Protocols for Integrity Constraint Checking in Federated Databases*

Paul Grefen
Department of Computer Science
University of Twente
7500 AE Enschede, The Netherlands
grefen@cs.utwente.nl

Jennifer Widom
Department of Computer Science
Stanford University
Stanford, CA 94305 USA
widom@cs.stanford.edu

Abstract

A federated database is comprised of multiple interconnected database systems that primarily operate independently but cooperate to a certain extent. Global integrity constraints can be very useful in federated databases, but the lack of global queries, global transaction mechanisms, and global concurrency control renders traditional constraint management techniques inapplicable. This paper presents a threefold contribution to integrity constraint checking in federated databases: (1) The problem of constraint checking in a federated database environment is clearly formulated. (2) A family of protocols for constraint checking is presented. (3) The differences across protocols in the family are analyzed with respect to system requirements, properties guaranteed by the protocols, and processing and communication costs. Thus, our work yields a suite of options from which a protocol can be chosen to suit the system capabilities and integrity requirements of a particular federated database environment.

1 Introduction

The integration of multiple database systems has become one of the most important topics in both the research and commercial database communities. Information servers are being developed that provide integrated access to multiple data sources. Legacy database systems and applications are being coupled to form enterprise-wide information systems. Large workflow management applications require the routing of information through multiple, autonomous local systems. The integration of autonomous database systems into loosely-coupled *federations* requires the development of novel database management techniques specific to these environments. Many concepts and techniques from centralized or tightly-coupled distributed databases are not directly applicable in a federated environment.

One important issue in federated database systems is checking *integrity constraints* over data from multiple sites in the federation. Integrity constraints specify those states of the (global) database that are considered to be semantically valid. In a federated environment, integrity constraints might specify that replicated information is not contradictory, that information is not duplicated, that certain referential integrity constraints hold, or that some other condition is true over the data in multiple databases. Below, we describe an example scenario involving cooperative hospital information systems with cross-system constraints.

In traditional centralized or tightly-coupled distributed databases, transactions form the cornerstone of integrity constraint checking: Before a transaction commits, it ensures that all integrity constraints are valid. If a constraint is violated, then the transaction may be aborted, the constraint may be corrected automatically, or an error condition may be raised [GA93]. Unfortunately, the lack of inter-site transaction mechanisms in federated databases renders traditional constraint checking mechanisms inapplicable.

This paper addresses the problem of integrity constraint checking in federated databases; we make a threefold contribution. First, the constraint checking problem is formulated in the specific context of federated databases. In particular, an alternative notion of constraint checking correctness must be defined, since the transaction-based approach from traditional environments is inappropriate. Then a family of constraint checking protocols is developed along a number of protocol "dimensions." Finally, the protocols in the family are analyzed and compared with respect to the requirements of the component database systems, the constraint checking properties guaranteed by the protocols, and the processing and communica-

*This work was supported at Stanford by ARPA Contract F33615-93-1-1339 and by equipment grants from Digital Equipment and IBM Corporations. A short preliminary version of this work appeared in the proceedings of the 1996 CoopIS conference.

tion costs. By providing a family of protocols, rather than a single protocol, we permit a protocol in the family to be chosen and tailored for the capabilities and requirements of a particular federated database environment.

1.1 Related Work

Most work addressing the problem of integrity constraint checking in multidatabase environments has considered tightly-coupled distributed databases in which global queries, global transactions, and global concurrency control are present, e.g., [GA90, Qia89, SV86]. Since these approaches rely on global services that typically are unavailable in federated databases, they are inappropriate for the environment we consider. Note that some approaches focus on relaxing the traditional notion of transaction serializability for constraints in distributed environments, e.g., [BGM92, Elm91], but some level of locking and global query facilities is still expected.

A few recent papers have addressed the issue of monitoring constraints in loosely-coupled, distributed, and sometimes heterogeneous database environments. One class of work involves *local constraint checking*—deriving tests whose success over one database implies the validity of a multidatabase constraint [BGM92, GSU94, GW93]. Local tests optimize the constraint checking process, but they still require a conventional (non-local) method when the local test fails. As will be seen, in this paper we develop protocols that integrate local checking with non-local methods. In [CGW96], a framework and toolkit are described for constraint management in loosely-coupled, highly heterogeneous environments. The focus in [CGW96] is on maintaining constraints across systems that have varying capabilities and varying "willingness" to participate in constraint checking protocols, on describing the timing properties associated with constraint checking, and on notions of "conditional consistency" that are weaker than the form of consistency we consider.

The issue of maintaining consistency of replicated data across loosely-coupled, semantically heterogeneous databases has been considered in, e.g., [CW93, DD95, RSK91]. Our constraint checking problem can be seen as a special case (or first step) of that consistency maintenance problem. In [CW93], a method is described that relies on active rules and persistent queues. The approach is similar to the simplest case in our family of protocols. Similar issues are addressed in [RSK91], but no specific protocols are provided. In [DD95], an implementation mechanism based on active rules is proposed for maintaining replication consistency.

A related problem is that of maintaining views over distributed data in loosely-coupled systems, addressed in [ZGH95] in the context of *data warehousing*. In [ZGH95], algorithms are presented for handling the anomalies that arise when materialized views are refreshed in an asynchronous manner. The algorithms rely on view definitions and *compensating queries*, and thus are not directly applicable to our problem. However, they may become relevant as we extend our protocols to handle more complex constraints or to incorporate consistency repair.

Finally, we note that in the field of distributed (operating) systems there has been considerable work in the area of *snapshots* and *consistent global states*; see e.g., [CL85, BM93]. Although this work appears highly related to the problem we are addressing, there are two significant differences:

- The conditions to be evaluated in the distributed system setting are *stable*, meaning that once a condition becomes valid, it stays valid. This property is not true of database integrity constraints; in fact, our goal is to track constraints as they move from validity to invalidity and vice-versa.
- Protocols for the distributed system setting are designed to obtain some (any) global state, but not to obtain all global states. In contrast, to effectively monitor database constraints it is necessary to monitor all global states, or at least a subset of those states corresponding to consistency "checkpoints" (see Section 2.1).

1.2 Structure of the Paper

The structure of this paper is as follows. Section 2 formalizes the problem of integrity constraint checking in federated databases. It provides necessary concepts and definitions, describes the basic system architecture we address, discusses implementation aspects, and defines the class of integrity constraints we deal with. In addition, a simple example application is introduced. Section 3 presents the family of constraint checking protocols we have developed. Section 4 analyzes the family of protocols presented in Section 3. First, the "design space" for protocols is further inspected. Then the various protocols are compared with respect to their requirements, properties, and costs. Finally, we describe how the protocols can be extended to more general architectures. In Section 5 we conclude and discuss future work.

2 Preliminaries

This section presents preliminary material for the remainder of the paper. First, the concepts used in our work are defined formally. Next, we present the basic architecture we consider—a federation of two autonomous, relational databases, each with a component for constraint checking—and we discuss implementation aspects of the architecture. Then we describe the class of integrity constraints we consider. The section ends with the description of an example application that is used to motivate our work.

Note that although we cast our work in the context of federations of relational databases, the relational restriction is introduced primarily for concreteness and clarity. All of our definitions, protocols, and analyses adapt easily to other data models, as well as to federations involving multiple data models.

2.1 Concepts and Definitions

We start with our definition of a federated relational database system.

Definition 1. A *federated relational database system* F is a set of n interconnected autonomous database systems $\{S_1, \ldots, S_n\}$. Each autonomous database system $S_i \in F$ hosts a local database D_i with schema $\mathcal{D}_i$. A local database D_i consists of relations $R^i_1, \ldots, R^i_{n_i}$ with schemata $\mathcal{R}^i_1, \ldots, \mathcal{R}^i_{n_i}$. The set of all relation schemata $\mathcal{R}^i_j$ in F is called the global database schema $\mathcal{G}$ of F. □

In the following, we assume a global clock so that we can refer to global times in defining certain concepts. The global clock is used for concept definition only—it is not a requirement of the federated database systems we consider. Also for definitional purposes, we assume that each local database processes its updates in the context of local transactions. However, many of the constraint checking protocols we present are also applicable to local systems that do not support transactions.

Definition 2. The *user-observable* state of a relation R^i_j at global time t is the state of R^i_j reflecting all and only those local update transactions committed before t at site S_i. □

Definition 3. The *global state* G of a global database schema $\mathcal{G}$ at global time t is the set of user-observable states of all relations in $\mathcal{G}$ at global time t. □

Due to the lack of global transactions in a federated environment, it can be difficult or impossible to observe the state of a global database at a single global time. If an application or protocol does not read local states simultaneously, it may observe a global state that has never actually existed. We call such an observed state a *phantom state*.

Definition 4. A *phantom state* Φ of a global database schema $\mathcal{G}$ observed by application A at time t_2 as a consequence of a request (or set of requests) by A at time t_1 is a set of states of all relations in $\mathcal{G}$ such that there exists no t_3 where $t_1 < t_3 < t_2$ and the global state of $\mathcal{G}$ at time t_3 is Φ. □

The relevance of phantom states will become clear when we develop our protocols in Section 3. Next, we turn to the definition of global integrity constraints, and we formalize our notion of correctness in constraint checking.

Definition 5. A *global integrity constraint* I is a boolean expression over a global database schema $\mathcal{G}$, i.e., a function $I\colon \mathcal{G} \rightarrow \{true, false\}$. A global integrity constraint cannot be expressed over a local database schema $\mathcal{D}_i \in \mathcal{G}$ (otherwise the constraint would be local, not global). A *constraint checking protocol* for I is an algorithm for evaluating I. □

In centralized database systems or tightly-coupled distributed database systems, the transitions between database states are determined by transactions—atomic operations whose intermediate states have no semantics beyond the transaction. Consequently, in these environments, integrity constraints generally are required to hold in the states immediately preceding and following each transaction. Since federated environments consist of multiple, autonomous database systems lacking global transactions, we must rely on other concepts to determine the global database states that should satisfy the integrity constraints. We define a notion of global states in which the federated system is "at rest." These *quiescent states* correspond roughly to the before and after transaction states in traditional database systems, and are the states in which we want to ensure that integrity constraints are not violated.

Definition 6. A federated database system F is in a *quiescent state* at time t if all local update transactions submitted before t have committed, and all constraint checking protocols triggered by any updates before t have completed. □

It is important to note that, similar to the before and after transaction states in traditional systems, quiescent states may not physically exist. However, the logical notion of such states is appropriate for defining the correctness of integrity constraint checking.

Two important properties of our constraint checking protocols are defined with respect to quiescent states: *safety* and *accuracy*.

Definition 7. Consider a global database schema $\mathcal{G}$ and a global integrity constraint I over $\mathcal{G}$. A constraint checking protocol for I is *safe* if the transition from any quiescent global database state $G_0 \in \mathcal{G}$ that satisfies I to any other quiescent state $G_1 \in \mathcal{G}$ that does not satisfy I always results in the protocol raising an alarm.[1] $\square$

Definition 8. Consider a global database schema $\mathcal{G}$ and a global integrity constraint I over $\mathcal{G}$. A constraint checking protocol for I is *accurate* if, after any quiescent global database state G_0 at time t_0, a protocol-generated alarm at time t_2 implies the existence of a global database state G_1 at time t_1 such that $t_0 < t_1 \leq t_2$ and G_1 does not satisfy I. $\square$

That is, a protocol is safe if it detects every transition to a quiescent state in which the constraint becomes violated. We assume that safety is required of any constraint checking protocol that will be useful in practice. A safe protocol may be "pessimistic," however, in the sense that it raises too many alarms. (For example, a protocol that constantly raises alarms is, in fact, safe.) A protocol is accurate if, whenever an alarm is raised, there is indeed a state in which the constraint is violated. Although accuracy is a desirable feature of a constraint checking protocol, it is not always necessary—in some environments a limited number of "false alarms" is tolerable.

Our last definition involves the representation of database updates.

Definition 9. Consider a relation R in a local database system of a federation. When R is modified, we use ΔR to denote all modified tuples (inserted, deleted, or updated), $\Delta^+ R$ to denote all new tuple values (inserts and after-images of updates), $\Delta^- R$ to denote all old tuple values (deletes and before-images of updates), and $\Delta^0 R$ to denote all unmodified tuples. We refer to all forms of Δ's as *delta sets*. We assume that delta sets correspond to the modifications performed by a single local transaction, however other update granularities can be used without affecting our protocols. $\square$

[1] In this paper we do not consider the reaction to constraint violations, although it is an important area of future work. Rather, we focus on the detection of constraint violations, and we say that when a protocol detects a constraint violation it raises an "alarm."

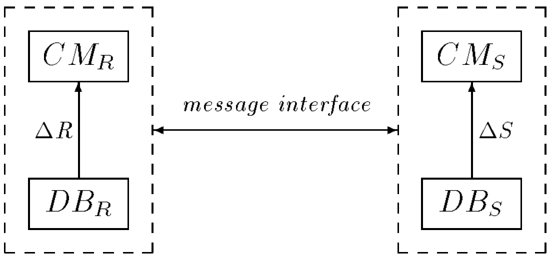

Figure 1: Basic architecture

2.2 Basic Architecture and Assumptions

Figure 1 depicts the basic architecture we address for integrity constraint checking in federated databases. To develop our protocols we consider the restricted case of two databases, each containing a single relation. In practice, most multi-site constraints involve only two databases or can be reformulated as an equivalent set of two-site constraints. However, our protocols can be generalized to handle constraints over more than two sites—see Section 4.3 for a discussion. Generalizing to more than one relation per site is straightforward.

In the diagram, DB_R and DB_S denote two local database systems managing relations R and S, respectively. Connected to the local database systems are *constraint managers*, CM_R and CM_S in the diagram. There is one constraint manager for each local database—each constraint manager handles the global constraints that may be invalidated by operations on its corresponding database. (Consequently, each global constraint is replicated at every site containing data involved in the constraint.) We assume that any local constraints are managed within the appropriate database system using traditional mechanisms (see, e.g., [GA93]).

The constraint manager is notified of all changes (Δ's) to the local database that may violate a constraint. For convenience, we may assume that delta notifications are sent at the end of each local update transaction; however, as mentioned earlier, any granularity of delta notification can be handled by our protocols. Of particular importance is that we do not generally assume that delta notifications or subsequent constraint checking protocols occur as part of a local transaction, and constraint managers need not be tightly coupled with their corresponding database systems.

The database systems in the federation are autonomous, in that global queries, global transactions, and global concurrency control mechanisms are not available. A message-passing interface connects the two sites. This interface allows the constraint managers to exchange messages in a cooperative fashion, and may allow a constraint manager to send requests directly to the remote database system.

2.3 Implementation Aspects

An essential aspect of implementing the above architecture is supporting the delta notification mechanism by which database systems in a federation communicate with their respective constraint managers. Various implementation schemes can be used to realize this mechanism, including *active database rules*, *passive database rules*, *database triggers*, and *polling*.

Active database rules follow the *event-condition-action* (ECA) paradigm [WC96]. These rules can be used to implement the delta notification mechanism in a straightforward way:

WHEN *relevant update has occured*
IF *global constraint possibly violated*
THEN *notify constraint manager*

Clearly, the event and condition (WHEN and IF clauses) are "pessimistic," since the actual violation of a global constraint cannot in general be determined by the local database system.

For database systems that do not include an active rule manager, *passive rules* can be used: Passive rules are processed by a transaction modification component built on top of a conventional database system [Gre93, Mon95]. While active rules respond to the actual effects of updates to the database, passive rules act on the mere execution of an update operation, regardless of its effects. Thus, the triggering behaviour of passive rules is more pessimistic, but their implementation and execution costs can be lower.

Database triggers, as supported by many recent commercial database systems [WC96], can be used in a similar way to the rules described above. Usually, their abstraction level is lower than that of active or passive rules, making the implementation of the notification mechanism slightly more complex.

Finally, in cases where no triggering mechanism is available at the database system level (e.g., in strictly closed legacy systems), delta notification must be handled by the constraint manager using a polling mechanism. In this approach, the constraint manager polls the local database system at a certain frequency to detect when changes have occurred. Certainly this approach is the least desirable from both a performance and architecture standpoint, but it is necessary in the case of closed systems with no notification mechanism.

2.4 Integrity Constraints

In this paper we consider constraints over two relations, R and S. In particular, we consider constraints that can be evaluated *incrementally*. By incremental, we mean that if the constraint is valid initially, and if one of the relations is changed—relation R, say—then the constraint can be reevaluated by considering only the changes to R (ΔR) and the relation S. This class of constraints includes many of the most common constraint types, such as referential integrity and mutual exclusion. See the earlier technical report version of this paper [GW94] for a complete definition of the class of constraints considered. The problem of incremental constraint checking has been studied extensively (see, e.g., [BLT86, GA90, Nic82, RCB89, QW91, Ull89]). The protocols we present can be extended in a straightforward way to constraints over more than two relations; the two-relation restriction is adopted for clarity and brevity only.

We assume that constraints are expressed as queries, where the constraint is satisfied iff the query result is empty. This is a common and convenient formulation [GA93, GSU94, GW94], equivalent to expressing constraints as logical formulae. Hence, we denote a constraint C over relations R and S as a query $Q(R, S)$. A query to check C incrementally with respect to changes on R is denoted $Q'(\Delta R, S)$, where again the constraint is satisfied iff the query result is empty (assuming the constraint held before R was modified [BLT86, QW91, Ull89]). $Q'(R, \Delta S)$ is similar.

2.5 Example Application

Suppose we have two hospitals, H_A and H_B, in two neighboring cities. Each hospital has its own database system, S_A and S_B respectively. Because the hospitals cooperate by sharing physicians, the two database systems are connected into a federation. The relations of the federated database are shown in Table 1. Both hospitals keep a local record of their patients, and each patient has a reference to his or her physician. To avoid duplication, all physicians are registered at hospital H_A.

The hospitals enforce a rule that each patient can be registered at only one hospital. In addition, all patients must have a registered physician. These rules lead to the following three integrity constraints, expressed as relational algebra queries. (Recall that the

system	relation
S_A	$Patients_A(SSN, Name, Address, DateOfBirth, Physician)$
S_B	$Patients_B(SSN, Name, Address, Phone, Sex, Physician)$
S_A	$Physicians(PhN, Name, Phone)$

Table 1: Example database relations

constraint is satisfied when the result of the query is empty.)

$$C_1 : \; Patients_A \ltimes_{SSN=SSN} Patients_B$$
$$C_2 : \; \Pi_{Physician} Patients_A - \Pi_{PhN} Physicians$$
$$C_3 : \; \Pi_{Physician} Patients_B - \Pi_{PhN} Physicians$$

Constraints C_1 and C_3 are global constraints, since they involve relations at two different sites. Constraint C_2 is a local constraint, so it can be enforced at its local site (S_A) using standard methods. Constraints C_1 and C_3 can each be "factored" into two incremental global constraints [Ull89], dealing with relevant updates to each of the two relations:

$$C_1^A : \; \Delta^+ Patients_A \ltimes_{SSN=SSN} Patients_B$$
$$C_1^B : \; Patients_A \ltimes_{SSN=SSN} \Delta^+ Patients_B$$
$$C_3^B : \; \Pi_{Physician} \Delta^+ Patients_B - \Pi_{PhN} Physicians$$
$$C_3^A : \; \Pi_{Physician} Patients_B \cap \Pi_{PhN} \Delta^- Physicians$$

3 The Family of Protocols

As motivated earlier, conventional integrity constraint checking methods are not applicable in federated environments, so alternative protocols must be designed. In the remainder of this paper we develop and analyze a family of constraint checking protocols suitable for federated databases. The members of this family have different requirements, different properties, and different performance characteristics. The "root" of the family is a very simple protocol, described in Section 3.1. Although this protocol is safe, it is inaccurate because it may evaluate constraints over phantom states (recall Definition 4). In Sections 3.2–3.8 we enhance the simple protocol along a number of different dimensions to obtain more useful protocols.

In designing the protocols, we are especially interested in the safety and accuracy properties as defined in Section 2.1. The root of the protocol family is safe, and so are all of the protocols derived from it. To obtain accuracy, we enhance the root protocol along the dimensions of *timestamping* mechanisms and *local transaction* mechanisms. Timestamping mechanisms annotate delta sets and query results with global timestamps,[2] enabling the algorithm to detect when phantom states may have been used in constraint evaluation. In contrast, local transaction mechanisms can be used to prevent the evaluation of constraints over phantom states.

[2]Here a global clock may be required; see Section 3.3.

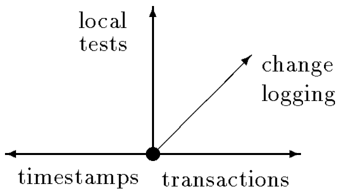

Figure 2: Protocol dimensions

In addition to safety and accuracy, we also are interested in the performance of the protocols, and in ensuring that the local databases remain as autonomous as possible. For these purposes, we enhance the protocols along the dimensions of *change logging* mechanisms and *local test* mechanisms. Change logging mechanisms accumulate updates in special purpose data sets so that database updating and constraint checking need not occur at the same granularity. Local test mechanisms check global constraints by accessing local data only (whenever possible), thus avoiding any kind of global coordination.

These four dimensions delineate a "design space" for constraint checking protocols, illustrated in Figure 2. In the remainder of this section the various protocols are developed, beginning with the simple protocol at the root—the central dot in Figure 2. For each protocol, we specify the protocol in a table that contains a set of sequentially occurring *steps*. Each step is performed by an *actor*—a database system or a constraint manager. Associated with each step is an *action*. Actions vary from simple query evaluations or messaging commands to more complex behavior. The table for each protocol is accompanied by a figure, which illustrates how the protocol behaves within the context of the basic architecture introduced earlier in Figure 1.

1	DB_R	send ΔR to CM_R
2	CM_R	receive ΔR from DB_R
3		send $Q(\Delta R, S)$ to DB_S
4	DB_S	receive $Q(\Delta R, S)$ from CM_R
5		evaluate $Q(\Delta R, S)$
6		send query result to CM_R
7	CM_R	receive query result from DB_S
8		raise alarm if query result is non-empty

Table 2: DRQ Protocol

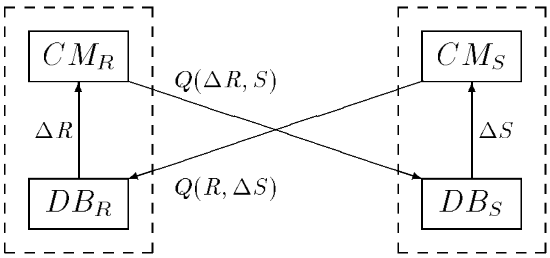

Figure 3: DRQ Protocol

3.1 The Direct Remote Query Protocol (DRQ)

The *Direct Remote Query Protocol* (DRQ) is specified in Table 2 and depicted in Figure 3. Table 2 specifies how the protocol responds to a delta notification for relation R.[3] A symmetric protocol is used to handle updates to S. Protocols handling updates to R and to S may run concurrently. We will assume that multiple updates to R or to S are handled sequentially, although this assumption is not strictly necessary. Figure 3 illustrates the protocol for both updates to R and updates to S.

The DRQ protocol is very simple: When the constraint manager is notified of an update, it sends the appropriate incremental query for evaluation at the other site. Delta sets are usually small enough that the actual data can be transmitted in a straightforward way. For example, $Q(\Delta R, S)$ may by a query over S with the values from ΔR "plugged in." If the query result is non-empty, then the constraint manager raises an alarm. The DRQ protocol always detects when a constraint is violated, i.e., it is safe. Unfortunately, DRQ can easily produce "false alarms," i.e., it is inaccurate. (As pointed out earlier, a protocol that raises an alarm every time there is an update also is safe but inaccurate. It should be evident that DRQ is much less inaccurate than such a protocol.)

To show that the DRQ protocol is safe, we show that starting from a quiescent consistent global state D_0 at time t_0, a quiescent inconsistent global state D_n at time t_n cannot be reached without an alarm being raised:

1. At least one update must have occurred between times t_0 and t_n to reach an inconsistent quiescent state D_n from consistent state D_0.

2. Assume that update u_1 occurring at time t_1, $t_0 < t_1 < t_n$, is the last such update producing an inconsistent state.

3. Update u_1 triggers a constraint checking process that does not raise an alarm, so there exists a time t_2, $t_1 < t_2 < t_n$, at which the state is consistent.

4. Since the state at t_n is inconsistent, there must be an update u_2 at time t_3, $t_2 < t_3 < t_n$, that causes the inconsistent state. This contradicts the fact that u_1 is the last update invalidating the constraint.

The inaccuracy of DRQ is shown by a counterexample in which an alarm is raised but there is never an inconsistent global state. The counterexample is based on the example application from Section 2.5:

1. Consider constraint C_1, and let relations $Patients_A$ and $Patients_B$ both be empty in an initial (consistent) global database state at time t_0.

2. Suppose a local transaction is executed that inserts one tuple T with $SSN = X$ into $Patients_A$ and commits at time t_1, $t_0 < t_1$. $\Delta Patients_A$ is sent to the constraint manager.

3. Let the triggered constraint checking process read relation $Patients_B$ at time t_2, $t_1 < t_2$.

4. Now suppose two other local transactions are executed, the first deleting tuple T from $Patients_A$ and committing at time t_3, the second inserting a tuple T' with $SSN = X$ into $Patients_B$ and committing at time t_4, such that $t_1 < t_3 < t_4 < t_2$.

5. The DRQ protocol will raise an alarm after time t_2, even though no inconsistent state ever existed (since there was never more than one tuple in the database).

[3] Recall that a delta notification is associated with a delta set ΔR. The delta set may contain the updates performed by a local transaction, or it may reflect some other unit of work relevant for local database DB_R.

1	DB_R	send ΔR to CM_R
2	CM_R	receive ΔR from DB_R
3		send $Q(\Delta R, S)$ to CM_S
4	CM_S	receive $Q(\Delta R, S)$ from CM_R
5		submit $Q(\Delta R, S)$ to DB_S
6	DB_S	receive $Q(\Delta R, S)$ from CM_S
7		evaluate $Q(\Delta R, S)$
8		send query result to CM_S
9	CM_S	receive query result from DB_S
10		send query result to CM_R
11	CM_R	receive query result from CM_R
12		raise alarm if query result non-empty

Table 3: IRQ Protocol

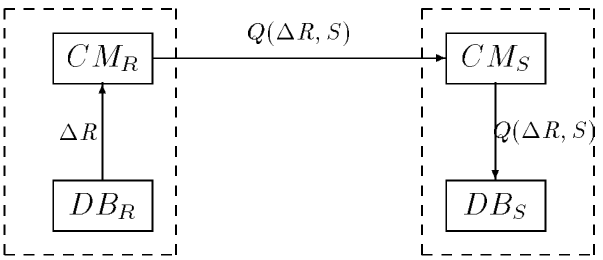

Figure 4: IRQ Protocol

Inaccuracy is due to the fact that the DRQ protocol may evaluate the constraint over a phantom state, since relations R and S are accessed at different times. Note again that the absence of global concurrency control mechanisms makes the traditional solution to this problem inapplicable here. We can solve the problem in two ways, by *detection* or by *prevention*:

- We can use timestamping information to detect possible evaluation over phantom states.
- We can exploit local transaction mechanisms to prevent evaluation over phantom states.

The first solution is explored in Section 3.3, the second solution in Section 3.5. First, a slight variation on the DRQ protocol is presented.

3.2 The Indirect Remote Query Protocol (IRQ)

The DRQ protocol relies on the capability of each database system to process queries issued from a remote constraint manager. If remote query services are unavailable, then we can use a variation on the DRQ protocol that uses peer-to-peer communication between the constraint managers. This *Indirect Remote Query Protocol* (IRQ) is specified in Table 3 and illustrated in Figure 4. For clarity, the figure (and all protocol figures to follow) only shows the case where R is updated. The case where S is updated is symmetric in all protocols.

The IRQ protocol is the same as the DRQ protocol, except queries are routed through constraint managers. In step 10 of Table 3, note that CM_S could raise the alarm itself if the query result is non-empty, instead of sending the result to CM_R. However, it may be preferable for alarms to be raised at the site of the constraint-violating update. Like the DRQ protocol, the IRQ protocol is safe but inaccurate.

3.3 The Timestamped Remote Query Protocol (TRQ)

The DRQ and IRQ protocols lack the accuracy property because they cannot distinguish true global states from phantom states. To overcome this problem, we first enhance the IRQ protocol with a timestamping technique. This technique allows the constraint manager to detect when a phantom state may have been used for query evaluation, and to reevaluate queries when this happens. To make the protocol more efficient, when reevaluation is necessary it is performed with a *cumulative delta set*, i.e., a delta set combining multiple delta notifications from a single site. (We assume a *net effect* semantics for cumulative delta sets [WC96].)

The *Timestamped Remote Query Protocol* (TRQ) is specified in Table 4 and depicted in Figure 5. In the figure and the table, the t_i's denote timestamps attached to messages, and $\Delta R@t$ and $\Delta S@t$ denote notifications of updates occurring at time t. Assume for now that a synchronous global clock is used to generate timestamps; Section 3.4 discusses how this assumption can be relaxed. The TRQ protocol requires that the interface between each constraint manager and its database system is *order preserving.* In particular, a query result sent to the constraint manager must follow the notification of a relevant update if that update was performed before the query was evaluated.

The TRQ protocol behaves as follows. When constraint manager CM_R is notified of an update to R, it sends the appropriate incremental query Q for evaluation to constraint manager CM_S, along with a timestamp for the update. CM_S requests evaluation of query Q at database DB_S. If an update occurs to S between the time of R's update and the evaluation of Q, then CM_S returns a *busy* status to CM_R (rather than a *quiet* status) along with the query answer. Consider what happens when CM_R receives the answer. If the return status is *quiet* then there have

1	DB_R	send $\langle \Delta R, t_1 \rangle$ to CM_R
2	CM_R	receive $\langle \Delta R, t_1 \rangle$ from DB_R
3		send $\langle Q(\Delta R, S), t_1 \rangle$ to CM_S
4	CM_S	receive $\langle Q(\Delta R, S), t_1 \rangle$ from CM_R
5		submit $Q(\Delta R, S)$ to DB_S
6	DB_S	receive $Q(\Delta R, S)$ from CM_S
7		evaluate $Q(\Delta R, S)$
8		send query result to CM_S
9	CM_S	receive query result from DB_S at t_2
10		IF $(\exists t_3)(\Delta S @ t_3 \wedge t_1 < t_3 < t_2)$
		THEN send query result and $\langle busy, t_3 \rangle$ status to CM_R
		ELSE send query result and *quiet* status to CM_R
11	CM_R	receive query result and status from CM_S
12		IF status=*quiet* OR $(\nexists t_4)(\Delta R @ t_4 \wedge t_1 < t_4 < t_3)$
		THEN raise alarm if query result is non-empty
		ELSE restart protocol from step 2 with cumulative ΔR

Table 4: TRQ Protocol

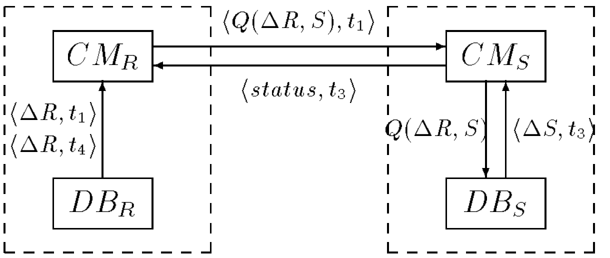

Figure 5: TRQ Protocol

been no relevant updates to S and the query has been evaluated over a true global state. If the return status is *busy* then there have been relevant updates to S. However, if R has not been updated further, then the query still has been evaluated over a true global state. In the case where there have been further updates to R, then the query may have been evaluated over a phantom state, and it must be reevaluated. Reevaluation takes place by restarting the protocol; for efficiency, the original and new updates to R are combined before restarting. Note that, provided that updates to R and S eventually cease, termination of the protocol is guaranteed since protocol restarts are triggered only by new updates. (In fact, updates at both sites are required to trigger a restart, so a given instance of the protocol is guaranteed to terminate as long as one site's updates cease.)

Safety of the TRQ protocol follows the same line of reasoning as safety for the DRQ protocol. We show that the TRQ protocol is accurate by arguing that every successful (i.e., not restarted) query evaluation is performed in a true global state. Hence, if an alarm is raised then there was a global state that violated the constraint:

- After an update ΔR at time t_1, t_1 is the first "candidate" global state for evaluation of Q. If evaluation does not produce a *busy* status, then Q has been evaluated in the state occurring at time t_1.
- If CM_S identifies a ΔS at time t_3 with $t_1 < t_3 < t_2$, then t_3 is taken as the next candidate global state. If there have been no further updates to R before t_3, then Q has been evaluated in the state occurring at time t_3.
- If CM_R identifies a new ΔR at time t_4 with $t_1 < t_4 < t_3$, then the process is restarted with t_3 as the next candidate global state.

The TRQ protocol also can be used with *time interval constraints*. Time interval constraints require a constraint to be evaluated within a certain time interval, instead of on a single global database state. This type of constraint relaxes the usual notion of consistency of integrity constraints, but is sufficient for many environments (such as those with low update frequency). When the TRQ protocol is used for time interval constraints, the protocol is restarted in step 12 of Table 4 only if $t_3 - t_4$ is greater than the time interval specified for the constraint.

3.4 The Semi-Timestamped Remote Query Protocol (SRQ)

The TRQ protocol assumes that the database systems can generate timestamps. A similar protocol can be used for the situation in which the database

1	DB_R	send ΔR within t_Δ to CM_R
2	CM_R	receive ΔR from DB_R at t_1
3		send $\langle Q(\Delta R, S), t_1 - t_\Delta \rangle$ to CM_S
4	CM_S	receive $\langle Q(\Delta R, S), t_1 - t_\Delta \rangle$ from CM_R
5		submit $Q(\Delta R, S)$ to DB_S
6	DB_S	receive $Q(\Delta R, S)$ from CM_S
7		evaluate $Q(\Delta R, S)$
8		send query result to CM_S
9	CM_S	receive query result from DB_S at t_2
10		IF $(\exists t_3)(\Delta S@t_3 \wedge t_1 - t_\Delta < t_3 < t_2)$
		THEN send query result and $\langle busy, t_3 \rangle$ status to CM_R
		ELSE send query result and $quiet$ status to CM_R
11	CM_R	receive query result and status from CM_R
12		IF status=$quiet$ OR $(\nexists t_4)(\Delta R@t_4 \wedge t_1 < t_4 \wedge t_4 - t_\Delta < t_3)$
		THEN raise alarm if query result is non-empty
		ELSE restart protocol from step 2 with cumulative ΔR

Table 5: SRQ Protocol

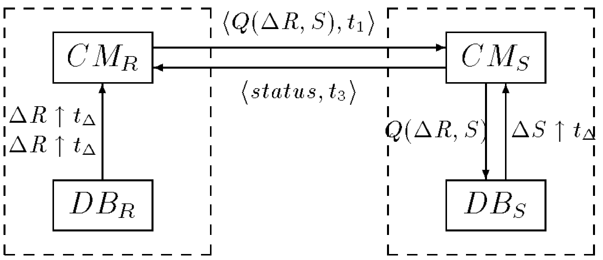

Figure 6: SRQ Protocol

systems cannot generate timestamps, but the constraint managers can.[4] To provide the accuracy property when timestamps are generated by the constraint managers, it is necessary to guarantee that delta notifications from database systems are always received within a certain maximum delay t_Δ [CGW96]. For this scenario we provide the *Semi-timestamped Remote Query Protocol* (SRQ). SRQ is specified in Table 5 and depicted in Figure 6. In the figure, $\Delta R \uparrow t_\Delta$ and $\Delta S \uparrow t_\Delta$ indicate delta notifications within delay t_Δ. The SRQ protocol is very similar to the TRQ protocol, except the determination of whether a phantom state may have been used must take into account the notification delays t_Δ. Like the TRQ protocol, the SRQ protocol is safe and accurate.

Both the TRQ and SRQ protocols require access to a synchronous global clock. With the availability of inexpensive "time services," this assumption does not appear to be highly restrictive. However, suppose that instead of a global clock we have a set of local clocks that are synchronized within a maximum drift t_D. Then the local clocks can be used instead of a global clock by incorporating t_D into our protocols, similarly to the way we have incorporated t_Δ to obtain the SRQ protocol from TRQ. In addition, we are investigating whether we can adapt our protocols so that *logical clocks* [BM93, Lam78] are sufficient.

3.5 The Transaction Remote Transaction Protocol (TRT)

The TRQ and SRQ protocols add timestamping to the DRQ protocol in order to achieve accuracy. This approach corresponds to one dimension in the protocol space of Figure 2. The next dimension we consider is transactions. Note that we are not proposing to add global transaction capabilities. Rather, we can exploit standard transaction capabilities provided by the local database systems. The *Transaction Remote Transaction Protocol* (TRT) is specified in Table 6 and depicted in Figure 7. In both the table and the figure, the enclosure of a query in square brackets indicates that the query is to be executed within its own local transaction, i.e., the query is effectively bracketed by **begin transaction** and **commit**.

The TRT protocol behaves as follows. When DB_R performs a delta notification, this notification occurs as part of the transaction τ that updated R, and τ does not commit until it receives a "release" message from the constraint manager. That is, transaction τ continues to hold its exclusive lock (*X-lock*) on R while the protocol runs. This approach requires that the database system is capable of performing a notification and waiting for an acknowledgment, all within

[4] As a general principle, we assume that the capabilities of the local database systems are likely to be fixed in advance (e.g., in the case of legacy systems), while constraint managers may permit modifications and enhancements for the purposes of implementing the protocols.

1	DB_R	send ΔR to CM_R, holding transaction X-locking R
2	CM_R	receive ΔR from DB_R
3		send remote transaction $[Q(\Delta R, S)]$ to DB_S
4	DB_S	receive $[Q(\Delta R, S)]$ from CM_R
5		execute $[Q(\Delta R, S)]$, S-locking S
6		commit transaction, releasing locks on S
7		send query result to CM_R
8	CM_R	receive result from DB_S
9		raise alarm if query result is non-empty
10		send release message to DB_R

Table 6: TRT Protocol

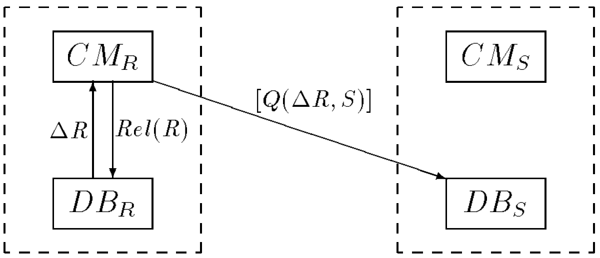

Figure 7: TRT Protocol

a single transaction. This capability is provided by most database systems supporting triggers or active rules [WC96]; see Section 2.3. Once CM_R has received the delta notification, it sends the appropriate query to DB_S, to be executed within its own transaction τ' at DB_S. During execution of the query, τ' will hold a shared lock (*S-lock*) on S. When the query answer is received by CM_R, an alarm is raised if the result is non-empty, and the pending transaction τ at DB_R is released.

Safety of the TRT protocol follows the same line of reasoning as for the other protocols. Accuracy of the TRT protocol results from the fact that every query is evaluated over a true global state (with respect to R and S). To see that only true global states are used, we can think of the bracketed query transaction τ' as being embedded within the pending transaction τ that generated the delta notification. In this way, the TRT protocol effectively emulates a distributed two-phase locking protocol, ensuring serializability across sites with respect to R and S [OV91].

Note that although the TRT protocol emulates global locking, the protocol does not rely on global transaction mechanisms of any kind.

Although the TRT protocol is relatively straightforward, and it satisfies both the safety and accuracy properties, it has two significant drawbacks:

1. Constraint manager processing is synchronous with the database system, unlike in the previous protocols. In TRT, the database system must wait for the constraint checking protocol to complete before it can commit its transaction and release its locks. This can be considered as a loss of autonomy, since progress on one site is dependent upon progress on the other site.

2. If R and S are updated concurrently then the risk of deadlock is relatively high. Since the TRT protocol emulates two-phase locking, the same deadlock cases arise as in centralized or tightly-coupled distributed databases. However, executing a remote transaction in a loosely-coupled federated environment may be much slower than transaction processing in a tightly-coupled or centralized environment, thereby increasing the chance of deadlock.

These drawbacks are addressed in an extended version of the TRT protocol called MDS, presented in Section 3.7. Another improvement to the TRT protocol is to use a *local test* to avoid the problems caused by remote transactions. This extension of the TRT protocol is described in Section 3.8. Below, we first present a slight variation on the TRT protocol.

3.6 The Indirect Transaction Remote Transaction Protocol (IRT)

If direct submission of remote transactions is not supported by the database systems, an *Indirect Transaction Remote Transaction Protocol* (IRT) is possible. In this protocol, the bracketed query is sent to the remote database system through its constraint manager. IRT bears the same relationship to TRT that IRQ bears to DRQ (recall Section3.2). The protocol is depicted in Figure 8.

Clearly, the IRT protocol has the same properties as the TRT protocol with respect to safety and accuracy.

1	DB_R	append change to ΔR
2		send Δ notification to CM_R
3	CM_R	receive Δ notification from DB_R
4		decide whether to run constraint check; if not, terminate
5		open transaction on DB_R, X-locking ΔR
6		submit $Q(\Delta R)$ to DB_R
7	DB_R	receive $Q(\Delta R)$ from CM_R
8		evaluate $Q(\Delta R)$
9		send query result to CM_R
10	CM_R	receive query result from DB_R
11		send $[Q(\Delta R, S)]$ to DB_S
12	DB_S	receive $[Q(\Delta R, S)]$ from CM_R
13		execute $[Q(\Delta R, S)]$, S-locking S
14		commit transaction, releasing locks on S
15		send query result to CM_R
16	CM_R	receive result from DB_S
17		raise alarm if query result is non-empty
18		submit $Delete(\Delta R)$ to DB_R
19	DB_R	perform $Delete(\Delta R)$
20	CM_R	commit transaction on DB_R, releasing X-lock on ΔR

Table 7: MDS Protocol

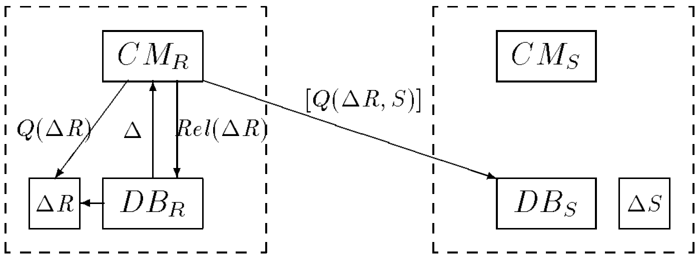

Figure 9: MDS Protocol

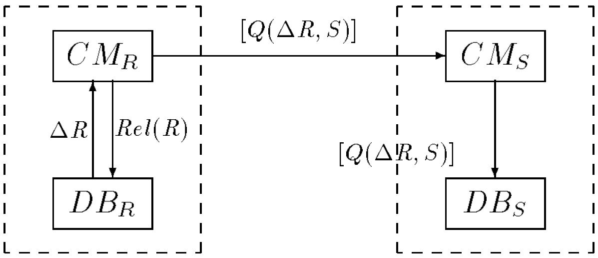

Figure 8: IRT Protocol

3.7 The Materialized Delta Set Protocol (MDS)

Our next protocol considers the *change logging* dimension of Figure 2. In this protocol, relevant changes are "logged" in special relations that we call *materialized delta sets*. Materialized delta sets allow us to develop a version of the TRT protocol in which constraint checking is decoupled from database execution, thereby addressing the drawbacks associated with the TRT protocol (discussed above). In addition, materialized delta sets enable variable, application-dependent granularities for constraint checking.

The *Materialized Delta Set Protocol* (MDS) is specified in Table 7 and depicted in Figure 9. It behaves as follows. When relation R is updated, DB_R appends the update to the materialized delta set ΔR. (Here, ΔR denotes an accumulated set of changes, not just an individual change or set of changes.) DB_R's append to ΔR must occur before the local transaction that updates R commits; however, no other actions need occur within the updating transaction. A notification Δ is sent to CM_R, indicating that an update occurred but not providing the actual delta set. When CM_R decides to proceed with constraint checking, it obtains an X-lock on ΔR for the duration of the protocol. CM_R then reads ΔR, and proceeds just as in the TRT protocol, executing a query $Q(\Delta R, S)$ within a transaction at the remote site. After receiving the query result and raising an alarm if appropriate, CM_R

$$
\begin{array}{ccc}
T_{C1}: & & T_{U1}: \\
\text{Hold(X-lock, } \Delta R) & \leftarrow & \text{Hold(X-lock, } R) \\
\text{Request(S-lock, } S) & & \text{Request(X-lock, } \Delta R) \\
\downarrow & & \uparrow \\
T_{U2}: & & T_{C2}: \\
\text{Hold(X-lock, } S) & \rightarrow & \text{Hold(X-lock, } \Delta S) \\
\text{Request(X-lock, } \Delta S) & & \text{Request(S-lock, } R)
\end{array}
$$

Figure 10: Deadlock in MDS protocol

1	DB_R	send ΔR to CM_R, holding transaction X-locking R
2	CM_R	receive ΔR from DB_R
3		submit $Q_L(\Delta R, R)$ to DB_R
4	DB_R	receive $Q_L(\Delta R, R)$ from DB_R
5		evaluate $Q_L(\Delta R, R)$
6		send query result to CM_R
7	CM_R	receive query result from DB_R
8		IF query result is non-empty THEN perform TRT protocol in same transaction

Table 8: LTT Protocol

erases the current contents of materialized delta set ΔR and releases its X-lock on ΔR.

As in the TRT protocol, accuracy is guaranteed because the protocol emulates distributed two-phase locking. However, unlike in TRT, local access to R is not entirely restricted while the protocol is executing. Local transactions that read R can execute concurrently with the protocol. Concurrent local transactions also can update R; however, because the protocol X-locks ΔR, such transactions cannot complete and commit until the protocol has finished. This behavior still represents some loss of local autonomy, but it is a significant improvement over the TRT protocol. In addition, the risk of deadlock is much lower with MDS than with TRT, although deadlock is not eliminated altogether. A deadlock cycle can exist involving two update transactions, T_{U1} and T_{U2} at DB_R and DB_S respectively, and two constraint checking transactions, T_{C1} and T_{C2} at CM_R and CM_S respectively. The cycle is illustrated in Figure 10. Note that the occurrence of a four-element deadlock cycle is far less likely than the two-element cycle that can occur with the TRT protocol.

3.8 The Local Test Transaction Protocol (LTT)

The last dimension of Figure 2 we explore is the use of *local tests*. Consider a delta notification ΔR. A local test is a query, $Q_L(\Delta R, R)$, such that if the query

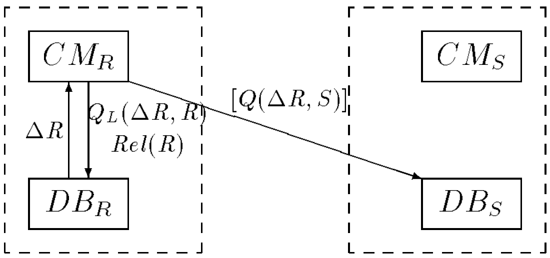

Figure 11: LTT Protocol

result is empty, then the result of the global constraint checking query $Q(\Delta R, S)$ is guaranteed to be empty as well. The test is "local" because it involves ΔR and R only, and does not require access to S. Hence, a local test can be used to avoid the many disadvantages of executing remote queries. Unfortunately, local tests are generally *conservative*, so if a local test fails it still becomes necessary to issue a remote query. A considerable theory of local tests has been developed in [GW93, GSU94]. Here we show how that theory can be put into practice in the context of constraint checking protocols for federated databases.

As an example of a constraint with a local test, recall factored constraint C_3^B from Section 2.5:

$$C_3^B \; : \; \Pi_{Physician} \Delta^+ Patients_B - \Pi_{PhN} Physicians$$

This constraint specifies that new patient records must match existing physician records. If all new patient records match physicians in existing patient records, then this local condition is sufficient to guarantee validity of the constraint without examining the physician records (under the assumption that the existing patient records satisfy the constraint). The local test is expressed as the following query, where the test succeeds if the query result is empty:

$$C_{3_L}^B \; : \; \Pi_{Physician} \Delta^+ Patients_B - \Pi_{Physician} \Delta^0 Patients_B$$

The *Local Test Transaction Protocol* (LTT) is specified in Table 8 and depicted in Figure 11. The protocol is very similar to TRT, except before issuing the remote transaction $Q(\Delta R, S)$, the local test $Q_L(\Delta R, R)$ is evaluated. If the local test succeeds, then the protocol terminates successfully with no remote activity. Note that the local test must be evaluated within the same transaction in which the update occurred in order to behave correctly [GSU94, ZGH95], so a materialized delta set approach cannot be applied here. The safety and accuracy of the LTT protocol follows from the correctness of local tests [GW93, GSU94],

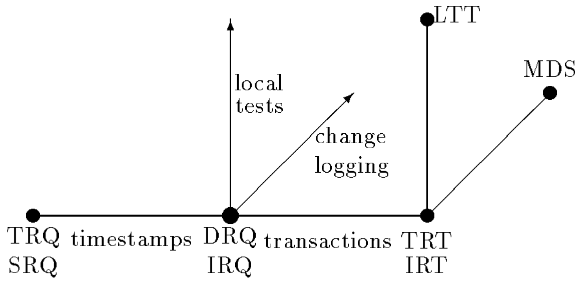

Figure 12: Protocols in the protocol space

and from the safety and accuracy of the TRT protocol.

4 Analyzing the Family of Protocols

In Section 3 we developed a family of constraint checking protocols along the four dimensions introduced in Figure 2. In this section we analyze the protocols. First, we consider the protocols in the context of the four-dimensional space to see how complete the family is. Then we compare the characteristics and costs of the various protocols. This comparison yields a "consumer report" that can be used to select the most appropriate protocol for a specific federated database environment. Finally, we discuss how the protocols can be extended to handle constraints spanning more than two databases.

4.1 The Protocol Space

Figure 12 places the protocols developed in Section 3 into their appropriate positions in the four-dimensional space of Figure 2. Although several combinations of dimensions have been explored, not every possibility has been considered. The four dimensions yield sixteen different possible protocols, listed in Table 9.

We first point out three potentially useful protocols not discussed in Section 3. These protocols combine features from our protocols in a relatively straightforward way.

1. Change logging can be used with the DRQ or IRQ protocol to produce a *Cumulative Remote Query* (CRQ) protocol. CRQ accumulates changes in a materialized delta set to allow variable granularities of constraint checking. When a remote query is posed, it uses the current contents of the materialized delta set.
2. Timestamps and change logging can be combined to obtain a *Timestamped Materialized Delta Set* (TMD) protocol. In this protocol, materialized delta sets containing timestamps are used in order to allow variable granularities of constraint checking and to keep track of cumulative updates for protocol restarts.
3. Timestamps, transaction mechanisms, and local tests can be combined to obtain a *Local Transaction Global Timestamped* (LTG) protocol. The LTG protocol resembles the LTT protocol, except when a local test fails, timestamping is used rather than transactions for the global query (to achieve more autonomy).

Some protocols appearing in Table 9 have an inappropriate combination of features: Local tests cannot be used without local transactions. (Otherwise, the query for the local test might be evaluated in an erroneous state [GW93, ZGH95].) This eliminates lines 9–10 and 13–14. Combining timestamps and transactions is "overkill" in the case where local tests are not used (eliminating lines 4 and 8). Finally, the combination of transactions, change logging, and local tests, with or without timestamping, yields rather complex protocols that we have yet to explore (lines 15–16).

4.2 Comparing the Protocols

The various protocols in the family can be compared with respect to: (1) what they require, and (2) what they deliver. The requirements of the protocols concern the functionality of the systems within the federation and the functionality of the interfaces between systems. Relevant requirements are: an interface for executing remote queries, availability of local database transactions, an order-preserving interface between each database system and its corresponding constraint manager, and a global clock. The protocols "deliver" certain properties: safety, accuracy, asynchrony, and flexible granularity for constraint checking. Table 10 summarizes the requirements and properties of the protocols discussed in Section 3.

The requirements and properties discussed so far are *static* characteristics of the protocols. The protocols also can be compared in terms of their *dynamic* characteristics, specifically their execution costs. We provide only a preliminary cost analysis here; as future work we plan to expand our analytical model as well as conduct empirical experiments. We distinguish three ingredients in the cost of a constraint checking protocol:

	timestamps	transactions	change logging	local tests	Protocol
1					DRQ, IRQ
2	X				TRQ, SRQ
3		X			TRT, IRT
4	X	X			*inappropriate*
5			X		CRQ (*see text*)
6	X		X		TMD (*see text*)
7		X	X		MDS
8	X	X	X		*inappropriate*
9				X	*inappropriate*
10	X			X	*inappropriate*
11		X		X	LTT
12	X	X		X	LTG (*see text*)
13			X	X	*inappropriate*
14	X		X	X	*inappropriate*
15		X	X	X	*to be considered*
16	X	X	X	X	*to be considered*

Table 9: Protocol space filled in

	requirements				properties			
	remote query interface	local trans-actions	order-preserving interface	global clock	safety	accuracy	asynch-rony	flexible granu-larity
DRQ	X				X		X	
IRQ					X		X	
TRQ			X	X	X	X	X	
SRQ			X	X	X	X	X	
TRT	X	X			X	X		
IRT		X			X	X		
MDS	X	X			X	X		X
LTT	X	X			X	X		

Table 10: Requirements and properties of the protocols

Local messages are messages between a database system and its constraint manager. We distinguish between notification-only messages and messages that convey data.

Remote messages are messages between a constraint manager and a different constraint manager or a remote database system. All remote messages convey data.

Database operations are operations performed by a local database system, where an operation can be either a query, a database modification, or a transaction commit.

The cost functions for the protocols from Section 3 are shown in Table 11. Costs are given for a single delta notification, i.e., for one local update transaction triggering the protocol. From the analysis in Table 11 some interesting preliminary observations can be made:

Although the TRQ protocol has better properties than the IRQ protocol, and TRQ appears more complex, the cost functions of the two protocols are the same. This is explained by the fact that, although TRQ may need to iterate, each iteration "consumes" a delta set that would have triggered an independent instance of IRQ. (In other words, a protocol restart in TRQ incurs no additional cost.) In practice, the actual cost of TRQ may be slightly higher than IRQ since TRQ must generate and communicate timestamps, and because restarts in TRQ may cause larger data sets to be transmitted.

The cost of the MDS protocol can be "tuned" by varying the granularity factor g, i.e., the number of delta notifications before constraint checking is initiated. This allows an application to establish a constraint checking policy that balances its integrity requirements against the available resources.

The efficiency of the LTT protocol depends heavily on the probability p that a local test succeeds; if p is low, the LTT protocol has higher cost than the TRT protocol. The actual value for p is strongly

	local msgs per Δ	remote msgs per Δ	database ops per Δ	total cost (simplified)
DRQ	λ_D	2ρ	δ_Q	$(\lambda + 2\rho + \delta)\nu$
IRQ	$3\lambda_D$	2ρ	δ_Q	$(3\lambda + 2\rho + \delta)\nu$
TRQ	$3\lambda_D$	2ρ	δ_Q	$(3\lambda + 2\rho + \delta)\nu$
SRQ	$3\lambda_D$	2ρ	δ_Q	$(3\lambda + 2\rho + \delta)\nu$
TRT	$\lambda_D + \lambda_N$	2ρ	$\delta_Q + \delta_C$	$(2\lambda + 2\rho + 2\delta)\nu$
IRT	$3\lambda_D + \lambda_N$	2ρ	$\delta_Q + \delta_C$	$(4\lambda + 2\rho + 2\delta)\nu$
MDS	$\lambda_N + (2\lambda_D + \lambda_N)/g$	$2\rho/g$	$\delta_U + (2\delta_Q + \delta_U + \delta_C)/g$	$(\lambda + \delta + (3\lambda + 2\rho + 4\delta)/g)\nu$
LTT	$3\lambda_D + \lambda_N$	$2(1-p)\rho$	$(2-p)\delta_Q + \delta_C$	$(4\lambda + 2(1-p)\rho + (3-p)\delta)\nu$
ν	number of update transactions			
λ_N	cost of a local notification-only message			
λ_D	cost of a local data-conveying message			
ρ	cost of a remote message			
δ_Q	cost of a local database query operation			
δ_U	cost of a local database update operation			
δ_C	cost of a local database commit operation			
g	granularity factor for MDS protocol			
p	probability of local test succeeding in LTT protocol			

Table 11: Cost functions for the protocols

application-dependent. In the example presented in Section 3.8, we expect that p would be close to 1, making LTT a very attractive protocol.

These observations are based on our relatively simple cost model. Deeper observations may be possible by refining the cost model, e.g., to more precisely capture message sizes. In addition to refining the cost model, in follow-on work we plan to perform analyses with varying parameter values in order to further understand the relative costs of the protocols in various environments.

4.3 Multi-Site Constraints

Our constraint checking protocols in Section 3 have been developed for constraints involving exactly two databases. It is our experience that two-site constraints predominate in practice. Furthermore, even when a constraint involves more than two databases, often it can be "split" into multiple constraints, each involving exactly two databases. For example, suppose our hospital application (Section 2.5) involved three hospitals instead of two. The constraint specifying that each patient can be registered at only one hospital (constraint C_1) now involves three sites. However, instead of enforcing the three-site constraint directly, we can enforce the following set of two-site constraints:

$$C_1^1 : \ Patients_A \ltimes_{SSN=SSN} Patients_B$$
$$C_1^2 : \ Patients_B \ltimes_{SSN=SSN} Patients_C$$
$$C_1^3 : \ Patients_A \ltimes_{SSN=SSN} Patients_C$$

Together, these constraints imply the desired three-site constraint.

There are, however, cases where we may need to handle multi-site constraints: We may wish to handle constraints such as the example above directly, rather than separately handling a set of equivalent constraints (which may be exponential in number). Furthermore, some constraints cannot be split as illustrated above. Suppose in our three-hospital example that patients may register at two hospitals but not three. On insertions into relation $Patients_A$, the constraint checking protocol must evaluate the following incremental query:

$$\Pi_{SSN}(PatientsA^+ \ltimes_{SSN=SSN} PatientsB) \cap$$
$$\Pi_{SSN}(PatientsA^+ \ltimes_{SSN=SSN} PatientsC)$$

When a delta notification for $Patients_A$ is received by the constraint manager at site S_A, this query can be evaluated easily using the DRQ or IRQ protocol—the constraint manager simply submits two remote queries (one to S_B and one to S_C) and intersects the results. Safety of the protocol is guaranteed by the same reasoning given in Section 3.1. Timestamp-based protocols TRQ and SRQ also can be extended in a straightforward way to handle multi-site constraints: A restart is necessary whenever a *busy* status is received from any of the remote sites and a local update has occurred; safety and accuracy follow from the results in Section 3.3.

It is less straightforward to construct safe and accurate versions of the transaction-based protocols (such as TRT) for the multi-site case. These protocols emulate two-phase locking, which requires coordinating local transactions across multiple sites. A two-phase "handshake" protocol could be used for synchronization, similar to distributed two-phase commit. Alternatively, a mechanism could be used where remote queries are "chained" from one site to the next (emulating multiple nested transactions). Unfortunately, both solutions incur a considerable loss of local autonomy.

These initial observations lead us to believe that non-transaction-based protocols are most suitable for constraints involving more than two sites. As future work we plan to explore the issues more carefully, and to develop protocols especially suited to the multi-site case.

5 Conclusions and Future Work

We have described a family of constraint checking protocols for federated database systems. The protocols have been developed by starting with a straightforward basic protocol, then improving upon the basic protocol by extending it along a number of dimensions.

We have isolated two properties of constraint checking protocols that are of primary importance: safety and accuracy. All of our protocols are safe, meaning that they are guaranteed to detect every constraint violation. A protocol is accurate if it does not raise "false alarms," i.e., it only detects true constraint violations. Obtaining accuracy requires additional mechanisms, as seen in our more complex protocols. In addition to varying in terms of accuracy, our protocols also vary in terms of their requirements of the underlying systems, their level of asynchrony, their flexibility, and their execution costs. We are quite certain that no one protocol will be suitable for all federated database scenarios. By providing a family of alternatives, one of our protocols can be chosen—and perhaps tailored—for a particular environment or application.

By formalizing the relevant concepts, and by identifying and analyzing a suite of protocols, this paper provides a sound basis for the problem of integrity constraint checking in federated databases. The work presented in this paper can be extended in a number of directions:

- The model for analyzing the execution costs of the protocols can be further elaborated. So far we have identified the cost "ingredients," and we have compared the protocols according to these ingredients. The next steps are to introduce and vary parameter values to understand relative costs in different environments, and to drop some of the simplifying assumptions we made for the initial cost analysis.
- The issue of *constraint repair* when violations occur requires further study. Even in traditional centralized databases, constraint repair is an important topic of current research (e.g., [CFP94, GL93]). In federated databases the problem is even more difficult. It has to be determined which of the protocols can be extended to incorporate constraint repair, and what the characteristics of these extended protocols are with respect to constraint maintenance properties, system requirements, and execution costs.
- The protocols identified by Table 9 that we have not studied but that may be applicable for certain environments merit further study. The same holds for protocols for multi-site constraints (expanding on the ideas of Section 4.3), and for protocols for handling constraints that do not have a straightforward incremental form.
- The use of *logical clocks* [BM93, Lam78] instead of absolute timestamps is an interesting modification to be studied for those protocols that currently rely on synchronized time services.

Acknowledgments

We are grateful to Stefano Ceri, Hector Garcia-Molina, and the rest of the Stanford Database Group for useful feedback on this work.

References

[BGM92] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Advances in Database Technology—EDBT '92, Lecture Notes in Computer Science 580*. Springer-Verlag, Berlin, 1992.

[BGM92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *VLDB Journal*, 1(2), 1992.

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Procs. ACM SIGMOD Int.*

Conf. on Management of Data, Washington, D.C., 1986.

[BM93] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*. ACM Press, New York, 1993.

[CFP94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Trans. on Database Systems*, 19(3), 1994.

[CGW96] S. Chawathe, H. Garcia-Molina, and J. Widom. A Toolkit for Constraint Management in Heterogeneous Information Systems. In *Procs. 12th Int. Conf. on Data Engineering*, New Orleans, LA, 1996.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1), 1985.

[CW93] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Procs. 19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland, 1993.

[DD95] L. Do and P. Drew. Active database management of global data integrity constraints in heterogeneous database environments. In *Procs. 11th Int. Conf. on Data Engineering*, Taipei, Taiwan, 1995.

[Elm91] A. Elmagarmid (ed.). *Special Issue on Unconventional Transaction Management,* Data Engineering Bulletin 14(1), 1991.

[GA90] P.W.P.J. Grefen and P.M.G. Apers. Parallel handling of integrity constraints on fragmented relations. In *Procs. Int. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, 1990.

[GA93] P.W.P.J. Grefen and P.M.G. Apers. Integrity control in relational database systems – an overview. *Journal of Data & Knowledge Engineering*, 10(2), 1993.

[GL93] M. Gertz and U.W. Lipeck. Deriving integrity maintaining triggers from transition graphs. In *Procs. 9th Int. Conf. on Data Engineering*, Vienna, Austria, 1993.

[Gre93] P.W.P.J. Grefen. Combining theory and practice in integrity control: a declarative approach to the specification of a transaction modification subsystem. In *Procs. 19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland, 1993.

[GSU94] A. Gupta, Y. Sagiv, J.D. Ullman, and J. Widom. Constraint checking with partial information. In *Procs. 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1994.

[GW93] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *Procs. ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1993.

[GW94] P. Grefen and J. Widom. Integrity constraint checking in federated databases. *Memoranda Informatica*, 94-80, Department of Computer Science, University of Twente, The Netherlands, 1994.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications ACM*, 21(7), 1978.

[Mon95] D. Montesi and R. Torlone. A transaction transformation approach to active rule processing. In *Procs. 11th Int. Conf. on Data Engineering*, Taipei, Taiwan, 1995.

[Nic82] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18, 1982.

[OV91] T. Oszu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Qia89] X. Qian. Distribution design of integrity constraints. In L. Kerschberg, editor, *Expert Database Systems—Procs. from the 2nd Int. Conf.*. Benjamin/Cummings, Redwood City, California, 1989.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Engineering*, 3(3), 1991.

[RCB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation monitoring for active databases. In *Procs. 15th Int. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, 1989.

[RSK91] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, 24(12), 1991.

[SV86] E. Simon and P. Valduriez. Integrity control in distributed database systems. In *Procs. 19th Int. Conf. on System Sciences*, Hawaii, 1986.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, California, 1996.

[ZGH95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Procs. ACM SIGMOD Int. Conf. on Management of Data*, San Jose, California, 1995.