FACE: A Flexible Approach for Constraint Enforcement and Compensation of Constraint Violations in Federated Databases

Myra Spiliopoulou[1] and Stefan Conrad[2]

[1] Institut für Wirtschaftsinformatik, Humboldt-Universität zu Berlin
myra@wiwi.hu-berlin.de, http://www.wiwi.hu-berlin.de/~myra
[2] Institut für Technische Informationssysteme, Otto-von-Guericke Universität Magdeburg
conrad@iti.cs.uni-magdeburg.de, http://wwwiti.cs.uni-magdeburg.de/~conrad

Abstract. The autonomy of the participants in a database federation makes the maintenance of consistency on the shared data a hard problem. A constraint mechanism at the federation level is often proposed as an elegant and effective solution. The introduction of active components is a powerful and natural way to enforce constraints over a federation, but is often not applicable due to the autonomy of the participants.
We propose a model which supports constraint enforcement by a suite of autonomy-preserving mechanisms. Those mechanisms exploit the functionality of triggers and active components, if provided by the participants. For participants without such functionality, violations are detected by transaction supervision or by querying dubious data. A compensator module undertakes the re-establishment of consistency. Inconsistencies that cannot be resolved due to the autonomy of the participants are hidden from the federation users by a view mechanism.

1 Introduction

The current development of database middleware products shows that there is an increasing need for multidatabases, and especially database federations, the participants of which are autonomous. Much research has been done in this area to resolve problems of schema integration and global transaction management. Another emerging issue is the enforcement of global integrity constraints. Few results are available on this topic, mainly because constraint enforcement reduces the autonomy of the federation members. Moreover, the heterogeneity of the members with respect to the local integrity subsystems, if any, is another impediment for a simple and efficient global approach to constraint enforcement.

In this study we present FACE, a flexible approach to global constraint enforcement in federated databases. On the one hand, FACE relies on active rule mechanisms for global constraint enforcement and for detecting local operations which potentially violate global integrity constraints. On the other hand, we provide additional mechanisms for coping with the heterogeneity and autonomy of the federation members. Hence, once a multidatabase member is willing to

participate in global constraint definition, its autonomy is less reduced due to constraint enforcement than by other approaches.

We first present an overview of the architecture of FACE, and we classify federation participants according to the support they provide for constraint enforcement. In Section 3 we describe then the components responsible for constraint specification and enforcement on the global and the local level. In Section 4, we further analyze the mechanism for the compensation of constraint violations, which plays a crucial role in FACE. The view mechanism envisaged to hide objects that do not conform to constraints from global applications is discussed in Section 5. In Section 6, we discuss related approaches. The last section summarizes the main properties of FACE and points out future work.

2 Outline of the FACE Architecture

FACE is designed for a federated database ("FDB") environment, the local DBMS ("LDBMS") of which are autonomous in their organization and may support local applications. Consequently, queries and updates are issued by both local applications accessing directly the LDBMSs and by global applications served by the FDB manager.

The FDBS box in Fig. 1 depicts the modules comprising the federation layer of FACE. The services provided include query optimization, global transaction management and constraint specification and enforcement.

2.1 The FDB layer

The FDB layer serves global applications and maintains global integrity constraints over the LDBMS participants. Constraints are stored in a rule base and administered by the "Global Rule Manager". Those constraints can be eventually violated by the updates performed by the global applications. FACE attempts to prevent constraint violations. If this is not possible due to the autonomy of the LDBMSs, inconsistencies are compensated or, in the worst case, hidden from the global application users.

The requests of the global applications, the "global transactions", are forwarded to the Transaction Manager, which monitors update and read-only transactions (queries). Updates are decomposed by the Transaction Manager into operations for each LDBMS involved; queries are decomposed by the optimizer. We term the subqueries and updates thus produced as "local transactions".

The local transactions are organized as components of a global transaction in the conventional way for federated databases [SW91, BGS92]. Global constraint violations caused by global transactions are identified by the Global Rule Manager and prevented by rolling the transactions back. Constraint violations caused by the individual local transactions must be handled at the LDBMS level, returning to the Transaction Manager enough information to decide whether the global transaction should be rolled back or not.

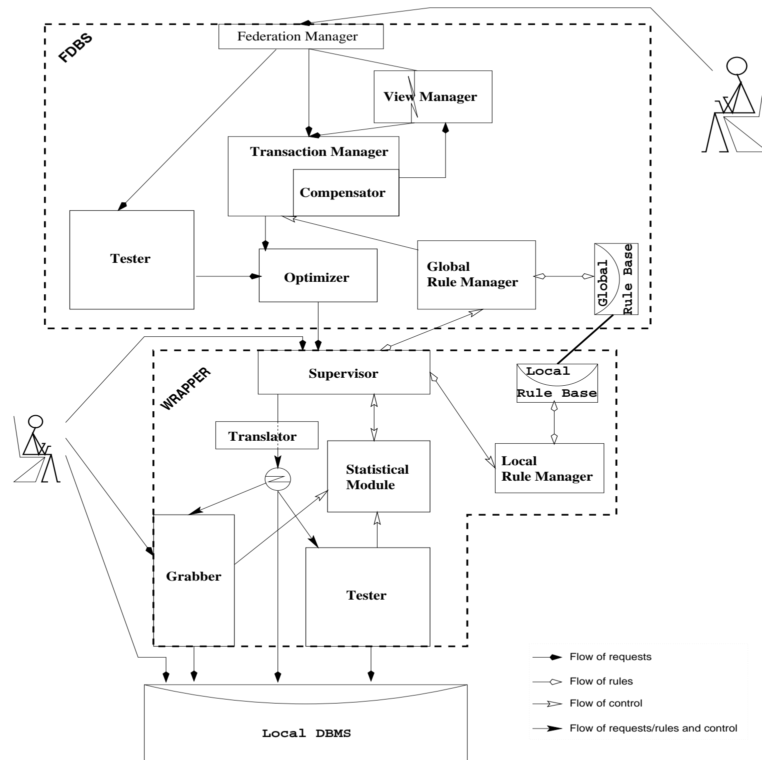


Fig. 1. The Architecture of FACE

2.2 A classification of LDBMSs.

Each LDBMS participant has a Wrapper supermodule attached to it, which is responsible for the enforcement of constraints and the detection of inconsistencies at the local database level.

The Wrapper of each LDBMS maintains the constraints pertinent to the LDBMS in a Local Rule Base administered by the "Local Rule Manager". Essentially, this rule base contains a subset of (derivatives of) the rules maintained in the Global Rule Base. However, on reasons of efficiency in local processing and of reliability in case of network failures, it is desirable to store in each Wrapper all the constraints relevant to the LDBMS. Since constraints are not updated frequently, the problem of maintaining the Local Rule Base identical to the corresponding part of the Global Rule Base is a simpler form of the problem of updates on replicated data (e.g. cf. [MT93]).

We categorize LDBMSs on the basis of their functionality in terms of constraint enforcement:

- An *active* LDBMS processes the local transactions enforcing the constraints concerning it locally.

- A *quasi-active* LDBMS possesses a trigger-based mechanism which can enforce the constraints of the local transactions, provided that they are translated into commands understood by the LDBMS parser.
- A *passive* LDBMS does not guarantee constraint enforcement.

Orthogonally, we classify LDBMSs permitting or supporting the enforcement of the constraints posed by the federation as "cooperative" LDBMSs. LDBMSs disallowing this are termed as "closed".

The Wrapper over a cooperative active or quasi-active LDBMS simply needs to translate the constraints of the local rule base to the form supported by the LDBMS. Thereafter, transactions towards this LDBMS are simply forwarded to it; the (quasi-)active mechanism is responsible for verifying whether a transaction violates some constraint and roll it back. For local transactions towards a passive LDBMS, constraint enforcement is undertaken by the modules of the Wrapper, as described in Section 3.

3 Analysis of the FACE Architecture

We now present the architecture of FACE in detail, discussing how constraints should be specified and how violations can be prevented or detected and compensated.

3.1 Constraint Specification

Constraints at the FDB level. For the specification of integrity constraints at the FDB level we assume a declarative language, e.g. based on predicate logic. In order to enforce them and to detect inconsistencies, constraints must be internally transformed to rules. Those rules are processed by the "Global Rule Manager" at the FDB level and by the "Local Rule Manager" in each Wrapper.

We now introduce a running example, in which we specify global integrity constraints and discuss the interactions of the FDBS with the Wrappers.

Example 1. Let ENTERPRISE be a federated database modelling an enterprise with n departments at different geographic locations. Each department D_i has its own database DEPARTMENT_i, containing a class `Personnel` on the employees of the department. Assuming that heterogeneities have been appropriately resolved, a class `Personnel` appears at the FDB level, containing data on the employees of the enterprise.

Department D_2 is dedicated to research. It keeps information on projects and their members in the classes `Project` and `Member`. Each project has at least one member not working in D_2. The class `Project` appears intact in ENTERPRISE, while the class `ENTERPRISE.Member` is similar to the class `DEPARTMENT_2.Member` after projecting out data already in `DEPARTMENT_2.Personnel`.

The information on the employees of the enterprise and on their activities within the projects should be consistent at the ENTERPRISE level. Therefore,

interdatabase dependencies must be enforced. In particular, we consider a constraint *C-1*, stating that a member of a project is an employee of the enterprise. Using first-order logic, this constraint can be expressed as:

$$\forall e, p : member(e, p) \rightarrow Personnel(e)$$

meaning that for each project p and person e working on it, e should appear in the `Personnel` class of ENTERPRISE. The respective formula explicitly referring to the local schemata can be expressed as:

$$\forall e, p : DEPARTMENT_2.member(e, p) \rightarrow Personnel(e)$$

where an object in `Personnel` belongs to some class `DEPARTMENT_i.Personnel`.

Constraint *C1* is violated if an employee participating in a project is removed from the `Personnel` class, or if a person not belonging to the personnel is inserted in the `Member` class. An active rule of the Global Rule Base handling the deletion case would be as follows:

define rule *C1-global-del-rule*
on $DEPARTMENT_i.delete_personnel(e)$
if $\exists DEPARTMENT_2.Member(e)$
do $DEPARTMENT_2.delete_member(e)$

An active rule of the Global Rule Base handling the insertion case could be:

define rule *C1-global-ins-rule*
on $DEPARTMENT_2.insert_member(e)$
if $\nexists Personnel(e)$
do $ask(i); DEPARTMENT_i.insert_personnel(e)$

where the interaction with the user is obvious: although FACE makes much effort in resolving inconsistencies automatically, not all constraint violations can be remedied without information from the user.

An alternative that could be processed without interaction with the user is an active rule aborting the transaction that issued the insert operation:

define rule *C1-global-ins-rule-A*
on $DEPARTMENT_2.insert_member(e)$
if $\nexists Personnel(e)$
do *abort*

Constraints at the LDBMS level. Enforcement of constraints in a LDBMS is initiated by the Wrapper. In a cooperative active or quasi-active LDBMS , constraints are simply translated into a set of instructions in the LDBMS's internal language. Constraint enforcement in passive and in closed LDBMSs is conducted by the *Supervisor* module and performed by the *Grabber* or *Tester* in cooperation with the Local Rule Manager, as explained hereafter.

Example 2. The Local Rule Base of Example 1 could contain an active rule detecting the operation $insert_member(\cdot)$ and sending a notification to the FDB level (via the Supervisor):

```
define rule DEPARTMENT_2.ins-rule
on          insert_member(e)
do          send_notification("DEPARTMENT_2","insert_member(·)",e)
```

Support for local applications. The full functionality of FACE is available to the users of the global applications. This covers constraint specification and enforcement, as well as optimization and transaction management. Local applications accessing directly the LDBMS are still allowed, though.

Local applications may specify constraints, if the LDBMS is an active or quasi-active one. Those constraints can be incorporated into the constraint management of FACE, by adding them to the Rule Base. Then, the local application communicates with the LDBMS via the Wrapper. The application's transactions are passed through the Supervisor, which is responsible for forwarding transactions and rules to the Translator. This translation does not concern the commands towards the LDBMS, which are in native code anyway, but the incorporation of the rules into the commands. Hence, local applications are not subject to global transaction control but exploit the constraint management of FACE. Constraint violations occuring locally can be prevented or compensated by the modules of the Wrapper, as described hereafter.

3.2 Constraint Violation Prevention – The Grabber

The Grabber module acts as an intermediary between the LDBMS and *all* applications accessing it. It eavesdrops update transactions and, in cooperation with the Local Rule Manager, prevents constraint violations by rolling back transactions not satisfying the constraints. Rolling back a transaction implies interferring with the execution of the transaction: after the execution of all the updates in the transaction and prior to the commit operation, the control should be returned to the Grabber, who decides whether the transaction should be committed or aborted.

A close coupling of the Grabber with the LDBMS's transaction mechanism is necessary to edit the transactions encoded for the LDBMS . Transactions expressed in 3GL programs cannot be monitored in that way. A solution could be the redefinition of the LDBMS commit protocol, so that the interference of the Grabber be perceived as the second phase of a two-phase commit.

If the Grabber is permitted to view the transactions' content but cannot affect the execution of the transactions themselves, the objects possibly violating the rules can still be detected before the violation occurs. This can be achieved by reading the LDBMS undo-log and keeping track of ongoing and committed transactions. This information is precious to the Transaction Manager, which will attempt to undo the effects of the local transaction that caused the violation. To obtain this information, the following possibilities should be considered:

- The Grabber could maintain an undo-log, which is actually a replica of the transaction log in the LDBMS. This has the disadvantage of very high space demand.

– In cooperation with the Local Rule Manager, suspect objects should be copied in a cache before the transaction is launched, and serve as a partial transaction log, if the transaction should be undone.
– The Grabber could attempt to reclaim the original contents of the updated objects from other objects related to them. This approach may be used to restore foreign key values and objects/attributes explicitly specified to have identical contents.

Beyond cachind data needed to restore the federation in a consistent state, the Grabber also collects statistical information on constraint violations actually taking place. This information is needed to decide whether the processing delays inevitably caused by the Grabber's intervention are necessary, or the Tester should be switched instead.

Example 3. The active rule *DEPARTMENT_2.ins-rule* in Example 2 instructs the Grabber to detect occurrences of the operation *insert_member* and to notify the Transaction Manager about it. This rule can be expanded as follows, to permit the Grabber to abort the transaction:

define rule *DEPARTMENT_2.ins-extended-rule*
on $insert_member(e)$ then $prepare_to_commit$
do $send_notification(\text{“}DEPARTMENT_2\text{”}, \text{“}insert_member(\cdot)\text{”}, e);$
$receive_action(a);$
$send_action(\text{“}insert_member(\cdot)\text{”}, e, a)$

In this rule, the Grabber notifies the FDB level on the operation going on and receives an instruction in the form of an “ABORT” or “COMMIT” action. This action is sent to the LDBMS. □

The applicability of the Grabber is very limited: it can be used in cooperative LDBMSs only; its development cost is high because it depends on the actual DBMS implementation; its intervention may be unacceptable to some applications on reasons of security and data protection. We intend to implement a Grabber module, but we expect its usefulness to be exhausted in experiments: we need it to compare its performance with the Tester, but we do not expect that the LDBMSs of a non-experimental federation will permit its usage.

3.3 Constraint Violation Detection – The Tester

The Tester module is used to issue queries towards the LDBMSs (i) to verify the violation of a constraint by a *running* transaction and (ii) to detect constraint violations by *already committed* transactions.

Assisting in the prevention of constraint violations. In a cooperative LDBMS, an active rule triggered by a local transaction often instructs the Grabber to notify the Transaction Manager at the FDB level, as in Example 3 above.

When the Transaction Manager receives this notification, it consults the Global Rule Manager, which identifies the corresponding global active rules and provides the operations necessary to cross-check whether a constraint violation

will indeed occur after committing this transaction. Those cross-checking operations are queries towards other LDBMSs of the federation.

Example 4. Continuing Example 3, let e be the object being inserted by the transaction being notified. The global rule triggered is *C1-global-ins-rule* as described in Example 2. To test the if-part of the rule body, the Tester should issue the following query in each department.

```
SELECT i FROM DEPARTMENT_i.Personnel WHERE key = e.key
```

Detecting constraint violations. The detection of already violated constraints is performed periodically: the Wrapper of a closed LDBMS and the Transaction Manager of a federation with some closed participants activates the Tester to identify inconsistencies introduced by previously committed local transactions.

Example 5. A query identifying objects not satisfying constraint *C-1* would be:

```
SELECT * FROM DEPARTMENT\_2.Member
WHERE key NOT IN
          (      SELECT key FROM DEPARTMENT\_1.Personnel
           UNION ...
           UNION SELECT key FROM DEPARTMENT\_n.Personnel)
```

As shown in Fig. 1, a Tester is attached to the FDB level as well. Since global constraints do not always translate to local ones and since some LDBMSs may be of the closed type, federation queries are occasionally necessary to identify constraint violations not echoed in either of the affected LDBMSs.

Transforming rules into queries. The major responsibility of the Tester is to generate a "query suite" to test each rule in the rule base, i.e. a set of queries towards the LDBMSs, which should be executed as a read-only transaction, to prevent updates on the data being queried. For the specification of query suites, we initially consider an offline translation: whenever a new constraint is introduced, a query suite testing for its violation should also be specified. Automating the query generation process will be later considered, though.

Optimizing query suites. A query suite must be optimized, so as to minimally imped the normal operation of the LDBMSs. Federated optimization concerns query decomposition, selection of the appropriate subquery granularity towards each LDBMS, ordering of subqueries and specification of execution sites for joins involving more than one LDBMS.

If the query suites are created offline, they can also be optimized offline, so that only an optimal query execution plan is stored for execution. This plan must be updated to reflect changes in the LDBMS physical and logical schema. Hence, the optimization problem for query suites has the following aspects:

1. Optimizing a federated database query: a cost metric is needed to express the optimization criterion, of minimizing the impact of the test queries on

conventional application queries. Parameters like the query/update rates, the duration of updates and the frequency of violations detected by the Tester should be incorporated in this metric.
2. Updating of the optimized query execution plan to remain efficient in the presence of changes in the LDBMSs. The parameters affecting the quality of a plan, such as: (i) creation of index structures, (ii) creation or removal of large numbers of objects, (iii) massive updates of attribute values, (iv) schema modifications, should be modelled. A mechanism detecting such changes in the LDBMSs should be established. Since the autonomous LDBMSs are not obliged to provide such meta-information, calibrating mechanisms can be considered to obtain it [DKS92, ZL94b, ZL94a, GST96, Spi96].
3. Multiquery optimization for queries belonging to the same query suite.

3.4 The Supervisor and the Statistical Module

While closed LDBMSs permit only the detection of inconsistencies, inconsistencies in cooperative LDBMSs can either be prevented by the Grabber or be detected by the Tester. The switching between the two modules is performed by the *Supervisor*. We list the situations for which the Tester "T" is more appropriate than the Grabber "G" and vice versa.

	Low	*High*
Update/Query rate	T	G
Number of constraint violations	T	G
Number of irreparable violations	T	G
Space availability *vs* demand	T	G

The *Statistical Module* is responsible for gathering information on the number of updates, queries, reparable and irreparable violations per time unit, and estimate the statistical evolution of those parameters over time. This data is provided by the other modules of the Wrapper, as shown in Fig. 1; information on irreparable violations is provided by the FDB layer through the Supervisor. Furthermore, the Statistical Module must monitor the load of the Wrapper and the load of the LDBMS during the activity of the Grabber or the Tester, so that the actual impact of either module on the LDBMS load can be assessed properly.

The decision on the activation of one of the two modules should be based on the impact of the *combination* of the above parameters. Therefore, a cost metric should be developed for the Supervisor, which weights the impact of each module at the selected timepoints. This implies that the computation of the impact of activating each module is itself a costly operation. Hence, an important derivative of the work of the Statistical Module should be the identification of the timepoints, at which a switch between the two modules should be considered: those timepoints coincide with changes in the load of the LDBMS changes or in the above parameters. We intend to consider mechanisms for the recognition of patterns in the evolution of the system load or the parameters' values.

4 Compensation of Constraint Violations

Since the LDBMSs in FACE are autonomous, the compensation of locally performed (sub)transactions must be supported. Since global transactions in a federation must be decomposed into several local transactions executed by the participants, a nested transaction model is needed on the global level. We are going to build on an existing model, like DOM [BÖH+92], ConTracts [WR92] or multi-level-transactions [Wei91, WS92], and expand it by a *Compensator* module.

4.1 Goal of the Compensator

In a closed LDBMS , constraint violations cannot be prevented, but only detected by the Tester. Repairing the inconsistencies thus arising differs from classical transaction rollback, because the original values of objects modified by a local application are irreciprocally lost. The purpose of the Compensator is to propose a compensating strategy bringing the federation back to a consistent state in an effective way after a constraint violation is detected.

Example 6. In our example database, consider a violation of constraint *C-1* caused by deleting a `Personnel` object e corresponding to an e' in `DEPARTMENT_2.Member`. According to *C1-global-del-rule* in Example 1, a compensating strategy is the deletion of e'; this may cause a cascade of further updates/deletions. An alternative would be to rebuild e from information in e' and other objects related to it, setting unknown values equal to NULL, if allowed, and asking the user for a value otherwise. This alternative is feasible in a relational LDBMS but may be impossible in an OODBMS, if the OID e cannot be reproduced. □

In general, there are several alternatives of resolving an inconsistency. Some of them may be impossible due to lack of information, such as recovering a deleted object in an OODBMS, or may be prohibited by the LDBMS. Other alternatives may have a very high overhead in terms of performance or information loss, e.g. in the case of a cascade of deletions. Others may require a large amount of information from the user. Hence, the prerequisites and the cost of each compensating "strategy" must be modelled and computed. Using this knowledge, the Compensator must select among a set of alternative strategies. In particular, the Compensator must have: (i) a "strategy description language" modelling the information concerning the cost and prerequisite data for the strategy to be feasible; (ii) A cost metric expressing the cost of a strategy; (iii) a mechanism generating strategies; (iv) an algorithm selecting among the strategies.

The strategy selected by the Compensator must be executed as a global transaction. If this transaction fails, or if the Compensator cannot find a feasible strategy, the inconsistency cannot be repaired immediately. Then, we can establish a view which filters out the objects not satisfying the constraints. We consider a View Manager module to this purpose, as described in Section 5.

4.2 Strategy Description

A first approach to the strategy description language of FACE is the following graph-oriented model, initially conceived for simple, relational-like operations.

Definition 1. A *compensation action (C-Act)* is a tree with up to 5 levels. The root at level-0 is one of the $\{INSERT, UPDATE, DELETE\}$ commands. A node at level-1 is a LDBMS class (relation or other entity set). A node at level-2 is the object of this class, on which the action is applied. A node at level-3 is an attribute, and a node at level-4 is its value.

Definition 2. A level-5 node in a C-Act may have a constant of `NULL` value and is labelled with a *status* and an *origin*. The status can be specified or unspecified. The origin is db-supplied, user-supplied or system-supplied.

A specified value is either db-supplied, i.e. can be found in the database, or user-supplied, if the user must provide it. An unspecified value is a system-supplied one, such as an object identifier. The following table holds:

Origin	*Status*	
	specified	unspecified
db-supplied	c	0
user-supplied	–	**C**
system-supplied	∞	**c**

where c, **c**, **C** are cost values produced by a cost function. According to this table, the acquisition of a db-supplied unspecified value has no cost, since it can be set to `NULL`. A user-supplied specified value is meaningless, while a system-supplied value cannot be specified in advance.

Definition 3. A *compensation strategy (CS)* is a set of C-Act's connected with three types of edges: (i) A *tree edge* emanates from a node of a C-Act and points to a node at the next higher level of the same C-Act. (ii) A *cascading edge* emanates from a node at level-2 of a C-Act and points to a level-2 node at another C-Act. (iii) A *scheduling edge* emanates from the root-node of a C-Act and points to the root-node of another C-Act.

Definition 4. An *abstracted compensating action (AC-act)* consists of the level-0 and level-1 of a C-Act.
An *abstracted compensation strategy (ACS)* is a directed acyclic graph, whose nodes are AC-Act's connected with cascading and scheduling edges.

4.3 The Compensation Process

In the compensation process, we distinguish three phases, in which compensation strategies are built and compared.

Preprocessing phase. To construct an ACS, the compensator takes one of the alternatives specified in the violated constraint and builds an AC-Act for each action in it, connecting them with scheduling edges. If there are constraints defined over the classes specified in an AC-Act, their AC-Act's are formed and connected to the original one with cascading edges.

An ACS provides a means of comparing alternative strategies without fully evaluating them: a cost metric can be developed, based on the weighting of the AC-Act nodes and the edges among them. The cost of an AC-Act node can be computed using parameters like (i) the cost of accessing each LDBMS, to which a class belongs, (ii) the size of the class etc. An upper limit on the number of scheduling edges in the ACS or on the number of cascading edges from/to an AC-Act may be used to remove ACSs requiring massive updates.

This phase has the disadvantage that cascading edges are only presumed; the actual values may or may not cause them. Moreover, infeasible strategies cannot be recognized promptly. We observe though, that infeasible strategies are those involving INSERT-actions only, because they may require system-supplied specified values. Insertions are generally expensive, because they usually need user-supplied values. Hence, cascading insertions are undesirable. On the other hand, cascading deletions and updates may or may not occur for a specific object. We can therefore alleviate the above problems by assigning a relatively high cost to ACSs involving INSERT-actions, thus allowing other ACSs to be considered in the next phase, even if they contain potential cascading updates and deletions.

Expansion phase. By the end of the preprocessing phase, the remaining ACSs are expanded to CSs by fetching the objects affected by the constraint compensation. A C-Act with an incompatible Origin/Status setting for a level-5 node results in the elimination of the whole CS as infeasible. Heuristics based on the number of objects being accessed can be introduced to filter out expensive strategies.

The output of this phase is a set of feasible strategies. If the set is empty, the constraint violation cannot be compensated. The process is completed at this point, and the overhead of the next phase is avoided.

Comparison phase. In this phase, the CSs are compared in terms of execution cost. Parameters to be considered in the cost function are: (i) the cost of accessing the LDBMS, to which a class belongs, (ii) the total number of nodes in the C-Act's, (ii) the ratio between affected objects and total number of objects in a class, (iii) the cost assigned to each value according to its status and origin. An algorithm reorganizing strategies by shifting C-Act's connected with scheduling edges can be invoked to find the optimal schedule for a CS.

The output of this phase is the compensation strategy suggested to bring back the FDB into a consistent state.

An example of the compensation process for constraint *C1* in Example 1 is discussed in the Appendix.

5 The View Manager

The establishment of views over a federation is generally desirable to hide semantic heterogeneities. In FACE, we expand this concept to the masking of inconsistencies caused by constraint violations. In particular, when the Compensator fails to repair an inconsistency, FACE attempts to hide it from the global applications. A view is established which filters out the objects not satisfying the constraints. These objects are still visible to the local applications, which are thus not affected.

5.1 View description

The first issue concerning the establishment of a view is whether the objects filtered out should be specified extensionally or intensionally. An "intensional" description means that the view is specified through the rules expressing the constraints; objects violating the constraints are automatically filtered out. An "extensional" description means that the view contains all objects except those explicitly specified.

Example 7. Continuing Example 6, we assume that the constraint violation were irreparable, so that a dangling object e appears in the `DEPARTMENT_2.Member` class. Although this is an inconsistency for the federation, it might be acceptable for the DEPARTMENT_2 database, for instance if its local financing applications ignore all employees appearing in `DEPARTMENT_2.Member` but not in `DEPARTMENT_2.Personnel`.

An intensional description of a view filtering out this object might have the form:

```
SELECT * FROM DEPARTMENT_2.Member
WHERE key IN ( SELECT key FROM DEPARTMENT_1.Personnel
               UNION
               SELECT key FROM DEPARTMENT_2.Personnel
               UNION
               ...
               UNION
               SELECT key FROM DEPARTMENT_n.Personnel )
```

where the predicate in the WHERE-clause implies querying *all* `Personnel` databases for every object in `DEPARTMENT_2.Member`.

An extensional description would have the form:

```
SELECT * FROM DEPARTMENT_2.Member
WHERE key != e.key
```

□

As can be seen from the example, the intensional description guarantees that only objects satisfying the constraints are visible. It has the advantage of remaining unaffected by subsequent violations of the same constraint. Moreover, if at some later point, another transaction removes the inconsistency, the object becomes automatically visible again. The disadvantage of this approach is the execution overhead.

The extensional description is straightforward and has low execution overhead. However, this view must be expanded whenever a new violation of the constraint occurs. If the inconsistency is removed later on, this must be detected and the view modified accordingly.

The pros and cons of the two approaches should be weighted against the frequency of uncompensatable constraint violations and the possibility of later inconsistency removal. We intend to initially provide the intensional description, but we are going to explore the probability of using the extensional description, whenever the update overhead is low.

5.2 View updates

The second issue to be addressed by the View Manager is that of view updates. Whenever views are established over the federation, global update transactions should be applied on them. If the view is described intensionally, the view update should be identical to an update satisfying the constraint(s) in the view's description. However, it should be guaranteed that no objects *not* satisfying the constraints are inadvertently updated. Methodologies for the view update problem are applicable here.

If the view is expressed extensionally, all objects specified in its description should be excluded from the update. If the update concerns objects relevant to those not filtered out by the view, objects relevant to those filtered out should similarly be excluded. Thus, the original update may need to be expanded by join or anti-join operations.

6 Related Work

A current proposal for global constraint checking assumes the presence of a constraint manager within each participant [GW96]. Unfortunately, such a built-in constraint manager is usually not available in existing databases. Another approach assumes the LDBMSs to be active databases [BKK96].

In [CT97, KLB96, VPR95], this requirement is relaxed: the federation members are conventional DBMSs monitored by an active federation manager, the modules of which catch signals related to updating transactions. This methodology is close to the approach underlying the Grabber module of FACE. However, these approaches are limited to cooperative LDBMSs and assume that *all* applications permit eavesdropping. FACE is appropriate for closed LDBMSs as well, since we focus on detecting and compensating constraint violations.

The Wrapper module of FACE is conceptually close to the active mediator of [KLB96] and the intermediate layer components used in other federation managers over multidatabases [CT97, FHLM96, KLB96, PGU96, VPR95]. They all stem from the typical approach of introducing a intermediate layer, which realizes a mechanism that is absent in the underlying system. However, FACE classifies the LDBMSs encapsulated by the Wrappers. Our classification of cooperative and closed and active, quasi-active and passive LDBMSs forms the basis of a methodology to manage constraints in each category. This classification and the compensation of constraint violations that cannot be enforced is, to the best of our knowledge, unique in this context.

7 Conclusions

In this paper we have presented FACE, a flexible approach to constraint enforcement in federated database systems. The flexibility is due to the fact that we allow to choose among and to combine different mechanisms for dealing with global constraint violations, even in passive closed LDBMSs.

FACE uses active rules for detecting violations and reacting to them. By active rules we can specify different kinds of actions like aborting the (local) transaction which issued the violating operation or carrying out certain repair actions for achieving a consistent state. A main prerequisite for this is that we are able to delay the commitment of local transactions, as our Grabber does. If this is not allowed, FACE provides a Tester that detects violations and a Compensator that recovers the federation data in a consistent state. If compensation can be performed using different strategies, statistics and information on the LDBMS properties are used to choose the most efficient one.

We are going to realize FACE by implementing a prototype system. Our first goal is the prototypic evaluation and comparison of the Grabber and the Tester and the establishment of a model for compensating strategies. Our current graph-oriented strategy description language must be extended to cover operations in an OODBMS. Furthermore, we focus on the optimization of the Tester's query suites and the automatic generation of rules for constraint enforcement and compensation of violations.

References

[BGS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 1(2):181–240, 1992.

[BKK96] G. von Bültzingsloewen, A. Koschel, and R. Kramer. Active Information Delivery in a CORBA-based Distributed Information System. In K. Aberer and A. Helal, editors, *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*. IEEE Computer Society Press, 1996.

[BÖH+92] A. Buchmann, M.T. Özsu, M. Hornick, D. Georgakopoulous, and F.A. Manola. A Transaction Model for Active Distributed Object Systems. In Elmagarmid [Elm92], pages 123–158.

[CT97] S. Conrad and C. Türker. Data Consistency in Enterprises by Integrating Preexisting ¿ Information System. In *Proc. Wirtschaftsinformatik'97*, 1997. *to appear (in german).*

[DKS92] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. Query optimization in heterogeneous DBMS. In *Int. Conf. on Very Large Databases*, pages 277–291, Vancouver, Canada, 1992.

[Elm92] A.K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[FHLM96] G. Flach, A. Heuer, U. Langer, and H. Meyer. Transparent Queries in Federated Database Systems. In S. Conrad, M. Höding, S. Janssen, and G. Saake, editors, *Workshop "Föderierte Datenbanken": Kurzfassungen der Beiträge*, number ITI-96-01 in Institutsbericht, pages 45–49. Universität Magdeburg, 1996. *(in german).*

[GST96] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Obect-Oriented Federated Database System. In T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, editors, *Proc. of the 22nd Int. Conf. on Very Large Data Bases (VLDB'96)*, pages 378–389. Morgan Kaufmann Publishers, 1996.

[GW96] P. Grefen and J. Widom. Protocols for Integrity Constraint Checking in Federated Databases. In K. Aberer and A. Helal, editors, *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, pages 38–47. IFCIS, The Intn'l Foundation on Cooperative Information Systems, IEEE Computer Society Press, 1996.

[KLB96] Thomas Kudrass, Andreas Loew, and Alejandro P. Buchmann. Active object-relational mediators. In *CoopIS'96*, pages 228–239, 1996.

[MT93] D. Mukhopadhyay and G. Thomas. Practical Approaches to Maintaining Referential Integrity in Multidatabase Systems. In H.-J. Schek, A. P. Sheth, and B. D. Czejdo, editors, *RIDE-IMS'93 Research Issues in Data Engineering: Interoperability in Multidatabase Systems (Wien, 1993)*, pages 42–49. IEEE Computer Society Press, 1993.

[PGU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In S.Y.W Su, editor, *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*, pages 132–141. IEEE Computer Society Press, 1996.

[Spi96] Myra Spiliopoulou. A calibration mechanism identifying the optimization technique of a multidatabase participant. In *PDCS'96*, Dijon, France, Sept. 1996.

[SW91] H.-J. Schek and G. Weikum. Erweiterbarkeit, Kooperation, Föderation von Datenbanksystemen. In H.-J. Appelrath, editor, *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'91)*, Informatik-Fachbericht 27, pages 38–71. Springer-Verlag, 1991.

[VPR95] Kanonkluk Vanapipat, Niki Pissinou, and Vijay Raghavan. A dynamic framework to actively support interoperability in multidatabase systems. In *RIDE-DOM'95*, pages 148–153, 1995.

[Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transactions Management. *ACM Transactions on Database Systems*, 10(1):132–180, 1991.

[WR92] H. Wächter and A. Reuter. The ConTract Model. In Elmagarmid [Elm92], pages 219–263.
[WS92] G. Weikum and H.-J. Schek. Concepts and Applications of Multi-level Transactions and Open Nested Transactions. In Elmagarmid [Elm92], pages 515–553.
[ZL94a] Qiang Zhu and Per-Åke Larson. Establishing a fuzzy cost model for query optimization in a multidatabase system. In *27th Hawaii Int. Conf. on System Sciences*, Maui, Hawaii, 1994.
[ZL94b] Qiang Zhu and Per-Åke Larson. A query sampling method for estimating local cost parameters in a multidatabase system. In *10th Int. Conf. on Data Engineering*, pages 144–153, Houston, Texas, 1994.

A Appendix

Example 8. We consider the violation of constraint *C1* caused by the deletion of an employee e being member of a project. Before studying alternative compensation strategies, we must consider an example schema of the classes involved. We assume that the underlying LDBMSs are relational; we will return to this assumption by the end of the example.

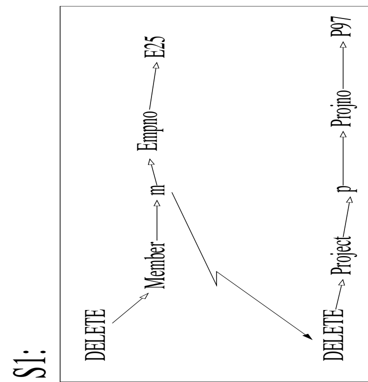
Let employee e belong to department D_1. Let `DEPARTMENT_1.Personnel`(Empno, Name, Forename, Salary, Address, Manager) describe an employee in that department. The primary key is Empno and all properties but the Address must be assigned a non-NULL value. Employee e corresponds to the tuple
$e = (E25, Smith, Gwendolyn, 12000, NULL, E31)$.

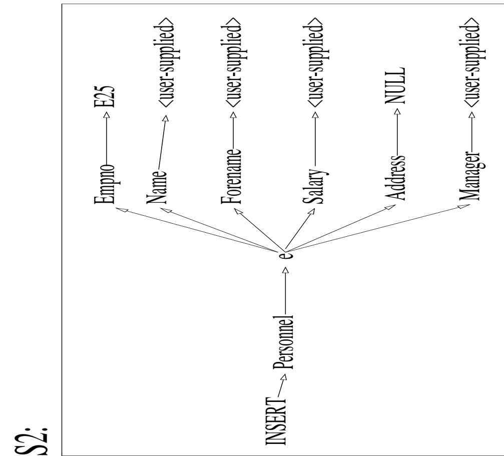
In the research department D_2, let `DEPARTMENT_2.Project`(Projno, Title, NumMembers, Manager, StartDate, EndDate) describe the projects, having Projno as its primary key. Let `DEPARTMENT_2.Member`(Empno, Projno, TaskDescription, StartDate, EndDate) describe the project members, having Empno as the primary key. An employee participates in only one project, while a project has one or more members.

A constraint *C2* disallows the existence of projects without members:
define rule *C2-global-del-rule*
on $DEPARTMENT_2.delete_member(e)$
if $\exists DEPARTMENT_2.Project(p)$ *and*
$DEPARTMENT_2.Project.Projno = e.Projno$ *and*
$DEPARTMENT_2.Project.NumMembers = 1$
do $DEPARTMENT_2.delete_project(p)$

Now, we assume that Gwendolyn Smith is member of a project with Projno $P97$. The tuple in `DEPARTMENT_2.Member` is
$m = (E25, P97, Gardening, 1-JAN-95, 31-DEC-98)$, while the tuple in `DEPARTMENT_2.Project` is
$p = (P97, The_ideal_garden, 1, 1-SEP-93, 31-AUG-99)$.

We consider two compensation strategies depicted in Fig. 8. Strategy $S1$, compensates the constraint violation by deleting tuple m, whereby tuple p must also be deleted. Strategy $S2$ attempts to re-insert e in `DEPARTMENT_1.Personnel`.





In the preprocessing phase, only the ACSs of the two strategies are considered. Then, it is not certain that $S1$ will cause a cascading update, because the actual values of the attributes in the objects m, p are not known before fetching them from the database. It is certain, though, that $S2$ has a high cost, because it involves an INSERT-action. The strategy to be preferred depends on the actual cost parameters.

This article was processed using the LaTeX macro package with LLNCS style