for optimization at that level. The BV execution model is currently implemented using Heraclitus[Alg,C].

We now describe several aspects of the BV execution model in some detail, and then present its specification. The execution model uses several notions from the Heraclitus paradigm (see Subsection 2).

A *VDP-rulebase* is a pair $(\mathcal{V}, \mathit{edge\_rule})$, where

(a) $\mathcal{V} = (V, E, \mathit{class}, \mathit{Source}, \mathit{def}, \mathit{Dist})$ is a VDP; and

(b) $\mathit{edge\_rule}$ is a function that maps each edge in E to a rule (or a set of rules, in case the edge is from a leaf node into a match-class node).

A description of the rules, and how they are generated, is given in Subsection 4.6. The following description of the BV execution model provides important context for understanding those rules.

Speaking at an abstract level, incremental updates will arrive in the queue of the Squirrel mediator in a serial but asynchronous manner. We assume that each incremental update arriving to the queue is in the form of a delta against one or more of the classes in a single source database. (A simple optimization would be to let the source databases filter the updates, so that they correspond to deltas against the lowest non-leaf nodes of the VDP.)

Each invocation of the execution model will form a separate transaction. Let the sequence of starting times of these transactions be $t_1, t_2, \ldots$. These are called *execution invocation times*. We require for each i that t_{i+1} is a point in time that is after the completion of the transaction for time t_i. By the phrase "*the state of the source databases at time* t_i" we mean the state of the source databases as reflected by the updates they have reported to the Squirrel mediator up to time t_i.

Suppose that a Squirrel mediator with VDP $\mathcal{V}$ has been deployed. Two repositories are associated with each non-leaf node v of $\mathcal{V}$. Suppose that $class(v) = R$. The first repository is denoted simply as 'R', and holds the "current" population of class R. The second repository is denoted by 'ΔR', and holds the smash of incremental changes for R that result from the incremental propagation of updates during a single execution of the execution model. Recall that classes of bag nodes (i.e., those defined using selection and/or projection and/or join, or union) will hold bags, and set nodes (i.e., those defined using difference or matching) will hold sets.

Let v be a node with $class(v) = R$. By the phrase "*process node* v" we mean to fire all eligible rules in $\{edge\_rule(v', v) \mid (v', v) \in E\}$ (in any order) and then to execute the following steps: (1) $R = apply(R, \Delta R)$; (2) $\Delta R = \emptyset$; In the BV execution model 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.

We now briefly present the BV execution model for VDPs that do not have match classes. First, break all incremental updates held in the queue at a given time t into a set $\Delta R_1, \ldots, \Delta R_k$ of subdeltas that refer to some set $R_1, \ldots, R_k$ of source classes that are associated with leaf nodes $v_1, \ldots, v_k$ (respectively) of the VDP. Then, all eligible rules in $\cup\{edge\_rule(v_j, v_i) \mid i \in [1, k], (v_j, v_i) \in E\}$ are fired, in any order, and all entries in the queue that contributed to the subdeltas are deleted. Finally, each non-leaf node is processed in an order where a node v cannot be processed until all of its children have been processed.

It is straightforward to verify that after completion of the execution of this algorithm for time t, the materialized data in the mediator reflects the state of the source databases at time t.

4.5. Providing support for n-ary matches

The Student/Employee example of Section 3 gave an overview about how binary match classes are supported in Squirrel mediators. This subsection presents the general framework used to support n-ary match classes, and describes how the BV execution model is generalized to accommodate this.

Suppose now that classes $A_1, \ldots, A_n$ from various source databases represent the same or overlapping sets of objects-in-the-world. A Squirrel mediator can support matching of objects from these classes by maintaining a match class $match\_A_1\_\ldots\_A_n$. Each of the source classes will contribute three kinds of attributes to the match class (these sets may overlap):

```
interface match_Stud_Emp {
  string      studName;                       string      empName;
  integer[7]  studID;                         integer[9]  SSN;
  string      major;                          string      divName;
  string      local_address;                  string      address;
  string      perm_address;                                 };
```

Figure 6. The class interface of `match_Stud_Emp`, used by `S_E_Mediator`

identification attributes: These are used to identify objects from source databases. They might be printable attributes known to be keys, or might be immutable OIDs from the source databases (see Section 2). Although OIDs are not technically attributes, we view them as such here.

match attributes: These are the (possibly derived) attributes referred to in the match predicates.

data attributes: These are attributes that are used in distinguished classes or by other intermediate classes in the VDP.

Speaking loosely, the class $match\_A_1\_\ldots\_A_n$ will hold an "outer join" of the underlying source classes, where each object in $match\_A_1\_\ldots\_A_n$ represents a single object-in-the-world. Each element of $match\_A_1\_\ldots\_A_n$ is called a *surrogate* object. A given surrogate object might represent objects from essentially any subset of the associated source database classes.

The interface of the match class `match_Stud_Emp` for the Student/Employee example of Section 3 is shown in Figure 6. The left column of 5 attributes of this class come from the `Student` class; the other 4 attributes in the right column come from the `Employee` class. The identification attributes are `studID` and `SSN`, which are printable keys; the match attributes are `studName`, `local_address`, `perm_address`, `empName`, and `address`; and the data attributes are `studName`, `major`, and `divName`.

The use of a match class such as $match\_A_1\_\ldots\_A_n$ is just one possible way of using materialization to support intricate object matching. Indeed, if there are relatively few matches, then it is possible that the class $match\_A_1\_\ldots\_A_n$ will waste a great deal of space on `nil` values. In the current Squirrel prototype, we allow the underlying physical implementation to optimize the internal representation of match classes.

In its current form, when specifying the match criteria for a family of corresponding classes, the ISL can support the specification of several binary match predicates. (More complex match predicates that simultaneously involve three or more classes may be useful in some applications, but these are not currently supported). Suppose that γ is the conjunction of all of the binary match predicates for classes $A_1, \ldots, A_n$. For objects a_i in A_i and a_j in A_j, we write $a_i \sim_\gamma^{A_i,A_j} a_j$ if a_i and a_j satisfy the match predicate of γ specified for the pair A_i, A_j. There may be complex interaction between the match predicates in a specification γ (see [27]).

The BV execution model presented in Subsection 4.4 above must be modified to accommodate match classes. Speaking intuitively, there are two reasons for this. The first stems from the fact that a match class can have more than one child from the source databases. (No other kind of node in VDPs has this property.) This complicates the initialization step of the execution model, because the incremental updates of all children of a match node must be brought up to the match node. These incremental updates should not be applied to the match class, because one of the guiding philosophies of the BV execution model is that incremental updates to a class are applied only when that class is "processed". However, when bringing incremental updates from one source class into the match class we need to refer to the effect of incremental updates already brought from other source classes. As a result, we use the Heraclitus operator `when` to obtain efficient hypothetical access to a match class and the incremental updates already propagated to it. The second reason that the BV execution model needs to be modified stems from the possibility of complex interaction between the possibly many binary match predicates that contribute to the definition of an n-ary match. Speaking intuitively, rules corresponding to these binary match predicates must be fired repeatedly until a fixpoint is obtained. A more formal presentation of the extended BV execution model is in [27].

4.6. Default VDP and rule templates

VDPs provide very flexible representations of the skeletons of update processing strategies in the mediator, which can be tailored to optimize the support of integrated views. The current Squirrel prototype constructs a reasonably efficient default VDP for a given integrated view. However, the user who created the ISL specification can explicitly modify the default VDP, so that the final Squirrel mediator will be generated according to the revised VDP. (For example, VDP $\mathcal{V}_1$ of Figure 4 and the VDP of Figure 5 are default, and $\mathcal{V}_2$ of Figure 4 is not.)

In this subsection we give a brief overview of the construction of default VDPs for supporting arbitrary ISL specifications. This subsection focuses on constructing "simple VDPs", that support individual class definitions of an ISL specification. As mentioned at the beginning of Subsection 4.3, in this discussion we focus exclusively on constructing mediators whose imported and exported classes are sets.

A *simple VDP* is a VDP that (i) has a single root, that is distinguished; (ii) has only source classes or distinguished nodes as leaves; and (iii) has no distinguished non-leaf, non-root nodes. Intuitively, a simple VDP can be constructed for each distinguished class defined in an ISL specification. When constructing VDPs for a full ISL specification, i.e., for multiple distinguished classes, combine the simple VDPs into a single VDP. In this subsection we describe how we build default simple VDPs for three representative distinguished classes, namely classes defined by difference, select-project-join (SPJ) queries, and object matching. More details about default simple VDPs for all kinds of distinguished classes, and about merging several simple VDPs into a single VDP can be found in [27].

Every edge (a, b) in a VDP is associated with an update propagation rule which computes an incremental update $(\Delta class(a))$ to the class $class(a)$ based on an update $\Delta class(b)$. (More than one rule is generated for each edge from a match class, and additional rules are associated with the match classes.) A family of *rule templates* is used to generate the update propagation rules. Translation of the templates into actual rules uses information about the source database classes, the classes of the Squirrel mediator, and possibly user-defined functions.

We now describe an algorithm for building a simple VDP for an individual class definition in an ISL specification. The algorithm uses an induction on the structure of the expression. We define a subexpression to be *special* if it is a source class, a distinguished class, or if its root operator is union, difference, or match.

There are two base cases in this construction, when the subexpression is simply a source class or simply a distinguished class. In both cases, the VDP for the subexpression has one node, labeled by that class.

There are five inductive cases in the construction; we now consider three of these. (The other two, for union and SPJ are similar.) Importantly, the deltas created by the rules presented below do not include any redundant inserts or deletes.

(1) Difference: Suppose that the subexpression is $T = R_1 - R_2$. Let $\mathcal{V}_i$ be the VDP for R_i $(i = 1, 2)$. The VDP for T is constructed as the union of $\mathcal{V}_1$, $\mathcal{V}_2$, along with an additional node labeled by T, and edges (T, R_1) and (T, R_2).

The rule templates for edges (T, R_1) and (T, R_2) are now presented. Recall that T is a set node. In these templates, $\Delta' R_i$ denotes the net change (as a set-based delta) between R_i, considered as a set, and $apply(R_i, \Delta R_i)$, considered as a set. Also, the operands for $-$ and $\cap$ are interpreted as sets.

```
rule template for diff1: edge $(T, R_1)$
ON      new $\Delta' R_1$
IF      true
THEN    $(\Delta T)^+ = (\Delta' R_1)^+ - R_2$; $(\Delta T)^- = (\Delta' R_1)^- \cap R_2$;
```

```
rule template for diff2: edge $(T, R_2)$
ON      new $\Delta' R_2$
IF      true
THEN    $(\Delta T)^+ = (\Delta' R_2)^- \cap R_1$; $(\Delta T)^- = (\Delta' R_2)^+ \cap R_1$;
```

(2) SPJ: Suppose now that the subexpression is T, which is non-trivial and does not have union or difference as root. Suppose further that $T = \epsilon(R_1, \ldots, R_n)$ where ϵ is an operator involving at least one join, along with zero or more selections and projections, and each R_i is a special node. Again generalizing a well-known normalization result, the expression $\epsilon(R_1, \ldots, R_n)$ can be normalized into the form: $T = \pi_p \sigma_f (R_1 \bowtie_{g_1} \ldots \bowtie_{g_{n-1}} R_n)$, where p is a subset of attributes of the join, f is a selection predicate, and each of $g_1, \ldots, g_{n-1}$ is a join condition,

In order to construct a default VDP for T we introduce n *pre-classes*, one each for $R_1, \ldots, R_n$. Intuitively, the pre-class R_i' for R_i will be a selection and projection of R_i, that includes all attributes needed for T, and includes only those tuples of R_i that will impact the join. More precisely, when constructing the pre-class R'_i for R_i we incorporate the set p_i of all attributes of R_i that are referred to in the join conditions $g_1, \ldots, g_{n-1}$, the selection condition f, or the projection list p. Also for each i we let f_i be a selection condition implied by f relevant to R_i. (We do not insist that f_i captures all of the restrictions that f makes on R_i; if f is complex, that might be inconvenient to compute.) The i-th pre-class is now defined as: $R'_i = \pi_{p_i} \sigma_{f_i} R_i$. Finally, since some of the arguments to the join may be projections of the original arguments, we may need to modify the join conditions $g_1, \ldots, g_{n-1}$ into corresponding conditions $g'_1, \ldots, g'_{n-1}$. (The VDP $\mathcal{V}_2$ in Figure 4(b) in Example 1 illustrates this construction.)

The VDP for T is now constructed from the VDPs for the R'_i's (constructed as in the previous case), along with the node T and edges (T, R'_i) for $i \in [1..n]$. The rule template for the edge (T, R'_i) is (using the bag semantics):

rule template for SPJ: edge (T, R'_i)
ON new $\Delta R'_i$
IF true
THEN $(\Delta T)^+ = \pi_p \sigma_f (R'_1 \bowtie_{g'_1} \ldots \bowtie_{g'_{i-1}} (\Delta R'_i)^+ \bowtie_{g'_i} \ldots \bowtie_{g'_{n-1}} R'_n)$;
$(\Delta T)^- = \pi_p \sigma_f (R'_1 \bowtie_{g'_1} \ldots \bowtie_{g'_{i-1}} (\Delta R'_i)^- \bowtie_{g'_i} \ldots \bowtie_{g'_{n-1}} R'_n)$;

(3) Match: A default VDP for an n-ary match involving source classes $A_1, \ldots, A_n$ consists of a node for $match\_A_1\_\ldots\_A_n$ with an edge from $match\_A_1\_\ldots\_A_n$ to each of the A_i's. (Figure 5 illustrates a binary matching.)

We present here the rule templates for supporting a match node that concern creation of objects for a source class A_i. The two rule templates presented here would be used to generate the rules `R1` and `R2` informally described in Section 3. The modification updates indicated in the second rule action is shorthand for a deletion followed by an insertion. Although the rules generated from the templates described here refer to individual objects, the execution model apply the rules in a set-at-a-time fashion.

rule template for match_edge insertion: edge $(match\_A_1\_\ldots\_A_n, A_i)$
ON new ΔA_i
IF (insert $A_i(x : a_1, \ldots, a_n)$) in ΔA_i
THEN [insert $match\_A_1\_\ldots\_A_n(new : \ldots, nil, x.m_1, \ldots, x.m_j, nil, \ldots)$];
where $m_1, \ldots, m_j$ are attributes contributed to $match\_A_1\_\ldots\_A_n$ by A_i.

Description: if a new object x of class A_i is inserted, insert a corresponding new object into class $match\_A_1\_\ldots\_A_n$, with nil for non-A_i attributes.

rule template for match_node insertion: node $(match\_A_1\_\ldots\_A_n)$
ON insert $match\_A_1\_\ldots\_A_n(x : \ldots, nil, x.m_1, \ldots, x.m_j, nil, \ldots)$
IF exists a unique y in $match\_A_1\_\ldots\_A_n$ such that $match(x, y)$
THEN [delete $match\_A_1 \ldots\_A_n(x)$;
modify $match\_A_1\_\ldots\_A_n(y : existing\ attr.\ of\ y, x.m_1, \ldots, x.m_j, \ldots)$];

Description: when a new $match\_A_1\_\ldots\_A_n$ object x is inserted, if an object y of class $match\_A_1\_\ldots\_A_n$ matches x, delete x and modify y by setting the values of attributes $m_1, \ldots, m_j$ to $x.m_1, \ldots, x.m_j$.

Analogous rule templates for deletions of source objects are also included. The match_edge deletion rule template is a bit more complex than for insertion. Recall that a surrogate object in $match\_A_1\_\ldots\_A_n$ may correspond to one or more *source objects*. If a source object a_i with surrogate object x is deleted, then the match_edge deletion rule will delete x from $match\_A_1\_\ldots\_A_n$, and then insert new surrogate objects into $match\_A_1\_\ldots\_A_n$, one corresponding to each of the source objects (other than a_i) that x corresponded to. (Note that the match_node insertion rules may now recombine some of these newly inserted surrogate objects.) In this manner, information inferred from the presence of a_i will not be retained in $match\_A_1\_\ldots\_A_n$ after a_i has been deleted from the source database.

5. A Taxonomy of the Solution Space for Data Integration

In its current form, the Squirrel system can be used to generate a rather narrow class of mediators, that assume the underlying data is stored in a relational or restricted object-oriented form, that are based exclusively on the materialized approach, etc. In this section we provide a survey of additional possibilities for supporting read-only integrated views, that covers both different aspects of the underlying application environment, and different approaches to supporting the view. While the survey does include the virtual as well as the materialized approach, more emphasis is placed on the materialized approach. The survey is presented in the form of a taxonomy, which is summarized in Table 1 at the end of this section.

Our taxonomy is based on seven spectra. The first four spectra are relevant to all solutions for data integration; these are (1) Data model heterogeneity, (2) Expressiveness of the integration language, (3) Object matching criteria, and (4) Materialized vs. virtual. The other three spectra are relevant to solutions that involve materialization; these are (5) Activeness of the source databases, (6) Maintenance strategies, and (7) Maintenance timing. We feel that these spectra cover the most important design choices that must be addressed when solving a data integration problem. In the discussion below we have identified within each spectra what we believe to be the most important points, relative to the kinds of data integration problems and environments that arise in practice. While the spectra are not completely orthogonal, each is focused on a distinct aspect of the problem.

A primary motivation for developing the taxonomy is to aid in the development of modular implementations for a broad array of mediators that support integrated views. As just one example, the taxonomy suggests that the implementation strategy used for incremental update can be largely independent from the choice and implementation of maintenance timing. Such modularity facilitates the reusability and maintainability of different components of mediators. We expect to use the taxonomy when choosing future extensions of Squirrel.

We now describe each of the seven spectra in turn.

Data model heterogeneity: This spectrum concerns the kind of data model that is used by the underlying data sources. The primary possibilities include files, legacy and *ad hoc* models, the relational model, and object-oriented database models. Constructs for modeling temporal, geographic, manufacturing and other specialized kinds of information also arise. To construct an integrated view across different data models, some data restructuring will be necessary. This may also be necessary if one or more of the underlying models is different from the data model used by the integration mediator. The different data models will generally entail different access languages; integration across multiple models will thus require language translation or wrapping.

The current Squirrel prototype assumes that both the source databases and view for export are represented in the relational or (ODMG) object-oriented database model.

Expressiveness: This spectrum concerns the expressive power of the language(s) used to specify integrated views. One aspect of this spectrum concerns the expressive power in terms of conventional query languages. In terms of the relational data model, some possibilities here include the relatively simple conjunctive queries (in other words, algebra expressions built up from selection, projection and join); these extended using negation (i.e., the relational algebra), or with aggregation, or with both; and the inclusion of recursion [1]. A somewhat orthogonal aspect concerns whether intricate object matching criteria are supported. Another orthogonal aspect is whether explicit constructs are provided in the language for temporal, geographical, and other specialized kinds of information.

A related aspect of the expressiveness spectrum concerns whether the integrated view can monitor conditions across multiple information sources, and if so, how expressive the language for specifying the conditions is.

Squirrel-generated mediators can support integrated views and monitor conditions expressed using a subset of ODMG's OQL that has roughly the expressive power of the relational algebra, extended with object matching capabilities.

Object Matching Criteria: In some cases the problem of identifying corresponding pairs of objects from different databases can be straightforward; in other cases this can be quite intricate or even impossible. We mention some key points from the spectrum, combinations and variations of these can also arise:

Key-based matching is the most straightforward one; it relies on the equality of keys of two objects to match them. WorldBase [26] and SIMS [3] are two examples using this approach. A generalization of this is to permit keys that involve derived attributes.

Lookup-table-based matching uses a lookup-table that holds pairs of immutable OIDs or keys of corresponding objects. References [26] and [21] support look-up tables.

Comparison-based matching provides in addition the possibility of comparing (possibly derived) attributes of two objects, either with arithmetic and logic comparisons or user-defined functions that take the attributes as arguments and return a boolean value, such as the function `close_names()` in the rule `R2` of the Student/Employee example.

Historical-based matching can be used to supplement other matching methods. For instance, an application can specify that two already matching objects stay matched, even if they cease to satisfy the other matching conditions.

The current Squirrel prototype supports all of the kinds of matching criteria mentioned above.

As an aside, we note that in the Student/Employee example, the `Student` class and the `Employee` class refer to the same kinds of objects in the world, namely, people. In the terminology of [9], [10], two entity classes from different databases that refer to the same or overlapping domains of underlying objects are called *congruent* classes. In some cases objects from non-congruent classes may be closely related. For example, one database might hold an entity class for individual flights of an airline, while another database might hold an entity class for "routes" or "edges" (connecting one city to another) for which service is available. The current Squirrel prototype focuses exclusively on matching objects from congruent entity classes.

Materialized vs. virtual: This spectrum concerns the approach taken by an integration mediator for physically storing the data held in its integrated view. The choices include

fully materialized approach, as presented in the current paper, which materializes in the persistent store of the mediator all information relevant to the integrated view and maintenance of it;

hybrid approach that materializes only part of the relevant information; and

fully virtual approach, as presented in [3], [15], which uses query pre-processing and query shipping to answer queries that are made against the integrated view.

The current Squirrel prototype focuses exclusively on the fully materialized approach. Reference [2] describes a system in which integrated views are primarily virtual, but some match information is materialized. Reference [29] describes a different kind of hybrid, in which the integrated view is materialized, but the source databases must be polled when incorporating new updates.

Activeness of Source Databases: This spectrum concerns the active capabilities of source databases, and is relevant only if some materialization occurs. This spectrum allows for both new and legacy databases. The three most important points along this spectrum represent three levels of activeness.

Sufficient activeness: A source database has this property if it is able to send deltas corresponding to the net effect of all updates since the previous transmission, with triggering based either on physical events or state changes.

Restricted activeness: A source database has this property if it cannot send deltas, but it has triggering based on some physical events (e.g., method executions or transaction commits), and the ability to send (possibly very simple) messages to the integration mediator. One useful case of restricted activeness is provided by "asynchronous replication servers". These systems, that are becoming commercially available for relational DBMSs [24], permit one database to hold an exact copy (no selections or projections) of a relation in another database. Another useful possibility here is the case that on a physical event the source database can execute a query and send the results to the integration mediator. Even if the source database can send only more limited messages, such as method calls (with their parameters) that were executed, the mediator may be still able to interpret this information (assuming that encapsulation can be violated).

No activeness: This is the case where a source database has no triggering capabilities. In this case the mediator can periodically poll the source databases and perform partial or complete refreshes of the replicated information.

The current Squirrel prototype is focused on the case of sufficient activeness. It would be relatively straightforward to extend Squirrel to make use of asynchronous replication servers within the restricted activeness case.

Maintenance Strategies: Maintenance strategies are meaningful only if some materialization occurs in the mediator. We consider three alternative maintenance strategies:

local incremental update approach, as presented in the Student/Employee example in Section 3, that stores relevant portions of source data in the mediator so that the incremental maintenance can be performed locally after the source notifies the mediator of relevant updates,

polling-based incremental update approach, as presented in [29], that does not store extra data for the purpose of incremental maintenance, but polls for data as needed from the sources, and

refresh of the out-of-date classes in the mediator by re-generating all their objects.

The current Squirrel prototype is focused on local incremental update.

Maintenance Timing: Maintenance timing concerns when the maintenance process is initiated. Many different kinds of events can be used to trigger the maintenance. Some typical kinds of events include: (i) a transaction commits in a source database, (ii) a query is posed against out-of-date objects in the mediator, (iii) the net change to a source database exceeds a certain threshold, for instance, 5% of the source data, (iv) the mediator explicitly requests update propagation, (v) the computer holding the mediator is reconnected via a network to the source databases, and (vi) a fixed period of time has passed.

The current Squirrel prototype is focused on the first case mentioned above. However, the execution model used by Squirrel is quite independent of the maintenance timing, so other points on this spectrum would be relatively easy to incorporate.

6. Conclusions

This paper presents a framework and prototype tool for generating Squirrel integration mediators, that use materialization to support integrated views over multiple data sources. The paper makes several contributions towards database interoperation. To provide context for research in this area, we (a) present a broad taxonomy that surveys much of the solution space for supporting and maintaining integrated views. At a more concrete level, we (b) introduce "Squirrel mediators"; these are

Table 1. Solution space of the data integration problem

#	Spectra	Range
1	Data model heterogeneity	file, legacy and *ad hoc* models, relational, object-oriented, ...
2	Expressiveness	integr. view (conj. query, neg., ...), obj. match., temporality,
3	Matching criteria	key ↔ lookup-table ↔ comparison ↔ historical ...
4	Materialized vs. virtual	fully materialized ↔ hybrid ↔ fully virtual
5	Activeness of source DB	sufficient activ. ↔ restricted activ. ↔ no activ.
6	Maintenance strategies	local incr. update ↔ polling-based incr. update ↔ refresh
7	Maintenance timing	trans. commit, net change, network reconnect, periodic, ...

a special class of active modules that support incremental maintenance of materialized integrated views in a relatively declarative fashion. Furthermore, (c) we develop a uniform approach for generating integration mediators based on the use of "View Decomposition Plans", and describe (d) the prototype Squirrel generator, which can generate integration mediators automatically. In Squirrel, (e) Squirrel mediators are specified using a high-level Integration Specification Language (ISL). Finally, (f) our framework provides substantial support for intricate object matching criteria.

Primary future directions include developing mediators that support hybrids of the virtual and materialized approaches, and experimentally comparing the relative efficiency of using the virtual, hybrid and materialized approaches.

Acknowledgments

We are grateful to Omar Boucelma, Ti-Pin Chang, Jim Dalrymple, Mike Doherty, and Jean-Claude Franchitti for numerous interesting discussions on topics related to this research.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1994.
2. 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.
3. 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.
4. C. Beeri. Formal models for object oriented databases. In *Proc. of First Intl. Conf. on Deductive and Object-Oriented Databases*, 1989.
5. 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.
6. 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.
7. R.G.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

8. 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.
9. T.-P. Chang. *On Incremental Update Propagation Between Object-Based Databases.* PhD thesis, University of Southern California, Los Angeles, CA, 1994.
10. T.-P. Chang and R. Hull. Using witness generators to support bi-directional update between object-based databases. In *Proc. ACM Symp. on Principles of Database Systems*, pages 196–207, 1995.
11. J. Dalrymple. *Extending Rule Mechanisms for the Construction of Interoperable Systems.* PhD thesis, University of Colorado, Boulder, 1995.
12. M. Doherty, R. Hull, M. Derr, and J. Durand. On detecting conflict between proposed updates. In *Proc. of Intl. Workshop on Database Programming Languages*, September 1995. to appear.
13. M. Doherty, R. Hull, and M. Rupawalla. Structures for manipulating proposed updates in object-oriented databases, 1995. Technical report in preparation.
14. F. Eliassen and R. Karlsen. Interoperability and object identity. *SIGMOD Record*, 20(4):25–29, 1991.
15. D. Florescu, L. Raschid, and P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. In *Proc. of Third Intl. Conf. on Cooperative Information Systems (CoopIS-95)*, Vienna, Austria, May 1995.
16. S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus[Alg,C]: Elevating deltas to be first-class citizens in a database programming language. Technical Report USC-CS-94-581, Computer Science Department, Univ. of Southern California, 1994.
17. S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 441–454, 1993.
18. 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.
19. 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.
20. 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.
21. W. Kent, R. Ahmed, J. Albert, and M. Ketabchi. Object identification in multidatabase systems. In D. Hsiao, E. Neuhold, and R. Sacks-Davis, editors, *Interoperable Database Systems (DS-5) (A-25)*. Elsevier Science Publishers B. V. (North-Holland), 1993.
22. W. Litwin, L. Mark, and N. Roussopolos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.
23. J.J. Lu, G. Moerkotte, J. Schue, and V.S. Subrahmanian. Efficient maintenance of materialized mediated views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 340–351, 1995.
24. D. Stacey. Replication: DB2, Oracle, or Sybase? *Database Programming and Design*, December 1994.
25. G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22(3):237–266, September 1990.
26. S. Widjojo, R. Hull, and D. Wile. Distributed Information Sharing using WorldBase. *IEEE Office Knowledge Engineering*, 3(2):17–26, August 1989.
27. G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization. Technical report, Computer Science Department, University of Colorado, October 1995.
28. 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.
29. 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.