InterBase^{KB}: Integrating a Knowledge Base System with a Multidatabase System for Data Warehousing

[†]Nick Bassiliades, [†]Ioannis Vlahavas[*], [‡]Ahmed K. Elmagarmid[**] and [‡]Elias N. Houstis

[†]Dept. of Informatics
Aristotle University of Thessaloniki,
54006 Thessaloniki, Greece.
E-mail: {nbassili,vlahavas}@csd.auth.gr

[‡]Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
E-mail: {ake,enh}@cs.purdue.edu

[*] This work was carried out while the author was on sabbatical leave at Purdue University.

[**] Partially supported by National Science Foundation under grants 9972883-EIA, 9974255-IIS and 9983248-EIA.

Abstract

This paper describes the integration of a multidatabase system and a knowledge-base system to support the data-integration component of a Data Warehouse. The multidatabase system integrates various component databases with a common query language, however it does not provide capability for schema integration and other utilities necessary for Data Warehousing. The knowledge base system offers in addition a declarative logic language with second-order syntax but first-order semantics for integrating the schemes of the data sources into the warehouse and for defining complex, recursively defined materialized views. Furthermore, deductive rules are also used for cleaning, checking the integrity and summarizing the data imported into the Data Warehouse. The Knowledge Base System features an efficient incremental view maintenance mechanism that is used for refreshing the Data Warehouse, without querying the data sources.

Keywords: Multidatabase, Schema Integration, Data Warehouse, Materialized View, Knowledge Base System, Deductive Rule, and Active Rule

1. Introduction

A *Data Warehouse* is a repository that integrates information from multiple data sources, which may or may not be heterogeneous and makes them available for decision support querying and analysis [13]. There are two main advantages to data warehouses. First, they off-load decision support applications from the original, possibly on-line transaction, database systems. Second, they bring together information from multiple sources, thus providing a consistent database source for decision support queries.

Data warehouses store *materialized views* in order to provide fast and uniform access to information that is integrated from several distributed data sources. The warehouse provides a different way of looking at the data than the databases being integrated. Materialized views collect data from databases into the warehouse, but without copying each database into the warehouse. Queries on the warehouse can then be answered using the views instead of accessing the remote databases. When modification of data occurs on remote databases, they are transmitted to the warehouse. Incremental view maintenance techniques are used to maintain the views consistent with the modifications.

Multidatabase systems are confederations of pre-existing, autonomous and possibly heterogeneous database systems [32]. The pre-existing database systems that participate in the multidatabase are called *local* or *component database systems*. Usually the term "multidatabase" denotes *nonfederated* systems that integrate various heterogeneous database systems by supplying a common query language for specifying queries and transactions, without a global, integrated schema. On the other hand, *federated* database systems support partial or total integration of the schemata of their component database systems.

From the above definitions, it is evident that there are many similarities between federated databases and data warehouses, and the former can become the infrastructure for building and maintaining the latter. However, before a federated database can be used for such a task, it must be supplied with powerful materialized view definition and maintenance tools in order to:

- keep the data stored at the warehouse consistent with the data modifications of the component databases, and
- provide for the data cleansing, integrity checking and summarizing needs of a data warehouse.

A nonfederated database system can also be used for Data Warehousing if a schema integration mechanism that provides uniform and transparent access to the data of the underlying component databases is used.

In this paper we argue that logic offers the desired power and flexibility to data warehousing and we describe the integration of a nonfederated multidatabase system [29] with a knowledge-base system [7], which provides the data-integration component of a Data Warehouse, fulfilling all the above requirements. It must be noticed that the system described in the paper is a prototype that tests the usefulness of logic in data warehousing

and cannot be considered as an industrial-strength warehouse system, which also requires a query and analysis component to support the information needs of specific end-users [37].

The actual contribution of this work is the integration of a logic language into the InterBase multidatabase [29] so that complex materialized views can be easily and efficiently defined and maintained. Furthermore, logic rules can be used for several data warehousing utilities, such as data cleansing, integrity checking and summarization.

In addition, this paper extends the logic language presented in previous work of ours [5, 6, 7] with a second-order logic syntax (i.e. variables can range over class and attribute names), which is unambiguously translated into first-order logic (i.e. variables can range over only class instances and attribute values). Second - order syntax proves extremely valuable for integrating heterogeneous database schemes and data models.

At the core of the knowledge base system lays an active OODB with metaclasses [6] that supports events and active rules and also integrates declarative rules (deductive and production rules). These features provide:

- Languages and efficient mechanisms for defining self-maintainable complex materialized views;
- Rich structural and behavioral representation capabilities due to the powerful mechanism of metaclasses.

The declarative rules of the knowledge base system offer a flexible, multiple purpose mechanism in the Data Warehouse, for:

- Defining and maintaining a global integrated schema;
- Defining and maintaining relational, recursive and aggregated views;
- Constructing and using data warehousing tools, such as data cleansing, integrity constraint checking, and summarization.

The outline of this paper is as follows: Section 2 presents related work concerning schema integration and materialized view maintenance in Data Warehouses and multidatabases; Section 3 overviews the architecture of the system; Section 4 describes the deductive rule mechanism for defining and maintaining complex views; Section 5 presents the method for translating and integrating the schemata of heterogeneous component database systems; and Section 6 discusses how deductive rules are used to provide for several necessary data-integration utilities for the Data Warehouse, such as data cleansing, integrity checking and summarization. Finally, Section 7 concludes this paper and discusses future work.

2. Related Work

This section briefly surveys several research issues and technologies that have contributed to the development of the InterBase^{KB} framework for data warehousing, namely integration of heterogeneous data and materialization of views. Furthermore, the relationships of these investigations with InterBase^{KB} are indicated. Finally, we briefly discuss the relationship of InterBase^{KB} with our previous research.

2.1 Schema Integration

There are two broad categories for integrating the schemata of individual, heterogeneous databases.

Object-Oriented (common) data model. Most of the existing approaches for schema integration belong to this category [32, 21, 20, 3, 10]. The databases participating in the federation are mapped to a common data model, which most commonly is object-oriented that acts as an "interpreter" among them. Furthermore, a common view that hides the structural differences on the schemas of the heterogeneous databases offers integration transparency. The major problem associated with the approaches in this category is the amount of human participation required for obtaining the mappings between the schemas of the object-oriented common data model and the data models of the heterogeneous databases.

Higher-order logics. The second approach for schema integration involves defining a higher-order language that can express relationships between the meta-information corresponding to the schemata of the individual databases [22, 25]. Thus, a common data model is not required, but the higher-order language plays the role of the data model. The major advantage of this approach is the declarativeness it derives from its logical foundation. The approaches above, however, are targeted towards the integration of heterogeneous relational databases into a multidatabase.

Finally, an interesting approach that falls in between the above two categories is [4], an extensible logic-based meta-model that is able to capture the syntax and semantics of various data models. However, the second-order nature of this approach probably makes impractical an efficient implementation on a real database system; in addition the semantics are quite complex.

Our approach, a combination of the above approaches, is targeted towards the integration of heterogeneous data sources with varying data models through materialized views in a data warehouse. More specifically, we provide a declarative logic-based language with second-order syntax, which is translated (using the metaclass schema of the OODB), into a set of first-order deductive rules. These deductive rules define a common view, which is the global schema for the integrated heterogeneous databases using the default view definition and incremental maintenance mechanism of the system. In this way, we combine the declarativeness and soundness of first-order logic and the flexibility of meta-models and second-order syntax, along with the rich structural and behavioral capabilities of an object-oriented data model. Finally, we efficiently implement the schema integration mechanism using the event-driven mechanisms of an active database.

Another interesting use of logic in multidatabase systems is querying specification through concurrent logic programming languages, such as VPL [23], which provide the specification of dynamic, parallel querying and static, sequential schema integration.

2.2 Maintenance of Materialized Views

Many incremental view maintenance algorithms have been developed, for centralized relational [19, 11, 12], and object-oriented [24, 1] database systems, as well as for distributed systems, such as data warehouses [2, 41, 33, 40]. The main difference of view maintenance between a centralized and a distributed system is that in centralized environments, base and view data are in the same place; therefore, the former are always available for querying in order to maintain the latter consistently. On the other hand, in a data warehouse, the maintenance of the views may require the querying of remote data sources [41], which may be unavailable for various reasons, unless self-maintainability of views is guaranteed [33]. Furthermore, the remote accesses may delay the process of maintenance.

The approach of [11] (and its generalization [12]) uses multiple active rules to incrementally maintain the materialized views or derived data, respectively. Our approach instead translates one deductive rule into a single active rule using a discrimination network matching technique [5, 6]. The main advantages of our approach are a) easier rule maintenance, b) centralised rule selection and execution control, c) straightforward implementation of traditional conflict resolution strategies of KBSs, and d) net effect of events. Furthermore, the performance comparison of our approach with the previous approaches (in [7]) showed that under set-oriented rule execution, which is required for bulk warehouse updates, our approach is considerably faster.

For the maintenance of the derived objects, we use a counting algorithm that is similar to the one described in [19]. However, they use the counting algorithm only for non-recursive views while for the recursive ones, they use a similar algorithm with [12]. In contrast, we use the counting algorithm for recursive views as well since the calculation of derived objects in our system always terminates even for infinite derivation trees [7].

The approaches of [2, 41, 33, 40] for view maintenance in data warehouses follow an approach similar to [11] although they might not use active rules explicitly. However, the functionality is the same. The work presented in [2, 41] is concerned with the maintenance anomalies that may arise when the maintenance algorithms query directly the data sources in order to maintain the views at the warehouse. The approach of [33] eliminates the need to query the data sources by replicating parts of the base data that are absolutely necessary for the maintenance of the views. The replicated base data along with the views are self-maintainable.

Our approach also creates self-maintainable views; therefore, there is no need to be concerned with the maintenance anomalies of [41], when querying the data sources, as in WHIPS [37] or Heraclitus [40]. Compared to other approaches to self-maintainability, we do not put any special effort to infer which base data should be replicated [33, 27] or to calculate “view complements” [26], because all the necessary information is "stored" inside the memories of the two-input events of the discrimination network.

In contrast to almost all the previous approaches in data warehousing, our approach handles also recursively-defined views, heir to using deductive rules and stratification for defining and maintaining them.

2.3 InterBase^{KB} Features

In conclusion, the distinct features of InterBase^{KB} compared to previous approaches to data warehousing and view materialization are the following:

- Combination of the declarativeness of logic and the flexibility of second-order data models and syntax, along with the rich structural and behavioral capabilities of an object-oriented data model for the schema integration of heterogeneous data sources with varying data models.
- Efficient implementation of the schema integration mechanism using the event-driven rules of an active database.
- Efficient, incremental implementation of the view materialization through the translation of deductive rules into one active rule and a discrimination network.
- Termination of the calculation for infinite derivations due to the event-based mechanism.
- Self-maintainability of views taking advantage of the discrimination network and not using source data replication.
- Use of a single rule language and mechanism for various purposes in a Data Warehouse, such as data cleaning, integrity checking, and summarization.

Compared to previous work of ours in multidatabases [29], InterBase^{KB} mainly offers a logic language and an efficient mechanism for defining and self-maintaining complex materialized views. In this way, the multidatabase system is extended to become the data-integration component of a data warehouse, since deductive rules are used for multiple vital purposes of data warehousing. This paper presents new techniques and protocols for exploiting the resources of the multidatabase system for integrating it with the knowledge base system.

Finally, comparing this work with previous works of ours in knowledge base systems [5, 6, 7] InterBase^{KB} offers:

- A second-order logic language with first-order semantics for integrating heterogeneous schemes and data models;
- New uses for production and deductive rules in terms of Data Warehousing, such as data cleansing, integrity checking and summarization.

The rationale for using an active OO knowledge base system to support data warehousing includes the following:

- High-level, deductive rules provide simplicity, declarativeness, and ubiquity in supporting functionality for data warehousing, such as view definition and maintenance, heterogeneous schema integration, data cleansing, integrity checking, and data summarization.
- Object-orientation offers rich data and knowledge modeling capabilities using the powerful and flexible mechanism of metaclasses. Specifically, the use of metaclasses in InterBase^{KB} contributes to the extensibility of the rule system [7] and the metamodeling ability that allows the translation of

second-order rule syntax into first-order semantics (see Section 5.2). Furthermore, we are currently investigating the use of versatile OO data types to implement OLAP multi-dimensionality [39].

- Commercial OODB systems do not offer the flexibility of metaclasses that our core Prolog-based OODB system [31] exhibits; therefore, the latter serves at best our purposes for fast prototyping and experimentation.
- Finally, our object-oriented KBS offers a natural and seamless integration with InterBase* multidatabase, which is also object-oriented.

3. The InterBase^{KB} System

In this section we describe the architecture of the InterBase^{KB} system along with the functionality of each of its subsystems.

The InterBase^{KB} system extends the InterBase* multidatabase [29, 9] with a KB module (KBM) that is responsible for integrating the schema of the component database systems and for running the inference engine that materializes the views of the component databases inside the data warehouse. The overall system architecture is shown in Figure 1. The components of the InterBase^{KB} system are the following:

InterBase^{KB} Server. This server maintains data dictionaries and is responsible for processing InterSQL queries, as in the InterBase* system. Furthermore, it hosts the materialized views of the data warehouse and Knowledge Base Module (KBM) that runs the engine for creating and maintaining those views.

Old InterBase Clients. The old InterBase* clients can still be used with InterBase^{KB} and issue InterSQL [30] queries against the component databases or the materialized views of the data warehouse which are stored inside the InterBase^{KB} server's database.

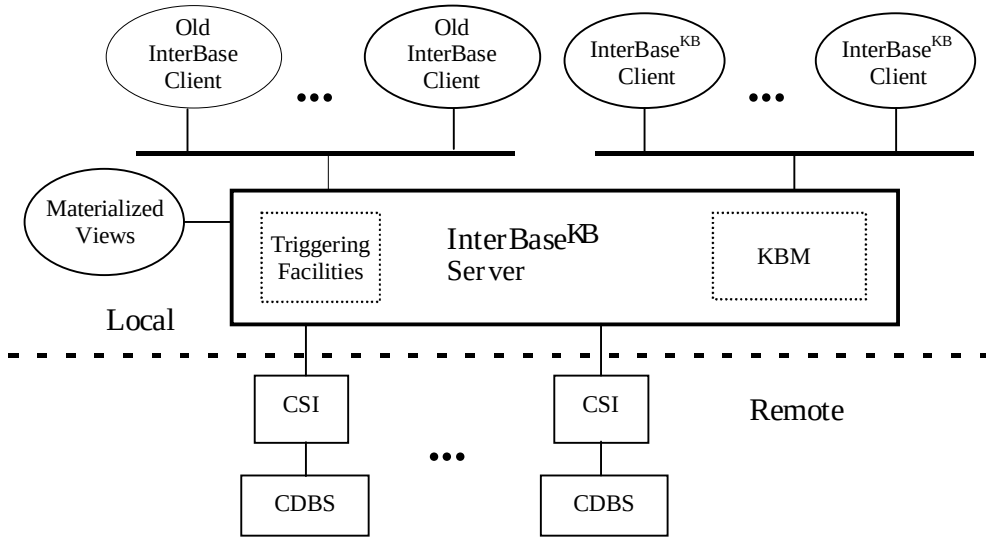

Figure 1. The Architecture of the InterBase^{KB} System

Knowledge Base Module (KBM). This module is part of the InterBase^{KB} server and includes an active OODB, extended with declarative rules and an inference engine for:

- Integrating the schemes of the component databases;
- Defining and maintaining the materialized views of the data warehouse;
- Providing tools for data warehousing, such as data cleansing, integrity checking and summarizing.

InterBase^{KB} Clients. These new clients have to be used in order to access the extended features of InterBase^{KB}, like global integrated schema, updateable materialized views, purely object-oriented database programming language, and declarative rules for programming expert database applications plus advanced data warehousing utilities, such as data cleansing, integrity checking and summarization.

Component Database Systems (CDBSs). These are the heterogeneous systems that are integrated into the multidatabase. Furthermore, they are the data sources for the data warehouse.

Component System Interfaces (CSIs). These components act as an interface for the InterBase^{KB} server to the heterogeneous CDBSs.

3.1 The InterBase^{KB} Server

The InterBase^{KB} server hosts the base data and materialized views of the data warehouse. That means that the users of the data warehouse need not access the source data of the CDBSs but can instead directly access either the base data or the views inside the warehouse. However, the old application programs written for the nonfederated multidatabase system (InterBase*) can still access the source data through the InterBase^{KB} server.

In addition to data warehousing, the InterBase^{KB} server is also used to maintain data for intelligent applications that run on top of the Data Warehouse for Decision Support. These data are read-write and not read-only, as the materialized views are.

The InterBase^{KB} server extends the InterBase* server with triggering capabilities. This means that when an InterBase^{KB} client inserts, deletes or updates data in the InterBase^{KB} server's database, an event is raised that signals the occurrence of such a data modification action. This event is communicated to the KBM and possibly triggers some active or declarative rule.

On the other hand, modifications to the data of the CDBSs are not captured by the triggering system of the InterBase^{KB} server but are handled by the CSIs as explained later, in Section 3.3. However, the changes that are detected at the CSIs level are propagated to the triggering subsystem of the InterBase^{KB} server, which is responsible for delegating it to the KBM for further processing.

3.2 InterBase Clients

The *old InterBase clients* are the clients of the nonfederated multidatabase system (InterBase*). They have been kept in InterBase^{KB} to support the old applications. They connect to the InterBase^{KB} server and issue InterSQL queries against the component databases or the materialized views of the data warehouse which are

stored inside the InterBase^{KB} server's database. They cannot be connected to the KBM because InterSQL cannot be translated to the fully object-oriented programming language of the KBM.

The *InterBase^{KB} clients* are simple clients that accept user queries interactively or user programs in batch mode and forward them through the network to the KBM. The language used is Prolog extended with OO and persistence features, like OIDs, messages, etc. Notice that currently user cannot use SQL through the InterBase^{KB} clients.

3.3 The Component System Interfaces

The CSIs act as an interface between the CDBSs and the InterBase^{KB} server. They translate InterSQL queries and commands to the native query language of the CDBS, and translate back the results, therefore they are specific for each different type of CDBS. While this is adequate for InterBase*, in InterBase^{KB} it is necessary for the interfaces to be able to detect changes of source data that have occurred inside the CDBSs by their native users and inform InterBase^{KB} (and the KBM subsequently) that the data warehouse views might be inconsistent. It is the task of the KBM to decide and propagate these changes to the InterBase^{KB} server's database. However, it is the responsibility of the interface to detect the changes.

Data changes at the data sources are detected by inspecting periodically the *transaction log files* of the CDBSs to extract “interesting” events. This technique is called “transaction shipping” [13]. CSIs have knowledge of the data that must be monitored for changes, i.e. those data that are used inside the warehouse. The inspection of the transaction log files reveals updates (transaction) of source data that must be propagated to the warehouse in order to keep materialized views consistent. The transactions relative to the warehouse are communicated from the CSIs to the InterBase^{KB} server and the latter is responsible to raise the corresponding events that trigger the view maintenance mechanism.

The communication of source data updates is performed periodically when the data warehouse is off-line, i.e. when it is not used for large decision support queries but runs in a maintenance mode. In this way, the maintenance of materialized data does not clutter the data warehouse during its normal operation.

3.4 The Knowledge Base Module

The *Knowledge Base Module* (KBM) is responsible for integrating the schema of the CDBSs, for running the inference engine that materializes the views of the component databases inside the data warehouse, and for providing the data warehousing utilities (data cleansing, integrity checking, summarization). The architecture of the KBM is shown in Figure 2. The components of the KBM are the following:

The Active Knowledge Base (A-KB) core. The KBM's core is an active object-oriented knowledge base system, called DEVICE [5, 6], which is built on top of the Prolog-based ADAM OODB [31, 18] and supports a) persistent objects, b) extensibility through metaclasses, and c) events and event-driven rules as first-class objects [15, 14]. More details on the A-KB are given later.

The A-KB is responsible for a) integrating the schemes of the component databases, b) defining and maintaining the materialized views of the data warehouse (stored at the InterBase^{KB} server), and c) providing the mechanism of deductive rules for implementing several data warehousing utilities.

The A-KB core communicates with the rest of the InterBase^{KB} system through a number of interface components. The administrator of the warehouse directly communicates with the A-KB core and can evoke methods for creating/destroying, enabling/disabling declarative rules for providing all the uses mentioned in the previous paragraph.

The OO-InterSQL interface. This interface translates the first-order rule definition language of A-KB into relational commands of InterSQL. Furthermore, it is responsible for translating simple object accessing methods into SQL retrieval/modification operations.

The Triggering Interface. This interface is responsible for capturing any data modification events trapped by either the triggering subsystem of the InterBase^{KB} server or the component system interfaces. The latter are not communicated directly to the KBM, but through the triggering subsystem of the InterBase^{KB} server. Therefore, the triggering interface of the KBM needs to capture a single event format. The events raised by the component system interfaces denote changes at the source data while the events raised by InterBase^{KB} server denote changes made by the InterBase^{KB} clients to the application data stored inside the server.

The Storage System. The KBM needs to store data and methods, both for the user and for internal purposes, such as rule and event objects. Discrimination network memories can be stored either at internal storage system for small-scale applications or at the InterBase^{KB} server for larger applications, which is usually

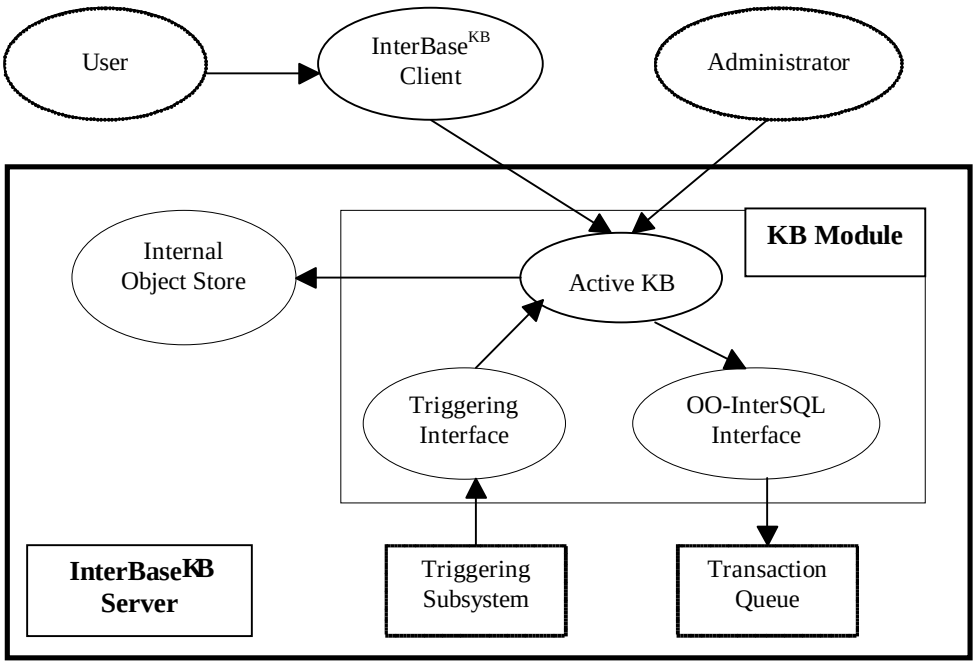

Figure 2. The Architecture of the Knowledge Base Module

the case. ADAM, the underlying OODB system, can support both since persistence is transparent to the upper object layers. The storage system is currently based on the built-in storage facilities of the underlying Prolog system, which is either ECLiPSe or SICStus Prolog. In the future, a more robust commercial storage system may be used.

3.5 The Active Knowledge Base Core

At the heart of the KBM lies the DEVICE system, which is an active knowledge base system built on top of Prolog. DEVICE integrates high-level, declarative rules (namely deductive and production rules) into an active OODB that supports only event-driven rules [14]. This is achieved by translating each high-level rule into one event-driven rule. The condition of the declarative rule compiles down to a complex event network that is used as a discrimination network that incrementally matches the rule conditions against the database.

The rationale for using DEVICE is that the enhanced functionality required for data warehousing can only be provided by a number of high-level features, which cannot be supported by a single rule type. Therefore, multiple rule types are needed for integrating schemata of heterogeneous databases, for defining and maintaining materialized views and for providing several utilities for data warehousing, such as data cleansing, integrity checking and summarization. DEVICE is unique in supporting all kinds of declarative rule types using an efficient event-driven mechanism. Furthermore, the rule mechanism is extensible, allowing more useful rule types to be added in the future [7].

In this section, we briefly describe the syntax and semantics for production rules, which is the basis for integrating other types of declarative rules in the active OODB. Deductive rules and their use for view definition and maintenance are more thoroughly described in Section 4. Furthermore, we discuss the issues concerning rule compilation and condition matching. In Section 5, we present how declarative rules can be used to provide useful tools for data warehousing. Further details about declarative rule integration in DEVICE can be found in previous work of ours [5, 6, 7].

3.5.1 Production Rule Syntax

Production rules mainly follow the OPS5 [16] paradigm, injected with some syntactic influences from the OODB context of DEVICE. Rules are composed of *condition* and *action*, where the *condition* defines a pattern of objects to be detected over the database and *action* defines the set of updates to be performed on the database upon the detection of the pattern occurrence.

Example 1. In the sequel we are going to use the database schema of Figure 3. The following rule example is an integrity constraint that does not allow sales of gun items in United Kingdom before year 1997:

```
IF      I@item(category='Gun') and
        L@line(sale:SL,item=I) and
        SL@sale(store:ST,year<1997) and
```

```
        ST@store(country='United Kingdom')
  THEN  delete => L
```

The condition of a rule is an *inter-object* pattern that consists of the conjunction of one or more (either positive or negative) *intra-object* patterns. The inter-object pattern above denotes the conjunction of instances of classes `item`, `line`, `sale` and `store` that are connected to each other through their object-identifiers, denoted by the variables `I`, `SL`, and `ST`.

The intra-object patterns consist of one or more *attribute* patterns. The first of the above intra-object patterns denotes an instance `I` of class `item` with attribute `category` equal to `'Gun';` the third intra-object pattern describes a sale at store `ST` before year 1997; and so on so forth

Variables in front of class names denote instances of the class. Inside the brackets, attribute patterns are denoted by relational comparisons, either directly with constants or indirectly through variables. Variables are also used to deliver values for comparison to other intra-object patterns (joins) in the same condition or to the action part of the rule. The variables are expressed as valid Prolog variables.

Path expressions inside attribute patterns are also allowed. The condition of Example 1 can be re-written as:

```
L@line(category.name='Gun', country.store.sale='United Kingdom', year.sale<1997)
```

The innermost attribute should be an attribute of class `line`, i.e. the class that the instances of the intra-object pattern stand for. Moving to the left, attributes should belong to classes related through object-reference attributes of the class of their predecessor attributes. We have adopted a right-to-left order of attributes, contrary to the C-like dot notation that is commonly assumed because we would like to stress the functional data model origins of the underlying OODB [18]. Under this interpretation, the chained "dotted" attributes can be seen as function compositions.

During a pre-compilation phase, each rule that contains path expressions is transformed into one that contains only simple attribute expressions by introducing new intra-object patterns. The above pattern is actually transformed into the condition of the rule in Example 1.

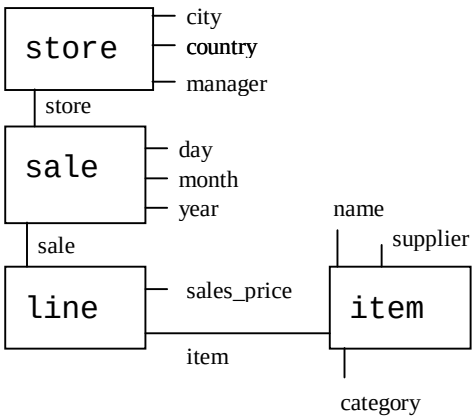

Figure 3. Example Database Schema

Intra-object patterns can also denote classes instead of class instances. For example, the following intra-object pattern "checks" whether the class variable `slot_desc` of the class `personnel` has the value `category`:

```
personnel(slot_desc=category1)
```

The difference between the two notations is just the presence of a variable in front of the class name. During parsing, the above pattern is rewritten as follows:

```
personnel@faculty_A(slot_desc=category1)
```

where `faculty_A` is the metaclass of `personnel`. The new pattern is handled as a normal intra-object pattern. Notice that before the '@' sign a constant instead of a variable is used when the object-identifier of the instance of the meta-class is a known class.

There can also be negated intra-object patterns in the condition. A negated intra-object pattern denotes a negative condition that is satisfied when no objects in the database satisfy the corresponding positive intra-object pattern. For example, the following rule does not allow a sale instance to exist, unless it has at least one line:

```
IF     SL@sale and not L@line(sale=SL)
THEN   delete => SL
```

Notice that the above is not a referential integrity constraint since sale objects are attributes of line objects and not vice-versa.

Only safe rules [34] are allowed, i.e. a) variables that appear in the action must also appear at least once inside a non-negated condition and b) variables that are used inside a negated condition must also appear at least once inside a non-negated condition; otherwise they are just ignored.

The choice for the logic-like condition language is justified by the fact that the condition is supposed to be a declarative specification of the state of the database, and therefore, it is not appropriate to use the procedural interface of the OODB as the condition language. However, the use of arbitrary Prolog or ADAM goals to express some small static conditions or to compute certain values is allowed in the condition through the special `prolog{}` construct.

The action part of the rule is expressed in an extended Prolog language, enriched with the default procedural data manipulation language of ADAM. In the appendix, we include the full syntax of the condition-part language. The syntax of ADAM messages can be found in [18]. We notice here that actions are usually update transactions, which are hardly used in Data Warehouses. However, these examples here illustrate the functionality of production rules, which are the basis for implementing deductive rules, as it will be shown in the following sections.

3.5.2 Production Rule Semantics

When the triggering interface of the KBM raises events to the A-KB core, data updates are propagated throughout the discrimination network and rule conditions are tested for matching against the data sources

incrementally. Normally events are captured during the warehouse-refreshing period, when the CSIs detect changes at data sources and forward them in batch to the InterBase^{KB} server. Alternatively, events may be raised during the execution of user applications that have been written using the rule language of the warehouse.

The warehouse administrator specifies *rule checkpoints*, upon which multiple rules and/or rule instantiations can be eligible for execution (*firing*). A usual checkpoint is the end of the warehouse-refreshing period. Alternatively, checkpoints can be specified by user applications according to the application semantics. The set of all triggered rule instantiations is called the *conflict set*. When the checkpoint event is raised, the production rule manager checks the conflict set and selects only one rule instantiation for execution.

The selection is done according to a customizable rule priority mechanism, which takes into account the order in which the rule instantiations enter the conflict set (recency), their complexity (specificity) and their ability to remain in the conflict set when they have been fired (refractoriness). Furthermore, a user-defined scoring mechanism can be used to order rule execution further.

When a rule is selected for firing, its actions are executed and the possible updates to the materialized views or to the warehouse application data are propagated to the discrimination network. This may result in new rule instantiations to be added into or removed from the conflict set. When all the actions of a rule are executed, a checkpoint is raised again to continue the production cycle until no more rule instantiations exist in the conflict set. After that, the control is given back to the user.

3.5.3 Rule Compilation and Matching

According to [36], there exist ways to translate production rules into ECA rules because they are not really completely different paradigms but rather a different view of the same aspect. In this way, one can embed production rules into an active database system using the primitives of the latter.

Production rules are compiled to ECA rules, in order to be constantly monitored by the active database. The condition of a rule is compiled into a complex event network, which is associated with the event-part of the ECA rule, while the action-part of the ECA rule is the action of the production rule. The compilation method uses complex events to translate a production rule into only one ECA rule. For example, the following (abstract) production rule:

`IF a & b THEN action`,

is translated into the following ECA rule:

`ON` e_a `&` e_b `[IF true] THEN action`,

where e_a, e_b are primitive events that detect the insertion of the data items `a`, `b`, respectively, and the operator `&` denotes event conjunction. Deletions and modifications of data items are also monitored to keep the discrimination network consistent, as discussed below.

The complex event manager of the OODB monitors the above primitive events and combines their parameters in order to detect the occurrence of the complex event incrementally. The parameters of the currently

signaled events are always combined with the parameters of the complete history of event occurrences, which are kept in event memories, in order not to miss any valid rule activation. When the complex event is detected, the condition of the rule has been matched, and the rule manager is responsible for scheduling it to execute.

Notice that the condition part of the ECA rule is always `true` because all conditions tests have been incorporated into the complex event. However, some small static conditions are allowed to be checked in the condition part of the ECA rule through the `prolog{}` construct.

The efficient matching of production rules is usually achieved through a discrimination network, such as RETE [17], TREAT [28], etc. DEVICE smoothly integrates a RETE-like discrimination network into an active OODB system as a set of first class objects by mapping each node of the network onto a complex event object of the active database system.

Each attribute pattern inside any intra-object pattern in the condition is mapped on a *primitive event* that monitors the insertion (or deletion) of values at the corresponding attribute. The insertion of data causes the *signaling* of an insertion event, while the deletion of data causes the *anti-signaling* of a deletion event. In both cases, a token (positive or negative) with the parameters of the triggering event are propagated into the complex event network. The modification of data emits a signal followed by an anti-signal.

Attribute comparisons with constants are mapped onto *logical events*, which perform simple attribute tests, and they are only raised when the associated condition is satisfied.

An intra-object pattern that consists of at least two attribute patterns is translated into a *two-input intra-object event* that joins the parameters of the two input events based on the OID the message recipient objects. Multiple intra-object events are joined in pairs based on the shared variables into *inter-object* events. The last inter-object event of the network maps the whole rule condition, and it is directly attached to the ECA rule that maps the original rule.

Two-input events receive tokens from both inputs whose behavior is symmetrical. The incoming tokens are stored at the input memories and are joined with the tokens of the opposite memory. According to a pre-compiled pattern, the join produces one or more output tokens, which are propagated further to the event network. Stored tokens can be only explicitly deleted by the propagation of anti-signals in the network.

Finally, when the last event in the network signals, it means that the corresponding production rule condition is satisfied, and it must be fired. The rule instantiation token is then forwarded to the rule manager, which stores it in the conflict set. The rest of the procedure has been described in the previous section. On the other hand, when the rule manager receives an anti-signal, it means that a rule instantiation, if it still exists, must be deleted from the conflict set.

4. Deductive Rules for View Definition and Maintenance

Schema integration in multidatabase and heterogeneous environments is usually achieved by defining common views of the underlying data. In this way, details of the heterogeneous data sources are abstracted away, and the user transparently sees a global schema. In this and the following sections, we thoroughly describe the view definition language of InterBase^{KB} along with the techniques for maintaining and updating the views.

In this section, we mainly focus on creating views in an OODB without taking into account the integration of heterogeneous data sources. In the next section, we describe how the basic view definition language is extended to cater for heterogeneous databases to be integrated.

The view definition language of InterBase^{KB} is provided by the deductive rules of the A-KB core, based mainly on Datalog [34]. Deductive rules describe data that should be in the database (intentional DB) provided that some other data and relationships among them hold in the current database state.

4.1 Deductive Rule Syntax

The syntax of deductive rules is very similar to the syntax of production rules (section 3.5.1), especially concerning the condition part, which is identical. The action part of the production rule is replaced by the *derived class template* (DCT), which defines the objects that should be in the database when the condition-part is true.

Example 2. The following deductive rule defines a derived class of the last year sales of toys in Italy:

```
IF      SL@sale(country.store='Italy',year=1998) and
        L@line(sale=SL,item:I,sales_price:SP) and
        I@item(name:N,category='Toy')
THEN    last_year_sales_italy(sale:SL,line:L,item:I,name:N,price:SP)
```

Class `last_year_sales_italy` is a derived class, i.e. a class whose instances are derived from deductive rules. Only one DCT is allowed at the THEN part (head) of a deductive rule. However, there can exist many rules with the same derived class at the head. The final set of derived objects is a union of the objects derived by the multiple rules. The DCT consists of attribute-value pairs where the value can either be a variable that appears in the condition or a constant. The syntax is given in the appendix.

Example 3. The following pair of deductive rules defines a derived class `has_part`, which defines the transitive closure of the part-subpart relation in a part warehouse:

```
DR1:    IF      P@part(subpart:S)
        THEN    has_part(superpart:P,subpart:S)
DR2:    IF      P@part(subpart:S) and
                HP@has_part(superpart:S,subpart:S1)
        THEN    has_part(superpart:P,subpart:S1)
```

The second rule in the above example is a recursive one, which uses the derived class `has_part` both in the condition and the conclusion.

4.2 Deductive Rule Semantics

Deductive rules are just an abstract way for defining new data in terms of existing or other derived data. The way the derivation is realized depends on the actual mechanism, the intended use of the derived data, and the frequency that the base data that they depend on are modified. In InterBase^{KB} we use deductive rules for defining and maintaining materialized views to be stored and re-used in a data warehouse independently from the data sources.

The semantics of deductive rules are an extension of the semantics of production rules. When the condition of a deductive rule is satisfied, the object that is described by the derived class template is inserted in the database. When base data are modified, the rule's condition may not be any longer satisfied, therefore, changes to base data are propagated to the derived data. When a condition becomes false, the object that has been inserted in the database should be deleted. A counter mechanism is used to store the number of derivations for each derived object [19]. In this way, it can be checked whether the object to be deleted is still deducible by another rule instantiation.

Other possible semantics for deductive rules are goal driven rules, which are activated when a query on the derived data is made. Deductive rules are then used to derive all the deducible objects; after the query is answered derived data do not persist. The set of derivable objects can be created using forward chaining techniques (like semi-naive evaluation, magic sets, etc.) or backward chaining, in the fashion of Prolog.

The A-KB core supports rules with such semantics, but in InterBase^{KB} we only use the materialized deductive rules, which are most useful for data warehousing.

4.3 View Materialization and Maintenance

Deductive rules are implemented on top of production rules by extending the simple semantics of the latter, which are not adequate for capturing the semantics of the more high-level deductive rules. This inadequacy can be illustrated by the creation of a newly derived object; a derived object should be inserted in the database only if it does not already exist, otherwise, two distinct objects with the same attribute values will exist. This is a consequence of the generic differences between the OID-based OODBs and the value-based deductive databases [35].

Furthermore, when the condition of a previously verified deductive rule becomes false, the derived object of the head must be removed from the database. Before this is done, however, it must be ensured that the derived object is not deducible by another rule instantiation. For this reason, we use a counter mechanism, which stores the number of derivations of an object [19]. If the derived object has a counter equal to 1, it is deleted; otherwise the counter just decreases by 1.

The above operational semantics of deductive rules can be modeled by the following production rules:

```
IF condition and exists(object) THEN inc_counter(object)
```

```
IF condition and not(exists(object)) THEN create(object)
IF counter(object)>1 and not(condition) THEN dec_counter(object)
IF counter(object)=1 and not(condition) THEN delete(object)
```

In order to integrate the semantics of the above 4 rules into a single (extended) production rule, simple production rules are extend with an `anti_action` (or ELSE) part that hosts the derived object deletion algorithm. Using this extended scheme, a deductive rule can be modeled by a single production rule:

```
IF      condition
THEN    (if exists(object) then inc_counter(object) else create(object))
ELSE    (if counter(object)>1 then dec_counter(object) else delete(object))
```

Furthermore, the rule manager is extended to execute the anti-action upon the receipt of an anti-signal. Notice that the nested *if-then-else* constructs in the above rule are not production rules but the usual conditional primitive of the host programming language (Prolog).

The *conflict resolution* strategies of deductive rules also differ from production rules. The rule search space is navigated in a breadth-first or iterated strategy to model the set-oriented semi-naive evaluation of deductive rules [34], instead of the depth-first (recency) navigation of production rules. The execution order of rules with negation is determined by *stratification*, using the algorithm of [34].

The above deductive rule execution algorithms, combined with the incremental condition checking techniques and fixpoint semantics for production rules that have been described in Section 3.5, provide the mechanism for materializing and correctly maintaining the views over the data sources inside the data warehouse.

4.3.1 Self-maintainable Views

An important feature of a data warehouse is the ability to self-maintain the materialized views, i.e. to be able to modify a view without querying the data sources, based solely on the data kept inside the warehouse and the current updates propagated from the data sources [33].

One of the most important advantages of InterBase^{KB} is that views are self-maintainable (as in [33]), but without using any additional mechanisms due to the existence of the discrimination network. All the data needed in order to derive what changes need to be made to the materialized views in the warehouse when source data are modified can be directly found inside the memories of the two-input complex events, without the need to query back the data sources [41]. This is an additional benefit of using a discrimination network instead of using multiple active rules.

The alternative technique that uses multiple active rules for a single deductive rule [11, 12] needs to query the data sources in order to derive the changes that need to be made to the view. This results in several update anomalies [41] that can occur, in addition to the delay or unavailability in accessing the data sources.

To be more specific, consider the following deductive rule:

```
IF a & b THEN c
```

Using multiple active rules this is translated to at least the following 2 active rules:

```
ON insert(a) IF b THEN insert(c)
ON insert(b) IF a THEN insert(c)
```

When item a is inserted in the data source, the first of the above rules needs to query the data source for item b in order to insert item c in the view of the data warehouse. Using a discrimination network, the information that item b exists is already available inside the memory of the equivalent two-input event. Some other approaches use the multiple rule translation scheme and store auxiliary information inside the data warehouse to avoid query the data source [33]. However, this approach is just an emulation of the data discrimination network.

5. Deductive Rules for Integrating Heterogeneous Data

In this section, we describe the mechanisms of InterBase^{KB} for integrating data from heterogeneous data sources into the Data Warehouse. First, the requirements for such a mechanism are outlined. In the following sections, we show how these requirements are fulfilled from extensions to the deductive rule language and semantics.

The main requirements for integrating of heterogeneous data are the following.

Schema translation of the component databases. The various component databases or data sources probably have their own schemata, which might have been expressed in different data models. Therefore, a mechanism is needed to translate the data model of each data source to the common data model of the data warehouse. InterBase^{KB} supports an object-oriented common data model [32], which is rich enough to capture the heterogeneity between the data models of the data sources.

Resolution of schematic and semantic conflicts. After the homogenization of the data models, there is still a need to resolve the conflicts among the schemata of the data sources. There can be many kinds of conflicts among the local schemata [32, 21, 8], such as schematic, semantic, identity, and data conflicts. The mechanism for schema integration should be general enough to be able to resolve most of them.

Integration transparency. After local schemata have been translated into the common data model and a single global schema exists, the users of the data warehouse should not know which data comes from which data source. Instead the system should distribute their requests transparently to the appropriate data source.

Throughout the section, we will use the following example of heterogeneous data sources.

Example 4.

Consider a federation of faculty databases in a university, consisting of databases (either relational or object-oriented) faculty_A, faculty_B and faculty_C, corresponding to each of the three faculties A, B and C. Each database maintains information about the faculty's departments, staff, and the total number of employees per staff category. The schemata of the three databases are shown in Table 1.

Database	***faculty_A***	***faculty_B***
class	personnel	personnel
attributes	dept: deptID category: string no_of_emp: integer	dept: deptID $category_1$: integer $category_2$: integer ... $category_n$: integer

Database	***faculty_C***			
class	$category_1$	$category_2$	...	$category_n$
attributes	dept: deptID no_of_emp: integer	dept: deptID no_of_emp: integer		dept: deptID no_of_emp: integer

Table 1. Schemata of faculty databases

The faculty_A database has one class, called personnel, which has one instance for each department and each category of staff. The database faculty_B also has one class, also called personnel, but staff category names appear as attribute names and the values corresponding to them are the number of employees per category. Finally, faculty_C has as many classes as there are categories and has instances corresponding to each department and the total number of employees for each category.

The heterogeneity of these databases is evident. The concept of staff categories is represented as atomic values in faculty_A, as attributes in faculty_B, and as classes in faculty_C. We assume that the names of the categories are the same in each database, without loss of generality, since it is not difficult to map different names using our deductive rule language. []

5.1 Extensions to the Rule Syntax

In this section, we present the extensions introduced to the deductive rule language in order to cater for the integration of heterogeneous data.

5.1.1 External Schema References

An external relational or OODB schema is translated into InterBase^{KB} as a collection of classes. The schema of the class is the same as the schema of the corresponding external relation or class, concerning the names and types of attributes. A relation/class is imported in InterBase^{KB} using a deductive rule for defining a derived class as a "mirror" of the external entity. The external (base) class is represented in the condition of the rule using the normal rule syntax extended with a reference to the name of the external database.

The personnel class of database faculty_A (Example 4) is imported into InterBase^{KB} as shown in Figure 4. The name of the database from which the relation/class is imported appears just after the name of the class. The interpretation of this reference to an external database will be presented in section 5.2.1.

5.1.2 Second-order Syntax

The derived class `personnel` will be also used to import personnel data from the rest of the faculty databases. However, the import of the other databases cannot be done in such a straightforward manner because the staff categories are either attribute or class names, and a second order syntax is needed. When variables of a deductive rule language can range over attribute or class names, we say that the rule language has a second-order syntax. The databases for `faculty_B` and `faculty_C` are imported as shown in Figure 4.

Rule DB_B has a variable `C` that ranges over all the attributes of the class `personnel` of database `faculty_B`, except attribute `dept`, which is explicitly mentioned in the condition. Rule DB_C has again a variable `C` that ranges over the classes of database `faculty_C`. Despite the second-order syntax, the above rules are interpreted using a set of first-order rules, as it will be described in section 5.2.2.

5.2 Schema Integration

In this section, we describe how the deductive rule language extensions are integrated in the view definition and maintenance mechanism of InterBase^{KB} providing schema integration for heterogeneous data sources.

5.2.1 Importing External Schemata

Each imported database is represented in InterBase^{KB} as a metaclass. This metaclass contains all the necessary information about the imported database, such as its name, type, network address of CSI and CDB, exported relation/classes, communication and/or storage protocols, etc. This information is copied from the system's data directory [29].

Each relation/class that is imported from an external database is an instance of the above metaclass. In this way, the information about the origins of a specific class can be easily traced by following the `is_instance_of` link. Figure 5 shows how the databases and classes of Example 4 have been imported in InterBase^{KB}. It is obvious that the name of the imported database metaclass is constructed by appending the

```
DB_A:   IF     P@personnel/faculty_A(dept:D,category:C,no_of_emp:N)
        THEN   personnel(dept:D,category:C,no_of_emp:N)

DB_B:   IF     P@personnel/faculty_B(dept:D,C\=dept:N)
        THEN   personnel(dept:D,category:C,no_of_emp:N)

DB_C:   IF     P@C/faculty_C(dept:D, no_of_emp:N)
        THEN   personnel(dept:D,category:C,no_of_emp:N)
```

Figure 4. Deductive rules for integrating the schemata of Example 4

`meta_class` keyword to the name of the database while the names of the imported classes are constructed by concatenating the original class/relation name and the name of the database. Of course, the renaming is done automatically by the system. The imported classes are now considered base classes when defining the common view for personnel data.

5.2.2 Translation of the Extended Rule Syntax

In this section, the translation of the extended rule syntax will be described. The most important and difficult is the translation of the second order syntax into equivalent rules with first-order semantics. In general outline, this is achieved by transforming:

- Second-order references for an OODB class into first-order references to the equivalent OODB metaclass;
- Deductive rules that contain second-order syntax into production rules that create first-order deductive rules.

First-order syntax. The translation of rules that contain first-order constructs (e.g. DB'_A) is straightforward. The classes of the external databases are automatically renamed, appending the name of the database.

Second-order syntax. The translation of rules with second-order syntax is based on the following transformations:

- The intra-object (class) patterns of the rule conditions that contain second-order constructs are replaced with patterns of the corresponding metaclasses. The new metaclass patterns match their instances, which are the classes to be "discovered" by the second-order constructs of the original rules.
- The attribute patterns of the original rules are transformed in attribute tests of the `slot_desc`

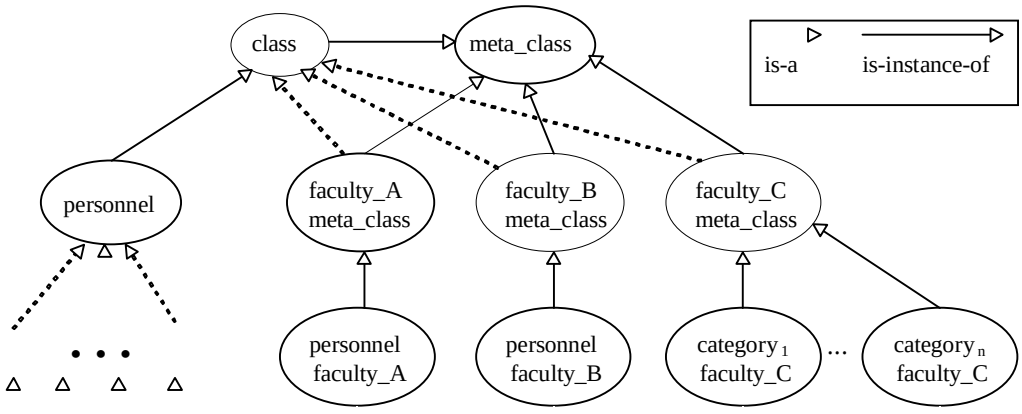

Figure 5. Schema translation and integration for databases and classes of Example 4

attribute of the metaclass. This set-valued attribute is present in every metaclass [18] and contains the description (name, type, cardinality, visibility, etc.) for each attribute of its instances (classes).

There are two cases of second-order syntax, which are treated differently:

- When a variable in the original rule stands for an attribute name (e.g. $\texttt{category}_I$ in rule DB'_B), the condition is directly translated into a metaclass pattern whose `slot_desc` attribute is retrieved and propagated to the rule action. Thus, the variable (`C`) stands now for a value of the `slot_desc` attribute, and the second-order construct is transformed into first-order. In the specific example, the class name (`personnel_faculty_B`) is explicit in the original rule and the instance of the metaclass (`faculty_B_meta_class`) in the metaclass pattern is instantiated.
- When the class name is a variable in the original rule, the same variable is the OID of the instance of the metaclass pattern, in the translated one (e.g. rule DB'_C). In order to match the correct instances of the metaclass, the attributes present in the original condition (`dept`, `no_of_emp`) must be members of the `slot_desc` attribute of the metaclass. (`faculty_C_meta_class`).

After the above transformations of the original rule conditions, production rules are created using the transformed first-order patterns. The action part of the production rules is a method call to create new deductive rules with a first-order syntax. Any variables that appear in the place of attributes or classes have already been instantiated by the condition of the transformed rule with the metaclass pattern. Figure 6 shows the translated deductive rules of Figure 4.

Notice that rule DB'_C contains a call to a Prolog built-in predicate in order to construct the proper name for the $\texttt{category}_i$ classes. Similar calls are also included in the actual implementation for creating the rule strings (e.g. for incorporating the variables), but are omitted here to ease the presentation.

The production rules of Figure 6 are triggered even if they are generated after the creation of the class and

```
DB'A:   IF     P@personnel_faculty_A(dept:D,category:C,no_of_emp:N)
        THEN   personnel(dept:D,category:C,no_of_emp:N)

DB'B:   IF     personnel_faculty_B@faculty_B_meta_class(slot_desc:C\=dept)
        THEN   new_rule('IF P@personnel_faculty_B(dept:D,C:N)
                         THEN personnel(dept:D,category:C,no_of_emp:N)')
               => deductive_rule

DB'C:   IF     C@faculty_C_meta_class(slot_desc⊇[dept,no_of_emp]) and
               prolog{string_concat(C,'_faculty_C',C1)}
        THEN   new_rule('IF P@C1(dept:D, no_of_emp:N)
                         THEN personnel(dept:D,category:C,no_of_emp:N)')
               => deductive_rule
```

Figure 6. Translation of deductive rules of Figure 4

metaclass schema because the A-KB core includes a rule activation phase at the end of rule creation. Furthermore, the A-KB core creates events for every method of the OODB schema, including metaclasses. Rules DB'_B, DB'_C will be fired as many times as the number of categories in the respective databases and the same number of deductive rules will be generated. The new deductive rules will also be activated and fired based on the same mechanism. Rule DB'_A is a deductive rule and it will behave as described in Section 4.

5.2.3 View Management

The view that has been created using the set of deductive rules is managed exactly the same as described in Section 4. The user is unaware of the local schemata, and he/she is supposed to query only the top-level class of the view. Through this derived class, all imported databases have a common schema. This is called integration transparency. This common view can be used by other rules in their condition, and it will be considered as a base class.

The materialization, querying, and maintenance of the view are done in the same way as for the rest of the views. Of course, the materialization of the base data means that all the source data are mirrored inside the data warehouse, which wastes a lot of space. A typical solution [33] for this is to define a common view that is actually used in the condition of another deductive rule, i.e. to project away all the unneeded attributes of the local schemata. This will reduce the space needed for materializing the source data.

When source data are updated, the modifications are propagated to the warehouse and the relevant events that trigger the deductive rules are raised, and the parameters of the modifications are propagated in the discrimination network to keep the materialized views consistent.

6. Deductive Rules as Data Integration Tools for Data Warehousing

This section demonstrates the power of deductive rules for providing several useful data integration tools for data warehousing, such as data cleansing, integrity checking, calculation and summarization. It must be noted that such operations are closely related to resolving heterogeneity at schema integration [38].

6.1 Data Cleansing

It is important that the data in the warehouse be correct, since a Data Warehouse is used for decision-making [13]. However, large volumes of data from multiple sources are involved, therefore there is a high probability of errors and anomalies in the data. Therefore, utilities for detecting and correcting data anomalies can prove very useful.

Data cleansing usually involves the following transformations of the source data:

- Renaming of attributes;

- Removal of attributes;
- Addition of attributes and supply of missing field values;
- Transformation of field values.

6.1.1 Attribute Renaming

Attributes names for the same relations or classes that originate from different data sources may be different. Deductive rules that integrate and translate the schemata of the data sources can be used to homogenize these attribute names by renaming them to a common one. In the sequel we will use the imported classes of Example 4.

The following rule imports class `personnel` from the data source `faculty_A` by renaming the `employees` attribute to `no_of_emp`:

```
IF      P@personnel/faculty_A(dept:D,category:C,employees:N)
THEN    personnel(dept:D,category:C,no_of_emp:N)
```

Notice that a different rule for each data source is required, but all attributes of the same data source can be renamed in a single rule.

6.1.2 Attribute Removal

Different data sources may keep more information for certain entities than needed in the Data Warehouse. In this case the rules that import the data source simply omit from the derived class certain attributes of the imported class. In fact, any attribute of the imported class not explicitly mentioned in the condition of the rule is ignored.

For example, assume that faculty's A policy is to have one contact person per each staff category per each department, while the rest of the faculties do not. Then `faculty_A` data source would have an extra `contact_person` attribute in class `personnel`, which would not be included in the warehouse by a rule similar to the previous example. Actually, this type of transformation does not require any special rule.

6.1.3 Attribute Addition

Some data sources may not keep all the information needed at the warehouse. In this case, the missing information should be filled-in for the imported classes. As an example, assume the opposite situation of the previous case; all faculties but faculty A keep a contact person per each staff category per each department. Then class `personnel` would be imported by a rule like the following:

```
IF      P@personnel/faculty_A(dept:D,category:C,no_of_emp:N)
THEN    personnel(dept:D,category:C,no_of_emp:N,contact_person:'UNKNOWN')
```

Different rules are required for each data source, but all new attributes can be added in a single rule for the same data source.

6.1.4 Field Value Transformation

Different data sources may have the same name for the same attribute but the contents (values) of the attributes may be represented differently. Thus, apart from attribute renaming, value transformation is also needed for different data sources. For example, the following rule transforms the value 'RA' of the attribute category to the value 'Research Assistant':

```
IF      P@personnel/faculty_A(dept:D,category='RA',no_of_emp:N)
THEN    personnel(dept:D,category:'Research Assistant',no_of_emp:N)
```

Multiple such rules are required for each different value that this attribute can take. Furthermore, a different set of rules is required for the translation of values of each attribute. Finally, different rules are required for each data source.

6.2 Data Integrity Checking

The integrity of data to be loaded into the warehouse must be checked both syntactically and semantically. Data that do not conform to integrity rules can either be discarded (filtered-out) or corrected before inserted in the warehouse. Integrity or business rules of individual data sources may not coincide to each other and to the integrity rules of the warehouse, therefore a common ground for the restrictions imposed on the data must be found by the warehouse administrator.

The following example, loads in the warehouse only those `personnel` objects of faculty A data source that have at least one employee per category per department:

```
IF      P@personnel/faculty_A(dept:D,category:C,no_of_emp:N>0)
THEN    personnel(dept:D,category:C,no_of_emp:N)
```

The objects that do not conform to the above rule are simply ignored. When multiple such criteria exist, different rules should be present. Integrity rules can also combine multiple relations/classes of the data source to achieve referential integrity or other semantic constraints.

Objects that violate data can also be corrected and loaded into the warehouse instead of being discarded, depending on the policy of the warehouse. For example, the above integrity rule can be complemented by the following rule that inserts the violating personnel objects into the warehouse with their `no_of_emp` attribute corrected to 1:

```
IF      P@personnel/faculty_A(dept:D,category:C,no_of_emp:N=<0)
THEN    personnel(dept:D,category:C,no_of_emp:1)
```

6.3 Data Calculation and Summarization

One of the most important features of a data warehouse compared to an operational database is that the latter contains raw, everyday and volatile data, while the former stores derived and summarized data to help decision-makers. The discussion for the deductive rule language of KBM so far did not include the case for calculation and maintenance of derived or aggregated data. In this subsection we present the extensions to the deductive rule language and the underlying mechanism to provide for such features.

6.3.1 Data Derivation

Data derivation usually involves the calculation of a new attribute of a class stored in the warehouse in terms of attributes stored in the original classes of the data sources. Notice that the original attributes are not actually included in the warehouse. The main concern about derived attributes is the correct maintenance of their values upon the change of the values of the attributes they depend upon.

The following example calculates the attribute `no_of_emp` for the class personnel of the warehouse from two attributes `male_emp` and `female_emp` of the corresponding class of the `faculty_A` data source:

```
IF      P@personnel/faculty_A(dept:D,category:C,male_emp:M,female_emp:F) and
        prolog{N is M + F}
THEN    personnel(dept:D,category:C,no_of_emp:N)
```

The already established semantics of deductive rules causes the following actions upon the update of the original derivator attributes:

- When both attributes `male_emp` and `female_emp` are present, then the derived attribute is calculated and stored.
- When either of the attributes is deleted then the derived object is deleted as well.
- When either of the attributes is updated then this is emulated by a deletion of the attribute followed by an insertion; therefore, the correct value is stored in the derived attribute.

The above concludes that derived attributes can be supported without extending the semantics of deductive rules.

6.3.2 Data Aggregation

Data aggregation involves the calculation of a derived attribute of a class stored in the warehouse in terms of the equivalent attribute of a set of objects of the data sources that are grouped by a set of attributes (not including the aggregating attribute). In this way a lot of source data is not included in the warehouse but summarized through the aggregate attribute. Since the value of an aggregate attribute is the result of an operation on many objects, the changes that occur at the data sources affect the value of the aggregate attribute that should change accordingly.

```
IF      P@personnel_faculty_A(dept:D,lab:L,category:C,no_of_emp:N)
THEN    (if    P1@personnel(dept:D,category:C,no_of_emp:Prev)
         then  (Next is Prev + N,
                update_no_of_emp([Prev,Next]) ⇒ P1)
         else  create(P1))
ELSE    Next is Prev - N,
        (if    Next > 0
         then  update_no_of_emp([Prev,Next]) ⇒ P1
         else  delete(P1))
```

Figure 7. Aggregate-attribute rule translation.

The following example assumes that the `personnel` class of `faculty_A` data source has one extra attribute `lab` in the schema. Each tuple/object of `personnel` represents the number of employees per category per lab per department. When this class is integrated into the warehouse the extra categorization of employee categories per lab is not required and the total number of employees per category per department (regardless of lab) must be summed.

```
IF      P@personnel/faculty_A(dept:D,lab:L,category:C,no_of_emp:N)
THEN    personnel(dept:D,category:C,no_of_emp:sum(N))
```

In order to maintain the aggregate attribute in the warehouse correctly, the following semantics should be emulated:

- When a new `personnel` object is inserted at the data source, the number of employees of the lab should be added to the value of the aggregate attribute at the warehouse.
  - If the new object introduces into the `personnel` class a new department or employee category then a new personnel object for this department and category must be created.
- When a `personnel` object is deleted from the data source, then the number of employees of the lab should be subtracted from the value of the aggregate attribute at the warehouse.
  - If the deleted object is the last one per category per lab of the corresponding department then the `no_of_emp` attribute gets the 0 value. In this case, we choose to delete the object from the warehouse.
- The update of the number of employees of a lab at the data source is emulated by a deletion followed by an insertion, so that the value of the aggregate attribute at the data warehouse remains consistent.

The above semantics are emulated by the action/anti-action rule of Figure 7 (in pseudo-language). The above discussion is a simplified description of the mechanism of the KBM for aggregate attribute rules, which is actually quite more complicated for the shake of extensibility in terms of user-defined aggregate functions [7].

7. Conclusions and Future Work

Data Warehouses are information repositories that integrate data from possibly heterogeneous data sources and make them available for decision support querying and analysis using materialized views. The latter need to be maintained incrementally in order to be consistent with the changes to the data sources. Multidatabase systems are confederations of pre-existing, autonomous, and possibly heterogeneous database systems.

In this paper, we have presented the integration of a nonfederated multidatabase system with a knowledge-base system (KBS) providing the data-integration component for a Data Warehouse. The multidatabase system integrates various heterogeneous component databases with a common query language, but does not provide schema integration. The KBS provides a declarative logic language, which a) offers schema integration for heterogeneous data sources, b) allows the definition of complex, recursively defined views in the warehouse over the base data of the data sources, and c) provides data-integration utilities for data warehousing, such as data cleansing, integrity checking and summarization. At the core of the KBS lies an active OODB that:

- Supports metaclass hierarchies which allow the customization of schema translation and integration;
- Supports events and event-driven rules;
- Integrates declarative (deductive and production) rules, which allow the materialization and self-maintenance of the complex views;
- Provides second-order rule language extensions, which allow the declarative specification for integrating the schemata of heterogeneous data sources into the warehouse.

The views are self-maintainable in a sense that the data sources need not and are not queried by the incremental view-maintenance mechanism, which uses only the changes to the data sources and the information stored at the discrimination network that selects matching deductive rules. Finally, the performance of the view maintenance mechanism has been studied in [7], where it is compared against other approaches. The comparison shows that our approach is considerably faster under set-oriented rule execution, which is required for maintaining a warehouse using bulk updates.

The described prototype system supports data-integration for a data warehouse; however, it is not yet intended as an industrial-strength warehouse system, which also requires several query and analysis tools to meets the specific end-users' information processing needs.

We are currently investigating the extension of the described system with query and analysis tools (OLAP) by supporting multi-dimensional data (data cubes) in the warehouse. The essence of multi-dimensional data is the use of efficient implementation mechanisms, for storing, indexing, accessing, summarizing and querying the data cubes, along with flexible modeling capabilities. We are working out suitable extensions to the existing aggregation language and mechanism that were described in this paper, combined with the powerful object-oriented data model of the Knowledge Base System. We believe that these will provide enough functionality for On-Line Analytical Processing.

Appendix - Declarative Rule Syntax

```
<rule> ::= if <condition> then <consequence>
<condition> ::= <inter-object-pattern>
<consequence> ::= {<action> | <derived_class_template>}
<inter-object-pattern> ::= <condition-element> ['and' <inter-object-pattern>]
<inter-object-pattern> ::= <inter-object-pattern> 'and' <prolog_cond>
<condition-element> ::= ['not'] <intra-object-pattern>
<intra-object-pattern> ::= [<inst_expr>'@']<class_expr>['('<attr-patterns>')']
<attr-patterns> ::= <attr-pattern>[','<attr-patterns>]
<attr-pattern> ::= <attr-function> {':'<variable> | <predicates> |
                                    ':'<variable> <predicates> }
<predicates> ::= <rel-operator> <value> [{ & | ; } <predicates>]
<predicates> ::= <set-operator> <set>
<rel-operator> ::= = | > | >= | =< | < | \=
<set-operator> ::= ⊂ | ⊆ | ⊃ | ⊇ | ⊄ | ∈ | ∉
<value> ::= <constant> | <variable>
<set> ::= '[' <constant>[,<constant>] ']'
<attr-function> ::= {[<attr-function>'.']<attribute> | <variable> }
<prolog_cond> ::= 'prolog' '{'<prolog_goal>'}'
<action> ::= <prolog_goal>
<derived_class_template> ::= <derived_class>'('<templ-patterns>')'
<templ-patterns> ::= <templ-pattern>
<templ-patterns> ::= <templ-pattern> [',' <templ-patterns>]
<templ-pattern> ::= {<attribute>|<variable>}':'
                     {<value> | <aggregate_function>'('<variable>')'}
<aggregate_function> ::= count | sum | avg | max | min
<inst_expr> ::= {<variable>|<class>}
<class_expr> ::= <class>
<class_expr> ::= <inst_expr>'/'<class>
```

<class> ::= *an existing class or meta-class of the OODB schema*

<derived_class> ::= *an existing derived class or a non-existing base class of the OODB schema*

<attribute> ::= *an existing attribute of the corresponding OODB class*

<prolog_goal> ::= *an arbitrary Prolog/ADAM goal*

<constant> ::= *a valid constant of an OODB simple attribute type*

<variable> ::= *a valid Prolog variable*

References

[1] R. Alhajj, and A. Elnagar, “Incremental materialization of object-oriented views,” *Data and Knowledge Engineering*, **29**(2), 1999, pp. 121-145.

[2] D. Agrawal, A.E. Abbadi, A. Singh, and T. Yurek, "Efficient View Maintenance at Data Warehouses," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, Tucson, Arizona, 1997, pp. 417-427.

[3] R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, and M.C. Shan, "The Pegasus heterogeneous multidatabase system," *IEEE Computer*, Vol. 24, No. 12, 1991, pp. 19-27.

[4] T. Barsalou and D. Gangopadhyay, "M(DM): An open framework for interoperation of multimodel multidatabase systems," *Proc. IEEE Int. Conf. on Data Engineering*, 1992, pp. 218-227.

[5] N. Bassiliades and I. Vlahavas, "DEVICE: Compiling production rules into event-driven rules using complex events," *Information and Software Technology*, Vol. 39, No. 5, 1997, pp. 331-342.

[6] N. Bassiliades and I. Vlahavas, "Processing production rules in DEVICE, an active knowledge base system," *Data & Knowledge Engineering*, Vol. 24, No. 2, pp. 117-155, 1997.

[7] N. Bassiliades, I. Vlahavas, and A. Elmagarmid, "E-DEVICE: An extensible active knowledge base system with multiple rule type support", *IEEE Trans. On Knowledge and Data Engineering*, Vol. 12, No. 5, pp. 824-844, 2000.

[8] C. Batini, M. Lenzerini, and S.B. Navathe, "Comparison of methodologies for database schema integration," *ACM Computing Surveys*, Vol. 18, No. 4, 1986, pp. 323-364.

[9] O. Bukhres, J. Chen, W. Du, and A. Elmagarmid, "InterBase: An Execution Environment for Heterogeneous Software Systems," *IEEE Computer*, Vol. 26, No. 8, 1993, pp. 57-69.

[10] M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas II, J.H. Williams, and E.L. Wimmers, "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach," *Proc. IEEE Workshop on Research Issues in Data Engineering - Distributed Object Management*, Taipei, Taiwan, 1995, pp. 124-131.

[11] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," *Proc. Int. Conf. on Very Large Databases*, Morgan Kaufman, Barcelona, Spain, 1991, pp. 577-589.

[12] S. Ceri and J. Widom, "Deriving incremental production rules for deductive data," *Information Systems*, Vol. 19, No. 6, 1994, pp. 467-490.

[13] S. Chaudhuri and U. Dayal, “An Overview of Data Warehousing and OLAP Technology,” in *SIGMOD Record*, Vol. 26, No. 1, 1997, pp. 65-74.

[14] O. Diaz and A. Jaime, *EXACT: An extensible approach to active object-oriented databases*, tech. report, Dept. of Languages and Information Systems, University of the Basque Country, San Sebastian, Spain, 1994.

[15] O. Diaz, N. Paton, and P.M.D. Gray, "Rule management in object oriented databases: A uniform approach," *Proc. Int. Conf. on Very Large Databases*, Morgan-Kaufman, Barcelona, Spain, 1991, pp. 317-326.

[16] C.L. Forgy, *OPS5 User Manual*, tech. report, Dept. of Computer Science, Carnegie-Mellon University, 1981.

[17] C.L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, Vol. 19, 1982, pp. 17-37.

[18] P.M.D. Gray, K.G. Kulkarni, and N.W. Paton, *Object-Oriented Databases, A Semantic Data Model Approach*, Prentice Hall, London, 1992.

[19] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, "Maintaining views incrementally," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1993, pp. 157-166.

[20] M. Kaul, K. Drosten, and E.J. Neuhold, "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views," *Proc. IEEE Int. Conf. on Data Engineering*, 1990, pp. 2-10.

[21] W. Kim, I. Choi, S. Gala, and M. Scheevel, "On resolving schematic heterogeneity in multidatabase systems," *Distributed and Parallel Databases*, Vol. 1, No. 3, 1993, pp. 251-279.

[22] R. Krishnamurthy, W. Litwin, and W. Kent, "Language features for interoperability of databases with schematic discrepancies," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1991, pp. 40-49.

[23] e. Kuehn, F. Puntigam, and A.K. Elmagarmid, "Multidatabase Transaction and Query Processing in Logic," *Database Transaction Models for Advanced Applications*, A. K. Elmagarmid, Ed. Morgan Kaufmann, 1991, pp. 298-348.

[24] H.A. Kuno, and E.A. Rundensteiner, "Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation," *IEEE Trans. On Knowledge and Data Engineering*, **10**(5), 1998, pp. 768-792.

[25] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian, "On the logical foundation of schema integration and evolution in heterogeneous database systems," *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, Springer-Verlag, Phoenix, Arizona, 1993, pp. 81-100.

[26] D. Laurent, J. Lechtenbörger, N. Spyratos, and G. Vossen, "Complements for Data Warehouses," Int. IEEE Conf. On Data Engineering, Sydney, Australia, 1999, pp. 490-499.

[27] W. Liang, H. Li, H. Wang, M.E. Orlowska, "Making Multiple Views Self-Maintainable in a Data Warehouse," *Data & Knowledge Engineering*, **30**(2), 1999, pp. 121-134.

[28] D.P. Miranker, "TREAT: A better match algorithm for AI production systems," *Proc. AAAI*, 1987, pp. 42-47.

[29] J. Mullen, O. Bukhres, and A. Elmagarmid, "InterBase*: A Multidatabase System," *Object-Oriented Multidatabase Systems*, O. Bukhres and A. K. Elmagarmid, Eds., Prentice Hall, 1995, pp. 652-683.

[30] J.G. Mullen and A. Elmagarmid, "InterSQL: A Multidatabase Transaction Programming Language," *Proc. Workshop on Database Programming Languages*, 1993, pp. 399-416.

[31] N.W. Paton, "ADAM: An object-oriented database system implemented in Prolog," *Proc. British National Conf. on Databases*, CUP, 1989, pp. 147-161.

[32] E. Pitoura, O. Bukhres, and A. Elmagarmid, "Object Orientation in Multidatabase Systems," *ACM Computing Surveys*, Vol. 27, No. 2, 1995, pp. 141-195.

[33] D. Quass, A. Gupta, I.S. Mumick, and J. Widom, "Making Views Self-Maintainable for Data Warehousing," *Proc. Conf. on Parallel and Distributed Information Systems*, Miami, Florida, USA, 1996, pp. 158-169.

[34] J. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, Maryland, 1989.

[35] J. Ullman, "A comparison between deductive and object-oriented database systems," *Proc. Int. Conf. on Deductive and Object-Oriented Databases*, Springer Verlag, Munich, 1991, pp. 263-277.

[36] J. Widom, "Deductive and active databases: Two paradigms or ends of a spectrum?," *Proc. Int. Workshop on Rules in Database Systems*, Springer-Verlag, Edinburgh, Scotland, 1993, pp. 306-315.

[37] J.L. Wiener, H. Gupta, W.J. Labio, Y. Zhuge, H. Garcia-Mollina, and J. Widom, "A System Prototype for Warehouse View Maintenance," in *Proc. ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, 1996, pp. 26-33.

[38] M.-C. Wu and A. Buchmann, "Research Issues in Data Warehousing," in *Proc. BTW 1997*, pp. 61-82.

[39] Y. Zhao, K. Ramasamy, K. Tufte, and J.F. Naughton, "Array-Based Evaluation of Multi-Dimensional Queries in Object-Relational Databases Systems," *Proc. IEEE Int. Conf. on Data Engineering*, 1998, pp. 241-249.

[40] G. Zhou, R. Hull, R. King, J.-C. Franchitti, "Supporting Data Integration and Warehousing Using H20," *IEEE Data Engineering Bulletin*, **18**(2), 1998, pp. 29-40.

[41] Y. Zhuge, H. Garcia-Mollina, J. Hammer, and J. Widom, "View maintenance in a warehousing environment," *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, San Jose, California, 1995, pp. 316-327.