Consistency Constraints in Database Middleware

Andreas Geppert 1
Department of Information Technology
University of Zurich
Winterthurerstr. 190
CH-8057 Zurich
geppert@ifi.unizh.ch

Edward L. Wimmers
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, California 95120-6099
edwimmers@yahoo.com

Abstract. Consistency constraints in database systems ensure that the data stored in databases is correct with respect to the modeled real world. In heterogeneous systems managed by database middleware, it is often not possible to meet this ideal or it can be met only under very strong assumptions about involved data sources. Constraint violations are often caused by external updates, so that the middleware cannot prevent inconsistencies from being introduced. Many data sources lack support for constraint-related tasks and thus are not able to contribute to constraint management. Moreover, data sources might wish to maintain autonomy, so that the database middleware does not have the authority to enforce constraints. We propose a novel approach to constraint management in this type of environment which provides applications with a high degree of flexibility regarding handling of inconsistencies. The spectrum of reactions ranges from correcting inconsistencies to annotating data with information about constraint violations. Applications can decide to use inconsistent data but are made aware of violations. Ultimately, the range of strategies for checking constraints and handling inconsistencies matches a spectrum of data sources with varying support for constraint-related tasks. Constraint checking and reacting to inconsistencies are therefore possible even if some of the data sources are not able or willing to participate in these tasks.

Keywords: consistency constraints, database middleware

1 Introduction

Consistency constraints in traditional relational database systems ensure that data is accurate with respect to rules found in the real world. The ACID property [16] guarantees that inconsistent data is not permanently stored in a database. Applications and users are thus shielded from using inconsistent data.

Database middleware [19] allows access to data from multiple data sources and lets these data be combined into an integrated view, often representing information needed for complex tasks. Such systems are needed in domains such as design, medicine, etc. Database middleware systems

1. Work done while with IBM Almaden Research Center
have in contrast to federated database systems [10, 23] to deal with a large diversity of data sources, which typically range from powerful database systems to simple and specialized file-based data systems.

Since data sources are built (often by different organizations) prior to integration through the middleware, many constraints found in the real world can be violated by the integrated data. Enforcing a rigorous notion of consistency is generally impossible due to lacking support and/or autonomy of data sources. Some data sources, such as those on the Web, do not even support updates of their data, so that correcting inconsistencies is not an option. Furthermore, some applications are not willing to trade completeness of information for consistency. Information which is inconsistent in one respect can still be useful in others. Applications thus should be able to use this data, but should also have the opportunity to become aware of constraint violations.

In this paper, we propose an approach to consistency constraints in database middleware which allows constraint management to be tailored to the needs of applications as well as to the capabilities of the data sources involved. In particular, a set of strategies defines how and when to check constraints. Some of these strategies can exploit data source functionality such as update notification, while others do not assume any kind of data source support. Another set of strategies is proposed for reactions to constraint violations; the adequate choice depends on the constraint itself, application requirements, and again the data source capabilities. The spectrum of strategies ranges from user-defined corrective (“repair”) actions to annotating query results with information about constraint violations.

The novelty of the approach described here thus is twofold. First, due to its high degree of flexibility, a broad spectrum of application needs with respect to consistency constraints can be met. Second, it is able to exploit, yet does not rely on, a diversity of data source capabilities and thus can handle “weak” data sources which do not support constraint management.

The remainder of this paper is organized as follows. Next, we discuss requirements of constraint management in database middleware in more detail. Section 3 introduces our approach to consistency constraints and the strategies it comprises, and section 4 discusses implementation issues and early experiences. Section 5 discusses related work, and section 6 concludes the paper.

2 Database Middleware and Consistency Constraints

In this section, we first discuss the notion of consistency constraints in traditional database systems and then contrast it with requirements in database middleware.

2.1 Consistency Constraints in Database Systems

A database consistency constraint defines a restriction on the data that can be legitimately stored in the database. A constraint is usually specified as a formula over the extent (i.e., relation or collec-
tion) for which the constraint is defined. If this formula evaluates to true, then the constraint is said to hold, and if it evaluates to false the constraint is said to be violated. The default reaction to constraint violations is to abort the transaction that caused the violation; some approaches allow more sophisticated reactions (e.g., cascading deletes in case of referential integrity violations).

According to the ACID-principle [16], all constraints defined in a database must hold at the end of a transaction. If some constraint is violated by a transaction, then the transaction is not allowed to commit. Constraints prevent inconsistent data from being inserted into a database, and also preclude updating data in an inconsistent way. Thus, constraints reflect a boolean, “black and white” view of data—only “good” data is allowed to persist in the database. The ACID-properties guarantee that consistent database states are transformed into (not necessarily different) consistent states. An empty database is considered to be consistent, thus defining constraints over an initial, empty database ensures that the database will always be consistent with respect to the set of defined constraints. In case a constraint is defined over an already populated database, then inconsistencies can be remedied in several ways, such as moving inconsistent data items to special places or deleting them. As a consequence, current database systems guarantee that the committed database state is always consistent with respect to the set of defined constraints.

The property that (committed) database states are always consistent implies that exceptions to the rule are not possible. If a constraint is defined for a collection, it must hold for all the elements of this collection. If in exceptional cases data items might be allowed to violate the constraint, then the constraint cannot be formulated. As an example, take the famous constraint that employees should not earn more than their manager. A meaningful exception would be an ingenious engineer being highly valuable for her company and being paid an extraordinarily high salary. This salary could not be recorded in the database in the presence of the aforementioned constraint. Thus, either data needs to be faked or the constraint could not be defined. In the latter case, the knowledge that the constraint exists (in the real world) is lost, and it could not be enforced for those employees for which it holds in the modeled part of the real world.

More flexible notions of consistency constraints (which, e.g., would allow applications to use inconsistent data in certain cases), have not been considered in traditional database systems because they provide limited additional benefits for typical users. However, a more flexible notion of consistency is of paramount importance in many database middleware scenarios for reasons to be discussed next.

### 2.2 Requirements of Consistency Constraints in Database Middleware

Many middleware scenarios require a more flexible approach to consistency constraints than the one supported in traditional database systems (and also than those proposed for federated database systems [4, 5, 9, 11, 12, 14, 18]).

From an application perspective, the purpose of the middleware is to leverage data from multiple sources, creating valuable information (which is not available from one of the sources alone).
As we argue below, it is in many applications no longer possible to make a clear cut distinction between good (consistent) and bad (inconsistent) data—even data that is inconsistent in some respect might still be valuable in others. From an administrative perspective, inconsistencies (such as conflicting values or missing data) might and usually do result from integration of multiple sources. Thus, the prerequisite that the initial database state is consistent is no longer valid. Finally, from a technical perspective, the means to check and enforce consistency constraints are usually restricted due to autonomy of data sources. In consequence, the middleware DBA might not have the authority to prevent the storage of inconsistent data in the data sources.

Many of the (database middleware) applications we consider need middleware to integrate (raw) data into information useful for supporting complex tasks. In these scenarios, it is often not possible to foresee under which conditions data is inconsistent and useless, and when it might still be usable and important even though it is inconsistent. Thus, for these applications, there is no longer a clear separation between consistent and inconsistent data, but a spectrum from fully consistent to fully inconsistent data, with shades of grey in between the two extremes. Applications should be allowed to use data violating some constraint, but, in order to avoid incorrect usage of the data, should also be provided with information about which constraints are violated.

As an example, consider an address database used for opinion polls and telemarketing. Assume one constraint requires that the mailing address of each person is valid, i.e., the street name/city/state/zipcode combination appears in a directory of zipcodes. Under the conventional approach to constraint management, an object with an invalid mailing address would not be tolerated in the database. However, as long as the phone number is correct, the data could still be used by some applications (e.g., for telemarketing), while for others the data should not be used (e.g., mailings).

After a (global) database has been created by integrating multiple data sources, inconsistencies are often encountered, such as conflicts between data items from different sources. For instance, two objects from two different sources representing the same real-world object could have two different values for an attribute. These two objects would therefore violate the constraint that different representations of the same entity have uniquely defined values for their common attributes. A straightforward solution would be to define views over data sources which exclude inconsistent data. Inconsistent data would then not be visible to applications; but applications would also have only a restricted and incomplete view of the entire database. Hence, middleware administrators need adequate means to address existing inconsistencies and to balance consistency and completeness requirements.

Finally, the standard solution to check and react to inconsistencies (preventing inconsistent updates or re-installing consistent values) is not always viable in database middleware either. Constraint violations can be caused by external updates of which the middleware cannot become aware. Furthermore, data sources are usually autonomous, and therefore the middleware has no privileges to update data sources. Even worse, some types of data sources (e.g., file-based dedicated sources), do not support a write-interface, and therefore the middleware could not even act as a normal client and execute appropriate repair operations.
The big open question thus is how to address consistency constraints in database middleware. The middleware should handle constraint violations in a way that is adequate for each constraint in question and the data source it affects, and that also fits the needs of applications. To that end, the constraint designer (the middleware database administrator) needs a spectrum of constraint management functions:

1. If a remedy for violations of a constraint is known, and repairs are possible, then repair inconsistent objects.
2. If constraint violations cannot be meaningfully used in any application, users and applications should never be able to access inconsistent data.
3. If partially inconsistent objects might still represent valuable information for some applications, they should be accessible for applications. Users should however be warned that the data they are using contains inconsistencies.

3 The Garlic Approach to Constraint Management in Middleware

In this section we introduce a novel approach to constraint management in database middleware. The broad spectrum of requirements and data source capabilities is covered by a collection of strategies for checking constraints and handling inconsistencies. These strategies have been proposed and implemented for the Garlic database middleware system [2], which is a typical representative of database middleware systems.

We next introduce Garlic and then describe a set of dimensions which span a design space of constraint management strategies. We then briefly list the set of proposed strategies and describe the various strategies in more detail.

3.1 The Garlic Database Middleware System

Garlic’s [2] (see Fig. 1) data model is based closely on the Object Data Management Group (ODMG) standard [3]. Schemas are defined in the Garlic Data Language (GDL) and are maintained in the global catalog. A schema is a collection of interfaces, which define the structure and behavior of types. An interface can have an extent which is the set of all existing instances of that interface. Garlic also has views, which have a derived extent. Objects can be accessed at the middleware level via a C++ programming interface or Garlic’s query language, which extends SQL with path expressions, nested collections, and method calls.

In order to abstract from and homogenize the varying capabilities of data sources, wrappers [24] are provided as intermediaries between data sources and the middleware proper. Each wrapper presents a uniform interface to Garlic and hides the specific characteristics of its data source’s data model, query facilities, and programming interface. Wrappers are well-suited to translate requests (in Garlic’s language and format) to the language and format their data sources understand.
In particular, the interplay of Garlic’s query processor and the wrappers is designed in such a way that the various level of query facilities (ranging from full-blown SQL to simple scans) of data sources can be exploited.

### 3.2 Dimensions of Constraint Management Strategies

The approach we propose here is organized along three dimensions:

- capabilities of the data sources involved,
- knowledge about the constraint,
- needs of applications using the data restricted by the constraint.

Data source capabilities determine the possibilities for checking constraints and reacting to inconsistencies. With regard to checking, the spectrum is defined by the following capabilities:

1. the data source is able to notify the middleware about updates,
2. the data source can maintain an update log and allows the middleware to query this log.
3. none of the above—the data source cannot communicate update information.

As an example for the first case, consider a relational database supporting triggers, where update information can be sent to the middleware from within action parts of a trigger. A typical example for the second case is a relational database system supporting triggers, which however does not allow to send events to the outside world. An example for the latter case are file-based image and document management systems.

The important capability of a data source with respect to reactions is whether it allows the middleware to update its data (i.e., whether it supports a write-interface). Alternatively, the middleware cannot update data sources: in this case it is irrelevant whether this is due to the absence of an API or because of missing privileges (data source autonomy).
Apart from the constraint proper (i.e., the condition that should hold), important information comprises how to repair violations of this constraint. If this knowledge is available, then middleware can remedy inconsistencies by performing certain updates on the inconsistent data provided that updates are possible (see above). For other constraints, no such repair function is known and the middleware cannot be told how to correct inconsistencies.

Application requirements are important in two respects. The first criteria is whether all applications using a collection of data want to see only the consistent data in this collection, or whether some of them could also use the inconsistent elements. The second aspect is whether constraint violations can be tolerated for some time period.

3.3 Constraint Management Strategies in Database Middleware

Due to the variations in the dimensions described above, a single one-size-fits-all solution does not exist. We therefore propose two groups of strategies; adequate strategies can be chosen on a per-constraint basis. Strategies in the first group determine when and how to check constraints, and those in the second group determine appropriate reactions to constraint violations.

With respect to constraint checking, the strategies are:

C-1: **notification-based** checking: a data source informs the middleware about updates to its local data; the middleware then checks constraints which the updated data might violate.

C-2: **constraint monitoring**: the middleware checks constraints in regular intervals by issuing queries over the data sources involved in the constraint.

C-3: **update polling**: the middleware retrieves information about updates from update logs in data sources. For each update record, constraints are then checked.

C-4: **just-in-time checking**: the middleware checks constraints at the latest possible point in time, as data is requested by queries.

As for reactions, there are also multiple strategies proposed:

R-1: **constraint repair**: the violation is remedied by executing further updates.

R-2: **alerting**: the (middleware) DBA or some responsible user is informed about violations.

R-3: **marking**: inconsistent elements of query results are annotated with information about constraint violations.

R-4: **nulling out**: inconsistent data in query results is replaced by NULL values.

R-5: **virtual repair**: inconsistent data for which a corrective action is known but cannot be performed on the data source is executed on the fly (i.e., users “see” the correct value, which is however not permanently written into the data source).

These options allow the selection of an appropriate strategy for each constraint, depending on the capabilities the data sources involved exhibit and on the requirements posed by the applications.
3.4 Constraint-Checking Strategies

Constraint-checking strategies are distinguished based on support of data sources involved for some form of update detection.

**Update Detection based on Notification.** Notification means the ability of a data source to inform the middleware layer about updates. It can be performed using triggers as offered by relational DBMS [7], using a built-in notification mechanism as supported in some DBMS (e.g., [21]) or using event services offered, e.g., by middleware platforms such as CORBA [22]. A notification includes the identification of the updated collection and object (i.e., key attribute values in the case of a relational data source). This information is used by the constraint manager to precisely determine which constraints to check, and for which objects to check them. In case notifications occur on a coarser level (e.g., only identifying the collection whose elements have been updated), the constraint manager still would be able to identify the constraints to be checked, but might check constraints unnecessarily for objects that have not been updated. Note that this strategy is only meaningful for constraints which have a repair as reaction or which raise an alarm.

The notification strategy guarantees that constraints are checked shortly after a potentially damaging update has occurred. Because the event needs to be sent over the wire, there is a small time window between the update and the check during which inconsistencies might be visible in query results even for those constraints that define a repair function.

**Update Detection based on Monitoring.** In case notification is not supported by data sources, monitoring has been advocated in several approaches. Monitoring means that a special application checks constraints at pre-defined intervals. This strategy comes in two flavors: constraint monitoring checks constraints (for the entire extensions over which the constraint is defined). Update polling accesses an update log in the data sources to retrieve information about updates performed since the last iteration. Update polling thus requires that the data source is capable of maintaining an update log. This requirement is met, e.g., by database systems which support triggers but do not allow events to be sent to the outside world. Both monitoring and polling give the same guarantee: they ensure that inconsistencies are detected at the latest after \( n \) time units, where \( n \) is the size of the interval. Inconsistencies introduced since the most recent check are visible in query results. Both are therefore practical only when the tolerance for inconsistencies is high, because a small checking interval will lead to performance degradations. Constraint monitoring is practical only for small data sources, because it always evaluates constraints for entire collections. Thus, each constraint checking iteration implies high extra system load for large collections.

This strategy is also only meaningful for constraints whose reaction is a repair or raises an alarm.

**Just-in-time Constraint Checking.** The option remaining for those cases where notification is not supported and monitoring/polling is not feasible, is to check constraints just in time, i.e., be-
fore or within queries. In other words, constraints are checked at the latest possible point in time and “on the fly” whenever queries against constrained data are submitted. Whenever a collection is queried, the constraint manager ensures that queries return only consistent results, or that inconsistent results are at least marked as such. This strategy guarantees that inconsistent results are never visible in query results (if a repair is known), or that users are made aware of the inconsistencies. Just-in-time checking therefore provides the maximal guarantee among all the strategies.

3.5 Strategies for Handling Inconsistencies

There are several strategies proposed for reactions to constraint violations.

**Repairs.** Repairs are corrective actions, which are defined by the constraint definer in the middleware DML or API. A repair consists of one or more update requests to the inconsistent data. Repairs require three prerequisites to be met: first, a meaningful repair must be known. In many cases, a corrective action is not known (e.g., in the case of uniqueness constraints, it may not be possible to decide which of the equal elements to delete). Second, the data source must provide a write-interface to the middleware, which might not be the case for simple data sources. Finally, the middleware must have the authority to update data in the data source. If one of these prerequisites is not met, another strategy must be chosen.

**Virtual Repair.** This strategy replaces inconsistent values by correct ones and therefore requires knowledge about how to determine correct values. The repair is only performed on elements of query results, not on data in data sources. Therefore, this strategy does not require write access to data sources.

**Nulling out Inconsistent Data.** This is a special case of a virtual repair operation, which replaces inconsistent objects or attributes in query results by NULL values. Data is modified only in the result of queries, but not in data sources. Thus, this option neither requires write access to data sources nor relies on knowledge about the correct values (of inconsistent objects/attributes).

**Marking Inconsistent Objects.** In this approach, information about consistency violations is attached to query results. Each element of a query result records whether it violates one or more constraints, and if yes, which ones.

Another, less informative, option is simply not to return results that do not fulfill the constraint. Note that this option could also be expressed by the user using a view mechanism: we therefore do not consider this option further.

**Alerting.** The final strategy is to alert some responsible user, such as the data source administrator or the middleware administrator, whenever an inconsistent object has been detected. This solution however only enables that inconsistencies are eventually taken care of, but it does not prevent users/queries from seeing inconsistent states in the meantime.
Table 1 summarizes the various strategies and shows for which kinds of data source capabilities they are applicable. We distinguish between consistency maintained in the database and (in)consistency as visible to users. This distinction is necessary, because the users’ view of inconsistent data can be different from the actual values of these data. “Strict” consistency means that constraints are enforced (either in the database or the users’ view), and “eventually strict” means that inconsistencies can be visible until they can be repaired.

<table>
<thead>
<tr>
<th>Data Source Capabilities</th>
<th>Approach</th>
<th>Level of Consistency (DB)</th>
<th>Level of Consistency (user visible)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• notification</td>
<td>asynchronous checking and repair</td>
<td>eventually strict(^a)</td>
<td>undefined(^a)</td>
</tr>
<tr>
<td>• write interface</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• notification</td>
<td>alerting</td>
<td>none</td>
<td>inconsistencies signalled</td>
</tr>
<tr>
<td>• no write interface</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• no notification</td>
<td>polling /monitoring and repair</td>
<td>eventually strict</td>
<td>undefined(^a)</td>
</tr>
<tr>
<td>• write interface</td>
<td>just-in-time checking with repair</td>
<td>eventually strict</td>
<td>strict</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• no notification</td>
<td>just-in-time checking and marking</td>
<td>none</td>
<td>inconsistent data identified</td>
</tr>
<tr>
<td>• no write interface</td>
<td>just-in-time checking and nulling-out</td>
<td></td>
<td>inconsistent data not visible</td>
</tr>
<tr>
<td></td>
<td>just-in-time checking and virtual repair</td>
<td></td>
<td>strict</td>
</tr>
</tbody>
</table>

Table 1: Summary of Constraint Management Strategies

\(^a\) the time window between (constraint violating) updates and repairs. Inconsistencies are visible during this time.

### 3.6 Examples

We use a simple telemarketing example to illustrate sample constraints. This example combines data about persons, phone numbers, and mailing addresses from different data sources. Correct addresses and phone numbers are important to actually reach the intended persons.

```plaintext
interface person {
```
string Name;
ref<addr> Address;
ref<phone> Phone;
}

interface addr {
  int StreetNumber;
  string Street;
  string City;
  string State;
  int ZipCode;
}

// The class zipListing is maintained by an external data source
// such as the post office or a mailing service.
interface zipListing {
  string Street;
  string City;
  string State;
  int ZipCode;
}

interface phone {
  int AreaCode;
  int phone;
}

If one desires to check that the address is valid (i.e., that there really is such an address within
the given zip code), it can be expressed as follows:

customization CorrectAddress on addr_ext// addr_ext is the extent of addr
  exists realZip in zipListing_ext
  this.Street = realZip.Street and
  this.City = realZip.City and
  this.State = realZip.State and
reaction ...

The constraint management subsystem determines which checking strategies are possible and
chooses the one that repairs the constraint most expeditiously. If notification is not possible and a
tolerance interval exists (say, inconsistencies are tolerable for at most 24 hours), then the update
polling or constraint monitoring strategy might be chosen. Otherwise, the just-in-time strategy will
be selected for checking constraints (note that some reaction-strategies would also require just-in-
time checking). Below, we illustrate possible scenarios for reacting to violations.
There are several possible reactions the owner of the data could have:

1. There is a repair operation. This requires a procedure that modifies addresses such as
   
   ```
   void repairAddr(addr addrToBeFixed);
   ```
   
   which would be a function that examines zipListing and tries to find the nearest possible address in that zip code. Alternatively, it could try to find a zipListing that matches the given address. Policy decisions of this sort would be left up to the writer of repairAddr.

   In the constraint definition, this repair is expressed as:
   
   ```
   reaction repairAddr(this);
   ```
   
   This repairs each addr-object that violates the CorrectAddress constraint after the violation has been detected.

2. There is an alert operation:
   
   ```
   reaction alert DBA;
   ```
   
   which sends email to the DBA with information about the object that violates the constraint.

3. Marking is chosen:
   
   ```
   reaction mark;
   ```
   
   In this case the user of the data can request a mark as part of the query:
   
   ```
   select p.Street, mark(p.oid,CorrectAddress) from persons p
   ```

4. NULLing out is chosen. In this case, a NULL value is returned when the constraint is violated.
   
   ```
   reaction NULLout this;
   ```
   
   Alternatively, only certain fields can be NULLed out:
   
   ```
   reaction NULLout this.ZipCode
   ```
   
   which would return a NULL int value when the ZipCode field is accessed.

5. Virtual Repair should be done. In this case, there needs to be one repair for each attribute:
   
   ```
   reaction replace ZipCode by this.ZipRepair()
   ```
   
   assuming that a method ZipRepair has been defined as part of the object:
   
   ```
   int ZipRepair();
   ```

Many other domains also need support for constraint management. For example, imagine a real estate office. There could be constraints that clients must have a credit report where the credit report refers to information obtained from a credit bureau. In drug development, there could be constraints that new drugs have passed a certain series of tests. An employer might have the constraint that the human resources department and the employee’s own department agree who the employee’s manager is. A company might wish to keep its sales information consistent with inventory reports maintained by the accounting department.

### 4 Implementation and Early Experiences

In this section, we describe the implementation of constraint management in the Garlic database middleware system [2]. Fig. 2 gives an overview of the various components and their interaction.
4.1 Implementation of Constraint Management in Garlic

Information about constraints is maintained by the middleware schema. This information comprises the constraint condition, the chosen strategy for checking and reactions, addresses for alerting and the name of a repair function (if applicable). Constraint definitions are also linked to the database root for which they are defined.

The constraint management subsystem is responsible for maintaining the constraint-related part of the schema and for common tasks such as generating and issuing constraint-checking queries.
and reacting to inconsistencies. All the strategies use the functions of the constraint manager, which made it possible that a broad variety of strategies could be effectively implemented.

The notification strategy is implemented by a new middleware component called event handler. The event handler waits for new events to be put on the event bus, and as soon as a new event can be read, processes the event. The constraint subsystem then first determines the constraints that need to be checked, issues constraint checking queries and processes the result (which represents inconsistent objects).

Events are put on the event bus by data sources capable of notification. An example for such a data source is DB2 Universal Database. Triggers (which are created upon constraint definition) fire after potentially damaging updates have been performed. Each such trigger then puts an event representing the update information on the event bus using a user-defined function.

Update polling and constraint monitoring are implemented using two new Garlic applications. The constraint monitor is started by the system clock whenever the interval defined for the constraint in question has passed. It then invokes the constraint manager, which in turn issues the constraint checking query and handles inconsistent objects returned as a result of this query. The update poller is also started by the system clock. It invokes the constraint manager, which reads information about new updates stored in the data sources’ update logs and checks all the constraints defined for the modified collections. Each update log is itself represented as a collection in Garlic; thereby it is possible to abstract from formats specific to data sources.

In some sample Garlic-applications, update polling has been done for Oracle (the used Oracle-version did not allow to externalize events in triggers). Per Oracle-based data source, a new table is introduced as update log. Oracle-triggers are defined upon constraint definition, each of them fires after updates that may cause constraint violations. In the action part of each of these triggers, information about the update is recorded in the update log.

Just-in-time checking has been implemented as an extension to the query processor. After query parsing, the constraint manager analyses the internal query representation whether it refers to collections to which just-in-time constraints are associated. If so, the constraint manager issues constraint checking queries and handles the (inconsistent) results. The query processor then resumes query execution. Just-in-time constraints are always checked for the entire collection (for which the constraint is defined), and they are always checked before the user query is processed. This is necessary because eventual repair operations on inconsistent objects can affect the query result.

Finally, marking is currently being implemented through a built-in function. Annotations about constraint violations thus are part of the query result, just like a normal select-expression. Marking only occurs if requested in queries. This solution was chosen because marking without explicit user request would have implied drastic changes in the query processor. Representing markings as part of the query result does not require the introduction of new flags and therefore does not affect the way applications process query results.
4.2 Discussion

Early experiences using the current implementation shows that the devised strategies are feasible and adequate. In particular, it is possible to choose an appropriate checking/reaction-strategy depending on each constraint in question, application requirements, and data source capabilities.

Although systematic performance measurements still need to be done, it is already apparent that the just-in-time checking with repair functions as reactions needs to be improved with respect to performance. Two optimizations are possible: by a tighter integration of constraint checking and (user) query processing, and by reducing the set of objects to check. A tighter integration means that constraint checks/repairs and the user query are not executed serially, but that constraints are checked within scans of input collections. Second, in many cases a more thorough analysis of constraint conditions and repair functions might reveal that the repair function of a constraint does not affect the query result. If a query does not refer to attributes in its select- or where-clause that are potentially updated by repair functions, than the constraint check can be skipped entirely. If the query however selects some of those attributes, but does not refer to them in the where-clause, then the constraint check and the repair can be performed on the result instead of all objects input to the query.

The implementation of other strategies is less critical to be optimized, for two reasons: some of them already allow to narrow down the set of objects for which constraints need to be checked (in the notification and update polling strategy). Second, notification, polling and monitoring all operate asynchronously. Although they affect overall system load, they do not directly increase query response time to the same degree as just-in-time with repairs does.

5 Related Work

Approaches to constraint management in heterogeneous and distributed environments proposed so far [4, 5, 9, 11, 12, 14, 17, 18] assume database functionality such as update detection (mostly using triggers) and transaction management. These assumptions are not met by many of the data sources encountered by middleware. We next discuss these approaches in more detail.

In [12, 13], Grefen and Widom propose a family of protocols for checking consistency constraints in federated database systems. Two important properties of these protocols are safety (each potential constraint violation is signalled), and accuracy (whenever an alarm occurs, there is indeed a constraint violation).

Ceri and Widom [4] address constraint management for loosely-coupled heterogeneous database systems. Supported types of constraints are existence and value constraints, which are specified in an SQL-like notation. Enforcement of constraints exploits triggers and persistent queues. Whenever a local system updates a data item for which a constraint is defined, a trigger (derived from the constraint) fires. The action part of such a trigger enqueues a request on a local persistent
queue. A queue manager reads requests from the queue and executes the requested operation on a remote system. Such a request might, for example, propagate updates to remote databases in order to enforce dependencies.

An approach to constraint management using interfaces, strategies, and guarantees is described in [5] and amplified in [6]. Interfaces are associated to data items and define how that data item can be monitored, read, or written. For instance, a database can support notification or polling. Strategies specify how constraints are monitored and enforced. Strategy statements define that events (conditionally) trigger other events within a certain amount of time. Guarantees define the “level” of consistency in such cases where strong consistency cannot always be guaranteed (i.e., a constraint holds at any point in time). Guarantees are logical expressions defined in terms of events and predicates over data items.

Do and Drew [9] use a dependency graph to express relationships between data items (i.e., data items that are related by a causal constraint or an equivalence constraint). These constraints are then monitored and enforced by wrappers which use ECA-rules for checking and enforcing constraints.

Karabatis et al. [18] present an approach to describe and enforce dependencies between data elements across multiple data sources. A data dependency descriptor (D3) specifies source objects and a dependent target object, a predicate specifying the relationship between source and target objects, a consistency predicate, and an action component. The relationship predicate can specify, for example, that the target object is a copy of the source object (i.e., both are equal). The consistency predicate can specify, for example, when P must be satisfied (e.g., at the latest after an update to the source object). The action component defines how consistency can be restored. In this approach, consistency restoration actions are executed in polytransactions. D3s define a limit of discrepancy which defines how “far away” a target object might be in terms of its state and time. External updates are allowed only if the updated object remains within its zone of consistency, which in turn is defined as the intersection of the limits of discrepancy of all the D3s it participates in. Thus, external updates are restricted, and the FDBMS controls external updates. In any case, the FDBMS needs information about external updates (in order to allow or prohibit them).

Apart from database constraint maintenance, consistency constraints have also been used for other purposes, such as schema integration in federated and interoperable DBS [8,25].

An approach similar to our marking strategy has been proposed by Motro [20] for relational databases. This approach allows to model correctness and completeness properties in terms of views, and allows to annotate query results with information about these properties. The major difference is that our approach allows marking individual elements in a result whereas [20] only permits marking of the result as a whole.
6 Conclusions and Future Work

In this paper, we have introduced an approach to consistency constraints in database middleware in the presence of “weak” data sources with little or no support for constraints and related tasks. Different strategies for constraint management address a range of application requirements such as operating only on consistent data, or accessing also adequately annotated inconsistent data. Moreover, the strategies match the varying capabilities of data sources (concerning update notification and write access). In consequence, the spectrum of data sources for which (at least a restricted notion of) consistency can be maintained, is much broader than in recent approaches assuming database functionality as given by data sources.

Most of the strategies have been implemented and are operational. As the implementation is completed, we expect to extend the approach in several ways. First, in some cases it might be sufficient to check constraints locally (as has been proposed for distributed DBS [e.g., 15, 1]), so that constraints have to be checked globally only if the local check failed. This technique would improve performance of constraint management. Second, in many cases query processing performance can be improved when information about constraints is taken into account. For instance, query predicates can be a subset (or the negation) of constraint conditions, so that predicates are known to be redundant (or false). Considering constraint conditions during query rewrite thus is also a topic of future research.

7 References


