Proc. of 1993 VLDB Conference, pages 108–119

Managing Semantic Heterogeneity with Production Rules and Persistent Queues

Stefano Ceri
Dipartimento di Elettronica
Politecnico di Milano
Piazza L. da Vinci, 32
I-20133 Milano, Italy
ceri@cs.stanford.edu

Jennifer Widom
IBM Almaden Research Center
650 Harry Road, K55/801
San Jose, CA 95120
USA
widom@almaden.ibm.com

Abstract. We show that production rules and persistent queues together provide a convenient mechanism for maintaining consistency in semantically heterogeneous multidatabase environments. We describe a specification language and methods for automatically deriving production rules that maintain (1) existence dependencies, in which the presence of data in one database implies the presence of related data in another, and (2) value dependencies, in which the value of data in one database is based on the value of related data in another. The production rules derived from dependency specifications use persistent queues to monitor and maintain the dependencies automatically, asynchronously, incrementally, and correctly.

1 Introduction

It is quite common for multiple databases to model overlapping portions of the real world. In such cases it is desirable whenever possible to maintain *consistency* across databases, i.e. to ensure that they do not contradict each other with respect to the existence or value of real-world entities. Maintaining consistency is particularly difficult in the presence of *semantic heterogeneity*, which occurs when the databases model the same real-world entities in different ways [3]. Semantic heterogeneity can take the form of, e.g., *naming conflicts*, when different databases use different names to represent the same concepts, *domain conflicts*, when different databases use different values to represent the same concepts, *meta-data conflicts*, when the same concepts are represented at the schema level in one database and at the instance level in another, and *structural conflicts*, when different databases use different data organization to represent the same concepts. These conflicts are based solely on data semantics, so they can arise even when databases share the same data model and query language. In this paper we consider multidatabase environments using the relational model; our methods can be adapted to other models.

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

We show that when a multidatabase environment includes facilities at each site for *production rules* and *persistent queues*, these facilities can be used to maintain consistency across semantically heterogeneous databases. Production rules in database systems (also known as *active database systems*) allow specification of database operations that are executed automatically whenever certain events occur or conditions are met [11]. Persistent queues in multidatabase (or client-server) environments provide a mechanism for reliable execution of asynchronous transactions on remote data [2, 14].[1] We consider multidatabase environments in which each database system includes a production rules facility, and persistent queues are supported for execution of remote transactions from each site. (With the rapid emergence of production rules facilities in database prototypes and products [11] and the presence of recoverable queuing systems in many database products [2], we do not believe that these are exceptionally strong assumptions.) In addition, we assume that sites may read remote data synchronously. Note that our assumptions do not require distributed two-phase commit, which usually is not provided in multidatabase environments.

In a multidatabase environment with production rules and persistent queues, consistency across semantically heterogeneous databases can be maintained automatically as follows: Rules are triggered by any changes to a database that may create inconsistencies. Triggered rules include knowledge of heterogeneous representations so they can determine whether consistency actually is violated (by reading remote data); they also include and implement policies for restoring consistency when violations occur (by modifying local data or queuing transactions to modify remote data). Hence, when rules are specified correctly, no inconsistency between related data can persist.

It has been observed that database production rule programming can be very difficult [1], and developing a correct set of rules to maintain consistency across semantically heterogeneous databases is no exception. We provide a framework that frees the user from the task

[1] Various terms for this and similar mechanisms are used in the literature; we have chosen "persistent queue" more or less arbitrarily.

of developing the rule set; rather, the user specifies the consistency requirements in a high-level declarative language. These specifications are translated automatically into production rules that, once installed, are guaranteed to monitor and maintain consistency efficiently (i.e. incrementally) and correctly.

1.1 Related Work

In [5] we describe a framework for semi-automatically deriving production rules that maintain integrity constraints, where constraints are specified as arbitrary predicates over a single database. The derived rules are triggered by any changes that may violate a constraint; they check if the constraint is indeed violated, and if so they execute user-specified compensating actions that restore the constraint. The consistency problem addressed in this paper can be expressed as the problem of maintaining constraints across databases, and consequently the methods in this paper are based partly on the methods in [5]. However, in this paper we consider a particular class of constraints that is expressive enough to encode most correspondences and conflicts that arise in semantically heterogeneous environments, yet is restrictive enough that rule generation can be fully automatic. Furthermore, in this paper we consider asynchronous (multi-transaction) rather than synchronous (single-transaction) consistency.

In [6] we describe a method for deriving, from a view definition, a set of production rules that maintain a materialization of that view. In certain cases the semantic heterogeneity problem can be expressed as maintaining the consistency of a materialized view that is stored in one database but defined over data in another database. Hence, the methods in this paper also are based partly on results in [6]. Again, however, here we consider a variation on the problem that is particular to semantic heterogeneity, and we consider asynchronous rather than synchronous consistency.

In [8] we consider production rule processing in parallel and distributed database environments. The focus is on tightly coupled homogeneous environments and the goal is to provide mechanisms for distributed rule processing that guarantee equivalence to centralized rule processing, i.e. that provide distribution transparency. Here, the focus instead is on loosely coupled environments where production rules are processed autonomously at each site and distribution transparency is not required. (For further discussion of this paper as related to [8] see Section 6.)

While there has been considerable work in the area of heterogeneous databases [9, 16, 19] and in semantic heterogeneity in particular [3], we have not seen any other proposals for automatically generating active database rules that maintain consistency in the presence of semantic heterogeneity. Our language for specifying consistency requirements is similar to the language proposed in [18]: *dependency specifications* in [18] are based on relational algebra while our language is based on SQL. [18] also suggests a mechanism for specifying *mutual consistency requirements*—flexible limits within which related data must remain consistent. We do not address this issue in our work but it might be a useful extension. The framework in [18] includes user-specified *consistency restoration procedures*, and we depart considerably from this approach: [18] suggests a special-purpose subsystem that will monitor consistency and, as appropriate, invoke user-specified procedures which are assumed to restore consistency. In our approach, monitoring is achieved automatically by an existing mechanism (production rules), the operations for restoring consistency are generated automatically, and correctness is guaranteed without relying on the user.

Our method for consistency maintenance relies on the notion of persistent queues as described in the context of client-server interactions in [2]. In follow-on work [14], persistent queues are used as the basis for a transaction management paradigm called *unilateral commit*, and it is argued that this paradigm is appropriate for heterogeneous multidatabase systems since such systems usually do not provide two-phase commit. Indeed, the same properties that make persistent queues appropriate for such a transaction paradigm make them appropriate in our framework for rule-based consistency maintenance.

One important feature that persistent queues provide in our framework is the ability for a portion of a rule's action to be executed in a separate transaction from the rule's triggering event. Similar capabilities are provided in database rule systems that support *coupling modes*, notably HiPAC [17]. However, the coupling modes suggested in HiPAC and other systems do not accommodate the partly synchronous and partly asynchronous nature of the rule actions generated in our framework. Furthermore, we have found that the properties of transaction execution provided by persistent queues are more suited than decoupled rule actions for consistency maintenance in loosely coupled environments.

Finally, it is important to distinguish our work from approaches in which a rich (e.g., object-oriented or functional) data model is used to provide high-level integration of semantically heterogeneous multidatabases, as in, e.g., [10,15]. In our approach we do not introduce a new integrating model, but rather manage conflicts using the existing data model augmented with production rule and persistent queue capabilities.

1.2 Running Example

We draw examples from a case study conducted jointly by the Politecnico di Milano and ENEL, the Energy Department of Italy. (An initial presentation of this application in the single database setting appears in [5].) Data describing power distribution networks—power supplies, transportation, and use—are stored in multiple relational databases. Some databases model power networks within a particular region, while other databases model networks over large areas. Here we consider just two databases. The first database, which we call $\mathcal{MN}$, supports the design and maintenance of a regional power network for the area of Milano. $\mathcal{MN}$ uses the following (simplified) schema, where the first attribute of each table is a key for that table.

```
plant(plant-id, location, power)
user(user-id, location, power)
node(node-id, location, power-loss)
wire(wire-id, fr, to, type, voltage, power)
wire-type(type, max-voltage, max-power,
          cross-section)
```

Database $\mathcal{MN}$ includes the power produced by each plant, the power required by each user, and the power loss incurred at each intermediate node. There are directed wires between plants, users, and nodes, each of a certain type and carrying a certain voltage and power.

The second database, which we call $\mathcal{IN}$, supports national-level monitoring of the power network throughout Italy. $\mathcal{IN}$ uses the following (simplified) schema, where again the first attribute of each table is a key for that table.

```
node(id, region, location, function, power)
connection(connection-id, fr, to)
wire(wire-id, connection-id, type, voltage,
     power, age)
wire-type(type, max-voltage, max-power, size)
```

Plants, users, and intermediate nodes are represented in the single table **node**. Directional connections between pairs of nodes are specified; wires are associated with connections, and each wire includes its age. Clearly it is desirable if, in this database, the portion of the power network pertaining to the region of Milano is consistent with the power network represented in database $\mathcal{MN}$.

Although this example uses simplified schemata, there are still several sources of semantic heterogeneity, including representatives of each of the four conflict types described in the introduction.

1.3 Outline of Paper

The remainder of the paper proceeds as follows. In Section 2 we describe our language for specifying consistency requirements across databases. A syntax and semantics is given for each construct; a number of examples are included. Section 3 provides necessary preliminaries to rule derivation: Section 3.1 discusses the multidatabase environment we consider, including the persistent queue mechanism; Section 3.2 provides an overview of our database production rule facility—the *Starburst Rule System*. (For concreteness it is necessary to consider a particular rule language, but our techniques could easily be used to generate rules in other languages.) Section 4 describes our method for generating production rules from consistency specifications. In Section 5 we prove the correctness of our approach, showing that the rules are guaranteed to maintain consistency. Finally, in Section 6 we conclude and discuss future work. Due to space limitations, a proof and some examples are omitted; these appear in [7].

2 Dependency Specification Language

Our language for specifying multidatabase consistency requirements is based on two notions of inter-database *dependencies*:

- *Existence dependencies*, in which the presence of data in one database implies that related data should be present in another, and
- *Value dependencies*, in which the value of data in one database should correspond (in some way) to the value of related data in another.

In addition, each existence or value dependency is either:

- *Directional*, so that values in one database are treated as primary while values in the other are treated as secondary, or
- *Nondirectional*, so that values are treated in a symmetric fashion between databases.

Further details and examples below illustrate the distinctions between the four forms of dependencies. As will be seen, our language is expressive enough that dependencies can encode most forms of semantic heterogeneity, easily handling, e.g., the four conflict types described in the introduction. It is important to note that dependency specifications are *declarative*—they specify static facts about consistency across databases and do not in themselves incorporate any notion of how these facts are (procedurally) enforced.

A grammar for specifying dependencies is given in Figure 1. The core of the specification language is based on the **select** statements and predicates of SQL, augmented with user-defined functions (as in, e.g., the query language of *Starburst* [13]). Note that our grammar does not include aggregate functions or subqueries. It is straightforward to add aggregate functions to our framework, though they must be treated separately and cannot always be handled efficiently (i.e. consistency cannot always be checked or restored incrementally). Subqueries complicate the process of determining when consistency may be violated [5]; they also can eliminate the possibility of incremental consistency restoration [6]. Most SQL **select** statements with subqueries can equivalently be expressed without subqueries [4], and we believe that the language given here is sufficiently expressive to specify a wide class of consistency requirements. If subqueries are desired, extending our approach to handle them can use results from [5] and should not be difficult.

Next, we enumerate the four dependency constructs, giving semantics, restrictions, and examples for each.

2.1 Directional Existence Dependencies

As specified by the grammar of Figure 1, the general form of a directional existence dependency is:

$$T.\langle C_1, \ldots, C_n \rangle \leftarrow \textbf{select } X_1, \ldots, X_m \ \textbf{from } T_1, \ldots, T_k \ \textbf{where } P$$

Additional restrictions require that the right-hand-side (RHS) is a semantically well-formed SQL **select** statement, that $m = n$, and that the domain of expression X_i matches the domain of column C_i, $1 \leq i \leq n$.[2] Furthermore, for meaningful maintenance of this dependency

[2] This domain matching requirement does not preclude encoding domain conflicts: X_i can be any SQL expression, including user-defined functions.

Dependency	::=	*Existence-Dependency* $\mid$ *Value-Dependency*
Existence-Dependency	::=	*Directional-ExDep* $\mid$ *Nondirectional-ExDep*
Directional-ExDep	::=	$T.Columns \leftarrow$ *Select-Stmt*
Nondirectional-ExDep	::=	$T_1.Columns_1 \leftrightarrow T_2.Columns_2$
Value-Dependency	::=	*Directional-ValDep* $\mid$ *Nondirectional-ValDep*
Directional-ValDep	::=	$T.Columns \Leftarrow$ *Select-Stmt*
Nondirectional-ValDep	::=	$T_1.Columns_1 \Leftrightarrow T_2.Columns_2$ [**where** *Predicate*]
Columns	::=	$C \mid \langle C_1, C_2, \ldots, C_n \rangle$
Select-Stmt	::=	**select** *Val-Exps* **from** *Table-List* [**where** *Predicate*]
Val-Exps	::=	$* \mid$ $Val\text{-}Exp_1$, $Val\text{-}Exp_2$, ..., $Val\text{-}Exp_n$
Val-Exp	::=	$[T.]\,C$ $\mid$ constant $\mid$ *Function*($Val\text{-}Exp_1$, ..., $Val\text{-}Exp_n$)
Function	::=	$+ \mid - \mid * \mid / \mid$ user-defined function
Table-List	::=	$T_1\ [V_1],\ T_2\ [V_2],\ \ldots,\ T_n\ [V_n]$
Predicate	::=	$Val\text{-}Exp_1$ *Comparison-Op* $Val\text{-}Exp_2$
	$\mid$	$Predicate_1$ **and** $Predicate_2$
	$\mid$	$Predicate_1$ **or** $Predicate_2$
	$\mid$	**not** *Predicate*
	$\mid$	**(** *Predicate* **)**
Comparison-Op	::=	$= \mid \neq \mid < \mid > \mid \leq \mid \geq$

Figure 1: Grammar for specifying dependencies

(see Section 4.1), it is necessary that if T has a key then the key is included in columns $C_1, \ldots, C_n$.

The interpretation of this dependency is:

> For each tuple of values $\langle V_1, \ldots, V_m \rangle$ produced by the RHS **select** statement, there is a tuple t in table T such that t projected on columns $C_1, \ldots, C_n$ has the value $\langle V_1, \ldots, V_m \rangle$.

Existence dependencies are most useful for specifying consistency between exactly two databases: table T resides in one database while tables $T_1, \ldots, T_k$ reside in another. We make this assumption here, but our framework is sufficiently general so that, with some modifications, tables $T_1, \ldots, T_k$ could span multiple databases, or they could reside in the same database as (and even include) table T; see Section 6.

As a simple example, the following directional existence dependency specifies that if there is a node in database $\mathcal{IN}$ with **region** = *Milano* and **function** = *plant*, then that plant also is in database $\mathcal{MN}$. Note that in the specification we qualify each table name with **mn** or **in** to indicate whether it refers to database $\mathcal{MN}$ or $\mathcal{IN}$.

```
mn.plant.plant-id ←
  select id from in.node
  where region = Milano and function = plant
```

2.2 Nondirectional Existence Dependencies

As specified by the grammar of Figure 1, the general form of a nondirectional existence dependency is:

$$T_1.\langle C_1^1, \ldots, C_n^1 \rangle \leftrightarrow T_2.\langle C_1^2, \ldots, C_m^2 \rangle$$

Additional restrictions require that $m = n$ and that the domains of corresponding columns match. It also is necessary that if T_1 has a key it is included in columns $C_1^1, \ldots, C_n^1$ and if T_2 has a key it is included in columns $C_1^2, \ldots, C_m^2$. The interpretation of this dependency is:

> There is a tuple t_1 in T_1 whose projection on columns $C_1^1, \ldots, C_n^1$ has the value $\langle V_1, \ldots, V_n \rangle$ if and only if there is a tuple t_2 in T_2 whose projection on columns $C_1^2, \ldots, C_m^2$ also has the value $\langle V_1, \ldots, V_n \rangle$.

Notice that every nondirectional existence dependency also can be expressed as a pair of directional existence dependencies:

$$T_1.\langle C_1^1, \ldots, C_n^1 \rangle \leftrightarrow T_2.\langle C_1^2, \ldots, C_m^2 \rangle$$

is equivalent to

$$T_1.\langle C_1^1, \ldots, C_n^1 \rangle \leftarrow \textbf{select } C_1^2, \ldots, C_m^2 \textbf{ from } T_2$$
$$T_2.\langle C_1^2, \ldots, C_m^2 \rangle \leftarrow \textbf{select } C_1^1, \ldots, C_n^1 \textbf{ from } T_1$$

We believe that such pairs of nondirectional existence dependencies are frequent and important enough that they merit separate treatment.

As a simple example, the following nondirectional existence dependency specifies that the same set of wire types is represented in databases $\mathcal{MN}$ and $\mathcal{IN}$, with corresponding cross-sections and sizes (but not necessarily with corresponding maximum voltage and power).

```
mn.wire-type.<type,cross-section> ↔
in.wire-type.<type,size>
```

2.3 Directional Value Dependencies

As specified by the grammar of Figure 1, the general form of a directional value dependency is:

$$T.\langle C_1, \ldots, C_n \rangle \Leftarrow \textbf{select } X_1, \ldots, X_m \textbf{ from } T_1, \ldots, T_k \textbf{ where } P$$

The RHS must be a semantically well-formed SQL **select** statement, except that predicate P may refer to columns of table T. We also require that $m = n$ and that the domain of expression X_i matches the domain of column C_i, $1 \leq i \leq n$. For maintenance of this dependency (see Section 4.3), table T must have at least one key, not necessarily included in $C_1, \ldots, C_n$. The interpretation of this dependency is:

> For each tuple t in T, if evaluating the RHS **select** statement using t produces exactly one tuple of values $\langle V_1, \ldots, V_m \rangle$, then t projected on columns $C_1, \ldots, C_n$ has the value $\langle V_1, \ldots, V_m \rangle$.

Typically, predicate P ensures that, for each t in T, at most one tuple is produced by the RHS **select** statement (as in examples below). When no tuples are produced, the dependency is satisfied vacuously, so t may have any value. In the case where multiple tuples are produced,[3] t projected on columns $C_1, \ldots, C_n$ may have the value of any such tuple. Directional value dependencies are most useful for specifying consistency between exactly two databases, i.e. table T resides in one database while tables $T_1, \ldots, T_k$ reside in another, but our framework is sufficiently general that this restriction can be lifted with some modifications.

As a simple example, the following directional value dependency specifies that the voltage of a wire in $\mathcal{IN}$ must be *low* if there is a corresponding wire in $\mathcal{MN}$ with **voltage** < 150. (A similar dependency would specify that the voltage of a wire in $\mathcal{IN}$ must be *high* if there is a corresponding wire in $\mathcal{MN}$ with **voltage** ≥ 150.)

```
in.wire.voltage ⇐
    select low from mn.wire
    where mn.wire.wire-id = in.wire.wire-id
    and mn.wire.voltage < 150
```

2.4 Nondirectional Value Dependencies

As specified by the grammar of Figure 1, the general form of a nondirectional value dependency is:

$$T_1.\langle C_1^1, \ldots, C_n^1 \rangle \Leftrightarrow T_2.\langle C_1^2, \ldots, C_m^2 \rangle \ \textbf{where} \ P$$

Additional restrictions require that $m = n$, that the domains of corresponding columns match, and that predicate P refers only to columns of T_1 and T_2. The interpretation of this dependency is:

> For every pair of tuples t_1 in T_1 and t_2 in T_2, if P is satisfied using t_1 and t_2, then t_1 projected on columns $C_1^1, \ldots, C_n^1$ has the same value as t_2 projected on columns $C_1^2, \ldots, C_m^2$.

Similar to directional value dependencies, predicate P typically ensures that for each t_1 in T_1 there is at most one t_2 in T_2 such that P is satisfied using t_1 and t_2, and vice-versa; see the example below. In the case where a tuple in T_1 satisfies P with more than one tuple from T_2 (or vice-versa), only one of the equalities must hold.[3]

As with existence dependencies, every nondirectional value dependency also can be expressed as a pair of directional value dependencies:

$$T_1.\langle C_1^1, \ldots, C_n^1 \rangle \Leftrightarrow T_2.\langle C_1^2, \ldots, C_m^2 \rangle \ \textbf{where} \ P$$

is equivalent to

$$T_1.\langle C_1^1, \ldots, C_n^1 \rangle \Leftarrow \textbf{select} \ C_1^2, \ldots, C_m^2 \ \textbf{from} \ T_2 \ \textbf{where} \ P$$

$$T_2.\langle C_1^2, \ldots, C_m^2 \rangle \Leftarrow \textbf{select} \ C_1^1, \ldots, C_n^1 \ \textbf{from} \ T_1 \ \textbf{where} \ P$$

Here too we believe that such pairs of nondirectional dependencies are frequent and important enough that they merit separate treatment.

As a simple example of a nondirectional value dependency, suppose that the sets of wire types in $\mathcal{MN}$ and $\mathcal{IN}$ need not have a one-to-one correspondence. The following dependency specifies that, for those wire types appearing in both databases, the **cross-section** in $\mathcal{MN}$ is equal to the **size** in $\mathcal{IN}$.

```
mn.wire-type.cross-section ⇔ in.wire-type.size
 where mn.wire-type.type = in.wire-type.type
```

2.5 Discussion

There is a clear conceptual difference between existence and value dependencies: existence dependencies specify that certain data must exist, while value dependencies specify that if certain data exists then it must have a certain value. However, in the relational model (as opposed to, e.g., some object-oriented models), data exists if and only if it has value. Hence, the reader may have observed that each existence dependency inherently includes a restricted form of value dependency as well. In general, some correspondences may require existence dependencies, some may require value dependencies, and some may require both [7].

As with database integrity constraints [5], it is possible for a set of multidatabase dependencies to be *contradictory*, i.e. certain databases cannot possibly satisfy all dependencies at the same time. As an example of this, consider the following two directional value dependencies. The first is the example from Section 2.3; the second specifies that the voltage of a wire in $\mathcal{IN}$ must be *high* if there is a corresponding wire in $\mathcal{MN}$ with **voltage** > 100.

```
in.wire.voltage ⇐
    select low from mn.wire
    where mn.wire.wire-id = in.wire.wire-id
    and mn.wire.voltage < 150
in.wire.voltage ⇐
    select high from mn.wire
    where mn.wire.wire-id = in.wire.wire-id
    and mn.wire.voltage > 100
```

If there is a wire in $\mathcal{MN}$ with voltage between 100 and 150 and there is a corresponding wire in $\mathcal{IN}$, then according to these dependencies the voltage of the wire in $\mathcal{IN}$ must be *low* and it must be *high*. Contradictory dependencies typically result from erroneous specifications (as in this example), and we suspect that static criteria can be developed to determine, conservatively at least, when a set of dependencies may be contradictory. Currently we allow contradictory dependencies in

[3] This is likely to be a specification error and may be detectable by static analysis.

our framework, and they result in production rules that may trigger each other indefinitely. Fortunately, such infinite triggering behavior can be detected and an error reported; see Section 5.1.

3 Preliminaries to Rule Derivation

Our framework takes dependency specifications in the language of the preceding section and automatically derives production rules that monitor and enforce the dependencies. Before describing rule derivation, it is necessary to explain certain assumptions about the multidatabase environment we consider, particularly with respect to remote operations and persistent queues. In this section we also describe the syntax and semantics of the language we use for derived rules.

3.1 Multidatabase Environment

We consider environments in which relational database systems, presumably containing related data, reside at relatively autonomous distributed sites. Each database system runs transactions consisting of (SQL) operations on local data in the usual way. We also assume there is support for read-only (**select**) operations executed on remote databases in a synchronous fashion, and support for modification (**insert**, **delete**, **update**) operations executed on remote databases in a reliable asynchronous fashion using persistent queues.[4]

For convenience, we denote remote read operations by

$T(C_1, \ldots, C_n) \leftarrow$ **Remote-Select**($\mathcal{DB}$, "$\langle$*SQL-Select*$\rangle$")

where $\mathcal{DB}$ is a (remote) database and $\langle$*SQL-Select*$\rangle$ is an SQL **select** statement. The **select** statement is executed at remote database $\mathcal{DB}$, then the results are returned and assigned to local (temporary) table T with schema $C_1, \ldots, C_n$. We denote remote modification operations by

Enqueue-Insert($\mathcal{DB}$, "$\langle$*SQL-Insert*$\rangle$")
Enqueue-Delete($\mathcal{DB}$, "$\langle$*SQL-Delete*$\rangle$")
Enqueue-Update($\mathcal{DB}$, "$\langle$*SQL-Update*$\rangle$")

where $\langle$*SQL-Insert*$\rangle$, $\langle$*SQL-Delete*$\rangle$, and $\langle$*SQL-Update*$\rangle$ are SQL **insert**, **delete**, and **update** statements, respectively. The **insert**, **delete**, or **update** statement is executed at remote database $\mathcal{DB}$ using the persistent queue mechanism described below; no results are returned. In all four remote operations, the SQL statements may refer to tables of $\mathcal{DB}$; they also may refer to tables at the site issuing the remote operation. In the latter case, copies of local tables referenced in the statement must be transmitted to the remote database along with the statement itself.[5] Numerous examples of remote operations appear in Section 4.

Since we are considering a loosely coupled environment, we assume there is no distributed transaction manager, i.e. there is no support for multi-site transactions, two-phase commit, etc. Hence, we assume that when a remote operation is issued, it is executed in a separate transaction at the remote site. In the case of **Remote-Select**, the remote transaction is executed synchronously with the issuing transaction. That is, suppose a transaction τ at database $\mathcal{DB}_1$ issues a **Remote-Select** operation for database $\mathcal{DB}_2$. The **select** statement is executed in its own transaction τ' at $\mathcal{DB}_2$. Transaction τ at $\mathcal{DB}_1$ suspends until τ' completes at $\mathcal{DB}_2$ and returns its result.

Remote modification operations are executed through persistent queues. Details of persistent queues can be found in [2]; here we give a brief overview sufficient to understand the mechanics and correctness of their use in our approach. Suppose a transaction τ at database $\mathcal{DB}_1$ issues an **Enqueue-Insert**, **Enqueue-Delete**, or **Enqueue-Update** operation for database $\mathcal{DB}_2$. The effect of this is to append a request for the operation to a local persistent queue (call it Q_1) at the site of $\mathcal{DB}_1$, such that the append becomes "valid" if and only if transaction τ commits. Asynchronously, a *queue manager* dequeues valid operation requests from Q_1 and executes these operations in their own transactions at appropriate remote sites, ensuring that each such transaction eventually commits. Hence, **Enqueue-Insert**, **Enqueue-Delete**, or **Enqueue-Update** issued by transaction τ at $\mathcal{DB}_1$ eventually results in successful execution of the **insert**, **delete**, or **update** operation in a separate transaction τ' at database $\mathcal{DB}_2$.

For our method to correctly maintain consistency across databases, at each site remote modification operations must be executed in the same order as they are enqueued, both for multiple enqueues within a single transaction (we assume transactions are sequential) and for multiple enqueues across transactions (where ordering is based on commit time). To ensure this property, we assume that queue managers execute remote operations sequentially using a strict first-in-first-out (FIFO) discipline, and that when transactions enqueue operations they lock the back of the queue to exclude concurrent enqueues from other transactions (concurrent dequeues are allowed). Although we expect that relatively few transactions will need to execute remote operations and consequently lock the local queue, in Section 5.2 we show that queue contention may be reduced if needed. Note that our additional assumptions regarding persistent queues appear quite reasonable in the context of [2].

3.2 Production Rule Language

We provide a brief overview of the *Starburst Rule System* [20], a set-oriented, SQL-based production rule language integrated into the Starburst extensible relational database system. Examples appear in Section 4.

Starburst production rules are based on the notion of *transitions*—database state changes resulting from execution of a sequence of data manipulation operations. Rules consider only the *net effect* of transitions, as defined in [20]. The syntax for creating a rule is:

[4] Such environments also may support asynchronous read-only operations and synchronous modification operations, but such capabilities are not needed by our framework.

[5] Consequently, it is important to reference only small local tables in remote operations, as in our generated rules.

create rule *name* **on** *table*
when *triggering operations*
then *action*

(Rules also contain optional **if** clauses for conditions and optional **precedes** and **follows** clauses for priority ordering, but these are not needed in our framework.) The *triggering operations* are one or more of **inserted**, **deleted**, and **updated**($c_1, \ldots, c_n$), where $c_1, \ldots, c_n$ are columns of the rule's *table*. The *action* specifies an arbitrary sequence of database operations to be executed when the rule is triggered, including SQL commands, assignment to temporary tables, and remote operations as described in Section 3.1.

Rule actions may refer to the current state of the database through top-level or nested SQL **select** operations. In addition, rule actions may refer to *transition tables*, which are logical tables reflecting the changes that have occurred during a rule's triggering transition. At the end of a given transition, transition table **inserted** refers to those tuples of the rule's table that were inserted by the transition. Transition tables **deleted**, **new-updated**, and **old-updated** are similar.

Rules are processed automatically at the commit point of each transaction.[6] The state change resulting from the transaction creates the first relevant transition, and some set of rules are triggered by this transition. A triggered rule R is chosen from this set and R's action is executed. After execution of R's action, all other rules are triggered only if a triggering operation occurred in the composite transition created by the initial transaction and subsequent execution of R's action. Rule R is triggered again only if a triggering operation occurred in the transition created by its action. From the new set of triggered rules, a rule R' is chosen for execution. At an arbitrary time in rule processing, a given rule is triggered if a triggering operation occurred in the (composite) transition since the last time it was executed; if it has not yet been executed, it is triggered if a triggering operation occurred in the transition since the start of the transaction. The values of transition tables in rule actions always reflect the rule's triggering transition. Rule processing terminates when there are no more triggered rules, and the entire transaction then commits.

4 Rule Derivation

Our framework automatically derives several Starburst production rules from each specified dependency. The derived rules are triggered by those database operations that may cause the dependency to be violated. When a rule is triggered, it reads remote data to determine whether the dependency actually is violated and, if so, it restores the dependency by modifying local data or enqueuing operations to modify remote data appropriately. (Note that due to the constructs of SQL, remote reads and modifications sometimes are combined in a single operation.)

Whereas dependency specifications are declarative, the rules derived from them are procedural—the rules monitor dynamic changes and specify further dynamic changes that enforce consistency. Consequently, each rule implements a particular policy for consistency restoration. Our rule generation algorithms produce what we believe are the most intuitive restoration policies based on the semantics of the different forms of dependencies. A straightforward extension to our approach is to offer a choice of policies, allowing the user to guide rule generation accordingly; see Section 6.

We consider each of the four forms of dependencies, describing the policies for dependency enforcement and specifying the generated rules for each. Example rules generated from the example dependencies in Sections 2.1–2.4 are given in [7]; space limitations preclude their presentation here. We assume that rules for a given dependency are installed in each database at a time when it is known that the dependency is satisfied. This assumption is not necessary, however, if when rules for a dependency are installed in a database, the rules are processed once in response to an imaginary transaction in which each rule's trigger table is inserted in its entirety.

4.1 Rules for Directional Existence Dependencies

Consider a generic directional existence dependency:

$$T.\langle C_1, \ldots, C_n \rangle \leftarrow \textbf{select } X_1, \ldots, X_m \textbf{ from } T_1, \ldots, T_k \textbf{ where } P$$

where table T resides at database $\mathcal{DB}_1$ and tables $T_1, \ldots, T_k$ at database $\mathcal{DB}_2$. Let $C'_1, \ldots C'_j$ denote all columns of tables $T_1, \ldots, T_k$ appearing in expressions $X_1, \ldots, X_m$ or in predicate P. Based on the semantics of SQL and the semantics of directional existence dependencies as described in Section 2.1, this dependency can become violated by any of the following operations (and by no others):

deletions from table T
updates to any column $C_1, \ldots, C_n$ of table T
insertions into any table $T_1, \ldots, T_k$
updates to any column $C'_1, \ldots C'_j$ of tables $T_1, \ldots, T_k$

In the case of deletions and updates on table T, if the dependency is violated then consistency can be restored by reinserting the required values that no longer exist. Null values are inserted for columns of T not included in $C_1, \ldots, C_n$; we denote this by "`select C1..Cn [nulls]...`" (concrete examples appear in [7]). Hence, for deletions and updates on T the following rule is generated, to be installed at database $\mathcal{DB}_1$.[7]

```
create rule DelUp on T
when deleted, updated(C1..Cn)
then Temp(C1..Cn) <-
       Remote-Select(DB2,
```

[6] The system also supports rule processing at arbitrary points within a transaction in response to special-purpose commands. The framework in this paper can exploit this feature with no additional mechanism.

[7] We often use multi-column **in**, **not in**, etc., which is available in some but not all SQL implementations; equivalent expressions in standard SQL can be substituted.

```
        "select distinct X1..Xm from T1..Tk
         where P and <X1..Xm> in
           (select C1..Cn from
            (deleted union old-updated))");
    insert into T
     (select C1..Cn [nulls] from Temp
      where <C1..Cn> not in (select C1..Cn from T))
```

Note the use of transition tables **deleted** and **old-updated** in the remote **select** statement. The effect of this is to make the rule *incremental*—rather than returning all tuples satisfying P, the remote **select** returns only those tuples satisfying P and corresponding to changed (deleted or updated) values in T. If it turns out the dependency is not violated, either because the deleted or updated tuples are not required in T or because they were reinserted in the same triggering transition, then the rule's action has no effect on T. (An optimization pertaining to this case is discussed in Section 4.5.) Finally, note the use of **select distinct** so that duplicates are not considered.

In the case of insertions and updates on tables $T_1, \ldots, T_k$, if the dependency is violated then consistency can be restored by inserting the newly required values into T. Hence, for each T_i, $1 \leq i \leq k$, the following rule is generated, to be installed at database $\mathcal{DB}_2$.

```
create rule InsUp on Ti
when inserted, updated(C1'..Cj')
then Temp(C1..Cn) <-
       select distinct X1..Xm
       from T1..
            Ti' as (inserted union new-updated)
            ..Tk where P[Ti'/Ti];
     Enqueue-Insert(DB1,
       "insert into T
          (select C1..Cn [nulls] from Temp
           where <C1..Cn> not in
             (select C1..Cn from T))")
```

The clause "`Ti' as (inserted union new-updated)`" in the **select** statement uses the *table expression* feature of Starburst. In general, the interpretation of "V **as** *Exp*" is that expression *Exp* is evaluated to produce a logical table which is used in the cross-product and assigned table variable V; see [13] for details. `P[Ti'/Ti]` denotes predicate P with any (explicit) references to T_i replaced by T_i'. Hence, replacing T_i with "`Ti' as (inserted union new-updated)`" and P with `P[Ti'/Ti]` has the effect that the union of transition tables **inserted** and **new-updated** is considered everywhere instead of T_i, making this rule incremental as well.

4.2 Rules for Nondirectional Existence Dependencies

Consider a generic nondirectional existence dependency:

$$T_1.\langle C_1^1, \ldots, C_n^1\rangle \leftrightarrow T_2.\langle C_1^2, \ldots, C_m^2\rangle$$

where tables T_1 and T_2 reside at databases $\mathcal{DB}_1$ and $\mathcal{DB}_2$, respectively. As discussed in Section 2.2, this dependency is equivalent to a pair of directional existence dependencies, so rules could be generated accordingly. However, by translating this nondirectional existence dependency to the equivalent pair of directional existence dependencies and then generating rules as in Section 4.1, the reader will find that although the generated rules are correct, their behavior can be inefficient and counterintuitive. For nondirectional existence dependencies, the consistency restoration policy that seems most intuitive is that if a value is inserted into table T_1 then the corresponding value is inserted into table T_2, if a value is deleted from table T_1 then the corresponding value is deleted from table T_2, and vice-versa in both cases. This is implemented by the following two rules for T_1, along with the corresponding rules for T_2.

```
create rule Ins on T1
when inserted
then Enqueue-Insert(DB2,
       "insert into T2
          (select C11..C1n [nulls] from inserted
           where <C11..C1n> not in
             (select C21..C2m from T2))")

create rule Del on T1
when deleted
then Temp(C11..C1n) <-
       select C11..C1n from deleted
       where <C11..C1n> not in
         (select C11..C1n from T1);
     Enqueue-Delete(DB2,
       "delete from T2
        where <C21..C2m> in
          (select C11..C1n from Temp)")
```

Both of these rules are incremental. Note that although the **not in** clause in the **select** statement of the delete rule may appear redundant, it ensures that values are not erroneously deleted from T_2 when the triggering transition at T_1 deletes then inserts the same value.

Now consider updates to columns $C_1^1, \ldots, C_n^1$ of table T_1. To restore consistency, some values may need to be deleted from T_2 and other values may need to be inserted into T_2. This can be achieved simply by modifying the pair of rules above to include **updated** triggering operations and transition tables. Hence, the following two rules are installed at database $\mathcal{DB}_1$, along with the corresponding rules generated for T_2 and installed at $\mathcal{DB}_2$.

```
create rule InsUp on T1
when inserted, updated(C11..C1n)
then Enqueue-Insert(DB2,
       "insert into T2
          (select C11..C1n [nulls]
           from (inserted union new-updated)
           where <C11..C1n> not in
             (select C21..C2m from T2))")

create rule DelUp on T1
when deleted, updated(C11..C1n)
then Temp(C11..C1n) <-
       select C11..C1n
       from (deleted union old-updated)
       where <C11..C1n> not in
         (select C11..C1n from T1);
     Enqueue-Delete(DB2,
       "delete from T2
        where <C21..C2m> in
          (select C11..C1n from Temp)")
```

4.3 Rules for Directional Value Dependencies

Consider a generic directional value dependency:

$$T.\langle C_1, \ldots, C_n\rangle \Leftarrow \textbf{select } X_1, \ldots, X_m \textbf{ from } T_1, \ldots, T_k \textbf{ where } P$$

where table T resides at database $\mathcal{DB}_1$ and tables $T_1, \ldots, T_k$ at database $\mathcal{DB}_2$. Let $C_1, \ldots, C_x$ denote column list $C_1, \ldots, C_n$ augmented to include any other columns of table T appearing in predicate P, and let $C'_1, \ldots C'_j$ denote all columns of tables $T_1, \ldots, T_k$ appearing in expressions $X_1, \ldots, X_m$ or in P. Based on the semantics of SQL and the semantics of directional value dependencies as described in Section 2.3, this dependency can become violated by any of the following operations (and by no others):

insertions into table T
updates to any column $C_1, \ldots, C_x$ of table T
insertions into any table $T_1, \ldots, T_k$
updates to any column $C'_1, \ldots C'_j$ of tables $T_1, \ldots, T_k$

In the case of insertions and updates on table T, if the dependency is violated then consistency can be restored by modifying values in the inserted or updated tuples as appropriate. Hence, for insertions and updates on T the following rule is generated for $\mathcal{DB}_1$, where `<key>` denotes a set of columns that form a key for table T.

```
create rule InsUp on T
when inserted, updated(C1..Cx)
then Temp(C1'..Cj',<key>) <-
       Remote-Select(DB2,
         "select distinct C1'..Cj', <key>[T'/T]
          from T1..Tk,
               T' as (inserted union new-updated)
          where P[T'/T]");
     update T
     set C1..Cn = (select one(X1..Xm) from Temp)
     where <key> in (select <key> from Temp)
```

(Recall that `[T'/T]` denotes replacing all references to T by T'.) The remote **select** statement considers the cross-product of $T_1, \ldots, T_k$ with the inserted or updated tuples from T (making the rule incremental), returning relevant values of all tuples in this cross-product satisfying P. The **update** statement then modifies the inserted or updated tuples in T accordingly. Aggregate function **one** in the **set** clause is present to handle the case when the RHS **select** statement of the dependency produces multiple values for a tuple in T (recall the semantics of directional value dependencies from Section 2.3): function **one** arbitrarily selects exactly one value.

In the case of insertions and updates on tables $T_1, \ldots, T_k$, if the dependency is violated then consistency can be restored by modifying values in T accordingly. Hence, for each T_i, $1 \leq i \leq k$, the following rule is generated, to be installed at database $\mathcal{DB}_2$.

```
create rule InsUp on Ti
when inserted, updated(C1'..Cj')
then Temp(C1'..Cj') <-
       select distinct C1'..Cj'
       from T1.. (inserted union new-updated) ..Tk;
     Enqueue-Update(DB2,
       "update T
        set C1..Cn = (select one(X1..Xm)
                      from Temp where P[Temp/T1..Tk])
        where exists
          (select * from Temp where P[Temp/T1..Tk])")
```

Here we use `[Temp/T1..Tk]` to denote replacing any table reference in $T_1, \ldots, T_k$ by *Temp*. The **exists** clause in the **update** statement ensures that a tuple is not updated unless a (new) required value for that tuple actually exists.

4.4 Rules for Nondirectional Value Dependencies

Consider a generic nondirectional value dependency:

$$T_1.\langle C^1_1, \ldots, C^1_n\rangle \Leftrightarrow T_2.\langle C^2_1, \ldots, C^2_m\rangle \textbf{ where } P$$

where tables T_1 and T_2 reside at databases $\mathcal{DB}_1$ and $\mathcal{DB}_2$, respectively. As with existence dependencies, we could generate rules for these nondirectional dependencies by translating them into the equivalent pairs of directional dependencies, then generating rules accordingly. We can, however, generate rules with a more intuitive restoration policy for the nondirectional case: changes to table T_1 are propagated to table T_2 as appropriate (only insertions and updates can cause consistency to be violated) and vice-versa.[8]

Let $C^1_1, \ldots, C^1_j$ denote column list $C^1_1, \ldots, C^1_n$ augmented to include any other columns of table T_1 appearing in predicate P. The following rule is generated for insertions and updates on T_1; a corresponding rule is generated for T_2.

```
create rule InsUp on T1
when inserted, updated(C11..C1j)
then Temp(C11..C1j) <-
       select distinct C11..C1j
       from (inserted union new-updated);
     Enqueue-Update(DB2,
       "update T2
        set C21..C2m = (select one(C11..C1n)
                        from Temp where P[Temp/T1])
        where exists
          (select * from Temp where P[Temp/T1])")
```

4.5 Discussion

From Sections 4.1–4.4 we see that production rules can be derived from dependency specifications by a relatively straightforward algorithm. Now consider the efficiency of the derived rules. Each rule performs exactly one remote operation and, in some cases, one local operation. All rule actions are incremental—whenever possible they consider the modified tuples in a table rather than the entire table. This is particularly important for remote operations, since tuples must be queued and transmitted to a different site. In all of our rule actions, the transmitted tuples are derived partly or entirely from

[8] If we generated rules for the equivalent directional dependencies, then changes to T_1 would either be propagated to T_2 or would be undone, depending on which rule was chosen first for execution.

transition tables, which we expect to be relatively small. Note that in many of the derived rules, the rule action first performs a local or remote **select** operation to produce a temporary table **Temp**, then performs a local or remote modification operation based on **Temp**. Intuitively, **Temp** contains data that may violate consistency; hence, when consistency is not violated, **Temp** may be empty. A simple optimization to our rule actions checks after the **select** operation whether **Temp** is empty and, if so, skips execution of the modification operation.

5 Correctness

Correctness entails showing that the rules generated for any set of dependencies are guaranteed to maintain consistency across databases with respect to those dependencies through arbitrary modifications to the databases. Since we are considering loosely coupled environments in which different databases may be modified asynchronously and continuously by different users, we must carefully define the correctness requirement.

Definition 5.1 Let $D_1, \ldots, D_n$ be a set of dependencies over databases $\mathcal{DB}_1, \ldots, \mathcal{DB}_m$, and let $R_1, \ldots, R_k$ be a set of rules derived from the dependencies. $R_1, \ldots, R_k$ are *correct with respect to* $D_1, \ldots, D_n$ if the following holds. Let S be any global state of databases $\mathcal{DB}_1, \ldots, \mathcal{DB}_m$ such that S satisfies dependencies $D_1, \ldots, D_n$. Consider any set of user transactions executed on databases $\mathcal{DB}_1, \ldots, \mathcal{DB}_m$ from state S. Suppose the transactions all commit, including their rule processing, and all persistent queues at all sites are empty (i.e. all enqueued operations have been executed), producing a new global state S'. Then S' satisfies $D_1, \ldots, D_n$. □

That is, the rules are correct if they guarantee that whenever the databases begin in a global state in which all dependencies are satisfied, then any future "quiescent" global states also will satisfy all dependencies.[9] The following theorem shows that our method for rule derivation produces correct rules according to this definition.

Theorem 5.2 (Correctness) Let $D_1, \ldots, D_n$ be any set of dependencies in the language of Section 2 and let $R_1, \ldots, R_k$ be the rules derived from $D_1, \ldots, D_n$ as specified in Section 4. $R_1, \ldots, R_k$ are correct with respect to $D_1, \ldots, D_n$.

Proof: Omitted due to space limitations; see [7].

5.1 Infinite Triggering

Definition 5.1 specifies the correctness requirement as an implication: if all transactions commit and all persistent queues are empty then all dependencies are satisfied. We have no control over successful completion of user transactions, and we rely on the system to eventually execute all enqueued operations. However, we would like to ensure that our methods do not introduce deadlock, and that rules cannot trigger each other and consequently perform or enqueue new operations indefinitely. Regarding deadlock, transactions with rule processing and transactions spawned by remote operations certainly may participate in deadlocks, but such deadlocks are not of any special nature and can be broken in the standard way. Infinite rule triggering is a more complex problem. As we will show, infinite rule triggering can occur, but it indicates certain error conditions and can be detected through a limited amount of additional mechanism.

There are two scenarios in which infinite rule triggering may occur:

(1) When the user has specified contradictory dependencies, say D_1 and D_2 (recall Section 2.5). In this case, a rule R_1 may modify data to satisfy D_1, invalidating D_2 and triggering a rule R_2. R_2 will then modify data to satisfy D_2, invalidating D_1 and triggering R_1, and so on.

(2) When simultaneous modifications are performed to data items, say X_1 and X_2, at different sites, and X_1 and X_2 are mutually value dependent. In this case, suppose X_1 is modified from V to V_1 and X_2 is modified from V to V_2. It is possible for rule behavior to be such that X_2 is modified to be consistent with V_1, then X_1 is modified to be consistent with V_2, then X_2 is modified to be consistent with V_2, then X_1 is modified to be consistent with V_1, and so on.

Both of these cases are considered as errors. For case (1), as mentioned in Section 2.5, we would like to detect contradictory dependencies at compile-time through static analysis, but if this is not possible we certainly should detect the resulting infinite rule triggering at run-time. For case (2), simultaneous modifications to mutually dependent data should be avoided by applications since this corresponds to unresolvable loss of serializability, but if such modifications occur we should detect the resulting infinite triggering at run-time.

In case (1), infinite rule triggering may occur within a single transaction at a single site, or it may occur across sites (i.e. remote modification operations trigger rules that enqueue new operations that trigger rules and so on). In case (2), infinite rule triggering occurs only across sites. In the Starburst Rule System [20], infinite triggering at a single site is detected by the rule processor as follows. Based on the set of rules present in the system, the system administrator establishes an upper limit on the number of rule triggerings within each transaction. If this limit is reached, it is assumed that infinite rule triggering has occurred, rule processing ceases, and an error condition is raised.

A related approach can be taken for detecting infinite rule triggering across sites in the context of our method for consistency maintenance. Assume that each dependency has an identifier, and that attached to each enqueued remote operation is a list of identifiers indicating the "history" of dependencies producing that operation.

[9] This correctness definition does allow the existence of inconsistent global states, just not the persistence of such states. Hence, it is possible for retrieval operations on multiple sites to yield data that does not satisfy the consistency requirements. This seems unavoidable in loosely coupled environments such as those considered here.

For example, suppose a rule R_1 derived from a dependency D_1 is triggered by user modifications and enqueues an operation O_1; then list $[D_1]$ is attached to O_1. If O_1's execution triggers a rule R_2 derived from a dependency D_2, and R_2 enqueues an operation O_2, then list $[D_2, D_1]$ is attached to O_2, and so on. There may be terminating rule executions in which the same dependency appears multiple times in a single history, so when a dependency appears twice in a list we do not immediately deduce that rules will trigger each other indefinitely. However, the system administrator can establish an upper limit on the number of times a single dependency may appear in a history, or perhaps on the length of each history. (In the latter case a simple counter can be used in place of the history.) If this limit is reached, it is assumed that infinite rule triggering has occurred, the operation is prevented from triggering additional rules, and an error condition is raised.

5.2 Queue Contention

In Section 3.1 we explained that at each site remote operations must be enqueued and dequeued in the commit order of the enqueuing transactions, and that this can be implemented by transactions locking the back of the queue when enqueue operations are performed. Since each transaction locks only its local queue, no deadlock can occur, but the queue may become a point of contention between transactions, reducing concurrency. To lessen this contention, multiple local queues may be used—one for each remote site—without compromising the correctness of our approach. When an enqueue operation is issued, the request is appended to the appropriate queue. Contention obviously is reduced, but deadlocks may now occur. Such deadlocks can be broken in the standard way. Alternatively, deadlock can be avoided by establishing an ordering on the queues, then specifying priorities between rules [20] to ensure that each transaction enqueues operations in an order consistent with the queue ordering.

5.3 Additional Rules

Finally, observe that correctness is guaranteed even when one or more databases include arbitrary production rules in addition to those generated for consistency maintenance. Recall the semantics of rule execution (Section 3.2): a rule is triggered if a triggering operation occurred in the transition since the last time it was executed; if it has not yet been executed, it is triggered if a triggering operation occurred in the transition since the start of the transaction. Hence, the first time a dependency-maintaining rule R is triggered during rule processing, it considers all changes since the start of the transaction. Suppose that, subsequently during rule processing, changes are made by a non-dependency-maintaining rule that may violate R's dependency. Then R will be triggered again and will restore the dependency according to the new set of changes. When rule processing terminates, no rules are triggered, so all dependency-maintaining rules have processed all relevant changes.

6 Conclusions and Future Work

We have described a framework whereby users specify consistency requirements across semantically heterogeneous databases using an expressive SQL-based language. Consistency requirements are translated automatically into production rules to be installed at individual databases. The rules are triggered by any local modifications that may cause consistency to be violated; they check consistency by performing remote queries and restore consistency by executing local modifications or enqueuing remote modifications. The rules are guaranteed to maintain consistency incrementally and correctly.

An important consideration for the usability of our approach is the environment in which dependencies are specified (and revoked), and the method by which derived rules are installed (and deleted) in appropriate databases. For an initial prototype, we believe it is adequate if dependencies are parsed and checked for semantic correctness, rules are generated by a simple compiler, and generated rules are installed "off-line" when it is known that relevant dependencies are satisfied. Subsequently, it would be useful to incorporate static analysis algorithms to aid dependency specification, similar to those given for rule definition in [1]. In particular, we would like to predict whether dependencies may be contradictory (Section 2.5) and whether value dependencies may involve multiple values (Sections 2.3 and 2.4). It also would be useful to incorporate a mechanism for installing rules when dependencies may not be satisfied, as suggested in Section 4.

In Section 4 we discussed allowing the user to choose from a number of alternative consistency restoration policies on a rule-by-rule basis, i.e. one policy for each type of modification that may cause inconsistency. For this extension we must identify reasonable alternatives and ensure that the implementation of each alternative indeed restores consistency. When this holds, any combination of policies selected by the user is guaranteed to behave correctly. Allowing **rollback** as a restoration policy also guarantees consistency, but in our framework this requires a transaction model that supports rollback across related transactions (i.e. across transactions spawned by enqueue operations). Note that many of the proposed extended transaction models support this or similar capabilities; see [12].

A number of extensions to the dependency specification language are possible. We have restricted our dependencies to operate over exactly two databases, since this appears to be the most common requirement. However, there certainly are scenarios in which the existence or value of data in one database depends on related data in several other databases. If we permit such dependencies, then unless the environment supports **select** operations that span multiple databases (which is unlikely), the generated rules must execute multiple remote **select** operations and combine the results to determine whether consistency is violated and how it should be restored. If dependencies instead operate over only one database, then all generated operations are local and our methods amount to a mechanism for managing a restricted form of single database integrity constraints. A substantial extension would be to make our language sufficiently powerful that arbitrary constraints across databases could be expressed, but user interaction is then required for generating consistency restoration actions [5], and rule generation is no longer fully automatic.

Our framework is designed for loosely coupled environments in which distributed transactions are not supported. However, there may be scenarios in which a heterogeneous multidatabase environment does support transactions across sites. In such environments, production rule processing might proceed in a more tightly coupled, synchronous fashion, e.g. as in [8]. By replacing all asynchronous **Enqueue** operations by synchronous **Remote** operations, our framework can be used to generate rules for such environments that maintain consistency in a synchronous (single-transaction) fashion.

Finally, our framework is based on environments in which all databases use the relational model. Although our specification language and generated rules clearly are model-dependent (since the language is based on SQL and the rule system operates on relational databases), the same approach can be used for other data models if an appropriate specification language is designed and appropriate production rule facilities are available. Even in multidatabase environments with several different data models, our approach can be used to monitor and maintain consistency provided that production rules, remote operations, and persistent queues are supported.

Acknowledgements

Thanks to Alex Aiken for valuable discussions about correctness, to Bobbie Cochrane for careful reading and useful contributions, to Bill Cody for suggested improvements, to Linda DeMichiel for helpful comments on heterogeneity, and to Hector Garcia-Molina for advice on transactions.

References

[1] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[2] P.A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 112–122, Atlantic City, New Jersey, May 1990.

[3] N. Cercone, M. Morgenstern, A. Sheth, and W. Litwin. Resolving semantic heterogeneity. In *Sixth International Conference on Data Engineering* (panel), Los Angeles, California, February 1990.

[4] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, April 1985.

[5] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[6] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[7] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. IBM Research Report RJ 9064, IBM Almaden Research Center, October 1992.

[8] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 339–351, Vancouver, British Columbia, August 1992.

[9] IEEE Computer. *Special Issue on Heterogeneous Distributed Database Systems*, 24(12), December 1991.

[10] U. Dayal and H.-Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering*, 10(6):628–645, November 1984.

[11] U. Dayal and J. Widom. Active database systems. In *ACM SIGMOD International Conference on Management of Data* (tutorial), San Diego, California, June 1992.

[12] A. Elmagarmid, editor. *Special Issue on Unconventional Transaction Management,* Data Engineering Bulletin 14(1), March 1991.

[13] L.M. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[14] M. Hsu and A. Silberschatz. Unilateral commit: A new paradigm for reliable distributed transaction processing. In *Proceedings of the Seventh International Conference on Data Engineering,* pages 286–293, Kyoto, Japan, April 1991.

[15] W. Kent. Solving domain mismatch and schema mismatch problems with an object-oriented database programming language. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 147–160, Barcelona, Spain, September 1991.

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

[17] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, Portland, Oregon, May 1989.

[18] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, 24(12):46–53, December 1991.

[19] A. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[20] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.