Enhanced Active Databases for Federated Information Systems

Inaugural-Dissertation

zur Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Christopher Popfinger
aus Landsberg am Lech

April 2006

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Stefan Conrad
Koreferent: Prof. Dr. Martin Mauve
Tag der mündlichen Prüfung: 30.06.2006

Preface

This thesis summarizes my research at the database group of the Department of Computer Science at the University of Düsseldorf, where I was working as a research assistant since October 2002. This work was primarily motivated by my diploma thesis that I wrote at the Ludwig-Maximilians-University of Munich, like this thesis also under supervision of Prof. Dr. Stefan Conrad.

I would like to thank all the persons that supported me in writing the thesis. In particular, I would like to express my sincere thanks to my supervisor and first referee Prof. Dr. Stefan Conrad for giving me the opportunity to work for my doctoral degree at his chair. I appreciate his support and confidence in my work and enjoyed working under his supervision. I would also like to thank Prof. Dr. Martin Mauve for his interest in my work and willingness to be the second referee.

I want to extend my compliments to my colleagues at the database group for the pleasant atmosphere, and especially to Cristian Pérez de Laborda, my longtime fellow student and coauthor of some nice papers, Evguenia Altareva, Johanna Vompras, and Tobias Riege for the stimulating tea brakes after lunch, as well as the members of the IGFZS community for the recreational activities.

Another word of thanks goes to the students that have contributed to my work with their bachelor theses, which are (in alphabetical order): Ludmila Himmelspach, Krasimir Kutsarov, Sandra Suljic, and Alexander Tchernin.

My special thanks go to Andrea Führer for supporting me throughout the time with her confidence and encouragement and for accompanying me through all the ups and downs.

Finally, I want to express my deepest thanks to my parents for their support, encouragement, and advise on my way so far, and especially for the care packages that made my time here a lot more comfortable.

Düsseldorf, April 2006

Abstract

Federated information systems provide access to interrelated data that is distributed over multiple autonomous and heterogeneous data sources. The integration of these sources demands for flexible and extensible architectures that balance both, the highest possible autonomy and a reasonable degree of information sharing. In current federated information systems, the integrated data sources do only have passive functionality with regard to the federation. However, continuous improvements take the functionality of modern databases beyond former limits. The significant improvement, on which this work is based on, is the ability of modern active database systems to execute programs written in a standalone programming language as user-defined functions or stored procedures from within their database management systems.

We introduce Enhanced Active Database Systems as a new subclass of active databases that are able to interact with other components of a federation using external program calls from within triggers. We present several concepts and architectures that are specifically developed for Enhanced Active Databases to improve interoperability and consistency in federated information systems. As the basic concept we describe Active Event Notifications to provide an information system with synchronous and asynchronous update notifications in real-time. Based on this functionality, Enhanced Active Databases are able to actively participate in global integrity maintenance executing partial constraint checks on interrelated remote data. Furthermore, we present an architecture for a universal wrapper component that especially supports Active Event Notifications, which makes it perfectly suitable for event-based federated systems with real-time data processing. This tightly coupled wrapper architecture is used to build up the DÍGAME architecture for a peer data management system with push-based data and schema replication. Finally, we propose a Link Pattern Catalog as a guideline to model and analyze P2P-based information systems.

Contents

Chapter 1

Introduction

1.1 Motivation

Since the first centralized databases found their way into the enterprises in the late 60s, the needs and requirements have changed towards a more distributed management of data. Today, there are many corporations and organizations that possess large amounts of databases, often spread over different regions or countries and typically connected to a network. These databases raised in an autonomous and independent manner to fit the special needs of users at the local sites which led to logical and physical differences. However, local applications produce or modify data that is often semantically related to data stored on other sources. The integration of these sources allows a company to keep track of its distributed data and thereby improving data quality and availability. One of the main challenges in such environments is the autonomy of the integrated sources. This autonomy implies the ability of a source to choose its own database design and operational behavior making it harder to integrate the source into a company-wide information system. Most companies have just started to see their distributed data as a valuable resource. Unfortunately, this data exists in various data models and formats. A study in 2004 [101] showed that although relational databases are by far the most popular databases in a commercial environment, other formats like flat files, XML, and object-oriented data sources are also widely-used in practice. The need for an integrated information platform is increasing steadily.

Federated information systems integrate information from multiple autonomous and heterogeneous data sources and provide centralized access to their data. Unlike distributed databases with homogeneous structures they allow the integrated sources to retain a certain level of autonomy. Thus, a federated architecture for distributed data has to balance both, the highest possible local autonomy and a reasonable degree of information sharing. Depending on the application field different federated architectures support different operations on the local

and global level but always have to address the problems imposed by distribution, autonomy, and heterogeneity to ensure consistency within the system.

In current federated information systems, the integrated data sources do only have passive functionality with regard to the federation. Like repositories they provide access to their data and do only respond to external requests. Although active databases are able to react on certain local events, their possibilities to actively support the federation are very limited or just nonexistent. Continuous improvements of (mainly relational) database systems and query languages resulted in the definition of SQL-invoked routines in the SQL-1999 standard. Those routines are stored procedures or user-defined functions that can be defined as external routines written in an external programming language like C or Java. We believe that this innovation takes data management and processing in federated information systems to a higher level. To fully exploit the new capabilities of these *Enhanced Active Databases* for information sharing in a federated environment, new techniques and architectures are required that are specifically designed for this data source activity class.

1.2 Overview

In this thesis we introduce Enhanced Active Database Systems (EADBS) as an extended class of active databases that are able to execute external routines written in an external programming languages to react on local events in a more complex way. We present concepts and architectures that are specifically developed for Enhanced Active Databases to support information sharing in federated information systems.

Following the introduction we start with a general overview of federated information systems and their characteristics in Chapter 2. We motivate the need for an integrated environment and introduce the theoretical background for the following chapters. We discuss distribution, autonomy, and heterogeneity as the main dimensions of federated information systems, summarize important integration challenges, that must be addressed during the system development, and sketch some well-known federated architectures.

In chapters 3 and 4 we provide an in-depth discussion on Enhanced Active Database Systems and our novel Active Event Notification mechanism as the basis for the following chapters. We first give a basic definition of Enhanced Active Databases and describe the enhanced activity that distinguishes this particular class of databases from others. The chapter also provides an overview of current databases with enhanced activity. Secondly, we present the concept of Active Event Notification that enables EADBSs to signal local data modifications to external components immediately after an update occurred. Besides a detailed description of this notification process, we provide a general overview of monitoring techniques and properties, and distinguish our approach from existing event

detection solutions.

In Chapter 5 we show how EADBSs are able to actively participate in global integrity maintenance in federated information systems. Active Event Notifications allow them to perform synchronous constraint checks on interrelated data stored on remote database systems. We introduce partial integrity constraints as a new type of constraints suitable for EADBSs and explain the checking mechanism for commonly used constraints. Furthermore, we present the COMICS constraint management architecture that extends the basic concept by introducing an external constraint manager that performs the remote part of partial constraint checks. EADBSs directly interact with the constraint manager during constraint checks using Active Event Notifications.

Since wrappers are an essential component in most federated information systems, we have developed a tightly coupled wrapper architecture for various types of data sources. Chapter 6 gives a detailed description of the wrapper that comprises an event detection subsystem which is used to extract data changes from the encapsulated source. It particularly provides a Notification Interface to support Active Event Notifications from Enhanced Active Databases, which makes the wrapper perfectly suitable for event-based information systems with push-based real-time event delivery.

Based on tightly coupled wrappers and EADBSs, we introduce the DÍGAME architecture for a peer-to-peer information system with push-based data and schema replication. Chapter 7 describes the basic functionality and components of this architecture, and discusses its major characteristics and application fields.

Finally, in Chapter 8 we propose a Link Pattern Catalog as a modeling guideline for recurring problems in information sharing environments like our DÍGAME architecture or similar P2P data management systems. We introduce the Data Link Modeling Language for describing and modeling data flows and describe commonly used Link Patterns and their applications.

Chapter 9 summarizes the concepts proposed and their contribution to the development of federated information systems. We conclude with an outlook on possible future work.

Chapter 2

Federated Information Systems

In this introductory chapter we describe the main characteristics of federated information systems (FIS) and the challenges that we face during their design and implementation. We start with a motivation for distributed information systems and answer the question, why it is not reasonable or feasible in many scenarios to maintain a centralized database. We continue with a description of distribution, autonomy, and heterogeneity as three important dimensions of an information system architecture, followed by a summary of concrete problems that must be addressed during the integration of autonomous and heterogeneous data sources. The chapter closes with an overview of selected integration architectures.

2.1 Why not one big database?

In general, an information system integrates multiple sources from several network nodes, which emerged autonomously to fit the special needs at a local site. Local applications typically imply specific hard- and software requirements with regard to the data sources. For example, the design department of a company could use a third-party CAD application to develop a new product. This CAD software possibly requires a specific type of database system to store its application data or ships with its own internal data management system. Other departments may have their own special purpose software to carry out their tasks, e.g. tools for accounting and billing, workflow management, or personnel administration. Furthermore, if a department retains a high level of autonomy, it is able to set up its own database systems according to its needs and abilities. For example, a department could favor a certain database system simply due to financial reasons.

Centralization of data holds many advantages considering technical or economic aspects. It limits the costs of redundant systems and increases data consistency and integrity based on uniform standards. However, besides the problem that most top down data planning efforts do not meet user expectations, techni-

cal considerations are only one factor regarding data centralization in a corporate environment. The more crucial factor for the success of an information system is the question of *data ownership* as discussed in [16, 116].

With the introduction of databases into business processes the traditional definition of data ownership has changes. Data ownership meant the total control over the creation, maintenance, and processing of the data. Subsequently, data sharing and data integration implied a loss of data ownership and with it the loss of total control over the content. Thus, the need for data sharing inevitably collides with the individual demand for ownership. According to Alystine et al. [116], a key factor for the importance of ownership is self-interest, which means that data owners have a greater interest in the success of an information system than non-owners. Thus, databases are maintained more conscientiously by their owners than by non-owners. Furthermore, they state that data quality can only be ensured if data ownership and data origination are not separated. Data should only be created and maintained by users with expert knowledge with regard to the application field. For example, consider a research department that is willing to share results of their experiments to the community. Cooperating departments should be able to process the data but not to manipulate it, since the results are specific to a certain experimental setup. If the department looses control over its data, it might not be willing to share further results.

In contrast to the ownership demands of individuals or individual departments, the company requires control of its data resources to know about the present use of the data, predict its future use, and constantly adjust that use to meet the company's goals. Data has to be considered as a corporate resource just like natural resources, finances, or personnel resources. If a company bears the costs of data, i.e. the costs for collecting, maintaining, and processing data, the company should be considered as the owner of that data. A decentralization of data means to delegate, transfer, and grant functional rights and privileges to individual departments and users so that these individuals can assist the company in achieving its goals. This decentralization gives the individuals a sense of data ownership, allowing them to plan and carry out their functional mandate autonomously.

Putting it all together, data ownership is the key reason for decentralization and autonomy. Interrelated data gets spread over multiple autonomous and possibly heterogeneous data sources. However, it is crucial for a company to keep track of its distributed data to make the right decisions and increase its productivity. An information system that accesses and processes this data has to address the problems arising from distribution, autonomy, and heterogeneity with regard to data ownership to ensure a high level of data quality.

2.2 Basic Architecture

Federated information systems integrate various autonomous and heterogeneous sources and provide access to their interrelated data. In the following we give a description of the basic architecture of a federated information system and its basic functionality based on [20].A *federated information system* consists of a set of distinct and autonomous information system components that give up part of their autonomy to participate in the federation and share parts of their information. An information system component offers one or more interfaces to its data that can be stored on a single computer or distributed over multiple, possibly heterogeneous network nodes. Single information systems can be database systems with DBMS or non-database systems like flat files, spreadsheets, or document collections without a standardized data model, predefined schema, or query language. However, non-database systems can be treated as database system if they enforce a strict format and offer some kind of declarative access language [1].

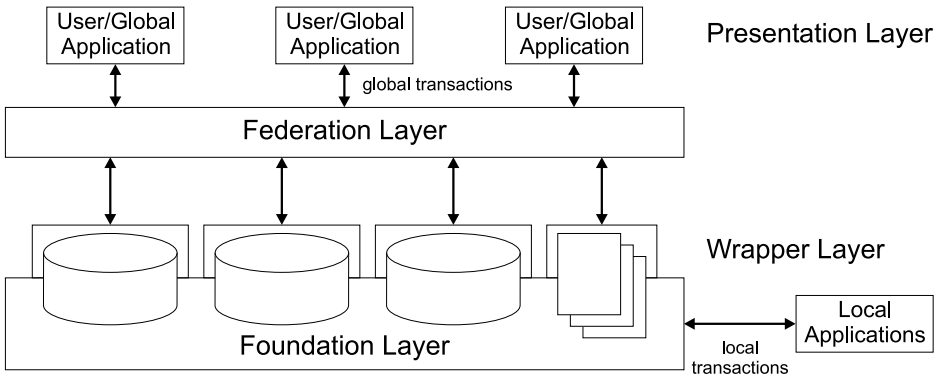

Figure 2.1: Basic architecture of federated information systems (based on [20])

Figure 2.1 depicts the general architecture of a federated information system. A federation layer provides uniform access to various autonomous and heterogeneous information system components that store interdependent data. These components usually are integrated in the infrastructure using wrappers to overcome source-specific differences to provide the federation layer with a uniform interface (wrapper layer). While the individual data sources may still be manipulated by local applications, global users and applications are able to access the integrated sources via the federation layer. The federation layer is a software component that implements a specific interoperation strategy that can be based on, for example, a federated schema, a uniform query language, or a set of source and content descriptors. The mechanisms depend on the composition and implementation of the federation layer. A selection of concrete FIS architectures can be found in Section 2.5.

In the following, a single information system component that is integrated

into a federated information system is called a *component database* or *component database system (CDBS)*. The data source comprised by a component database is called a *local database*. A *local transaction* is an operation that is submitted directly to a CDBS by a local user or application. It only affects data on the respective source. Contrary, *global transactions* (also denoted as *external transactions* from the point of view of a component database) are submitted by global applications and affect data stored on multiple CDBSs. In general, a global transaction is split into a sequence of local transactions which are executed on the affected component database. Query results from multiple sources are transformed into a common representation format and sent to the global user or application. The access to interdependent data using global transactions is managed by the federation layer of the federated information system. An explicit schema definition of a local database system or the implicit structure of a local non-database system enforcing a strict format is called the *local schema* of a component database.

Depending on the operations that are supported globally and locally we can distinguish between the following operational types of federated information systems:

- **Global read-only:** The information system provides read-only access to the integrated information sources. Data modifications are only supported locally preserving a high level of local autonomy of the participating component databases. The main challenge in a global read-only environment is a reasonable integration of the local schemas eluding the problem of global transaction management.

- **Local read-only:** Data can exclusively be modified globally via the federation layer. Local applications can only read interdependent data, revoking autonomy from the component databases. This requires the local schemas to be properly integrated into a global schema and a global transaction mechanism to execute updates. Due to the restrictions to local operations, the global transaction manager has full control of all update operations in the federation eluding the problems of global deadlocks and serializability.

- **Mixed:** This operational type allows data to be modified by both, global and local applications and is therewith the most complicated type. The system has to cope with the entire set of problems relating to database transaction management including serializability, deadlock detection, and atomic commit.

The concrete composition of the federation layer and the supported operations determine the problem areas to cope with during the integration of individual component databases. Before we deal with common integration challenges, we discuss the main dimensions of federated information systems in the next chapter.

2.3 Distribution, Autonomy, and Heterogeneity

To address the problems arising in a federated information system we first have to identify and understand the characteristics that impose them. The architecture of a federated information system can be classified according to three dimensions: distribution, heterogeneity, and autonomy. A fully centralized information system, for example, processes data from a single data source, whereas a fully distributed system could integrate multiple heterogeneous sources running autonomously on different network nodes. Each dimension implies advantages and disadvantages on the overall system, whereas the dimensions cannot be treated independently. A system with centralized data storage will surely not have to deal with heterogeneity and autonomy problems. The following short descriptions of each dimension are based on [52, 84, 102].

2.3.1 Distribution

Data of an information system may be located on a single data source or distributed among several databases on one or more physical machines. The benefits of a distribution of data clearly are the increase of availability and reliability and the improvement of access times. Furthermore, distribution is often required to satisfy data ownership and to adapt an information platform to the data policy of a company or organization (see Section 2.1). Data can be distributed over multiple sources in a non-redundant or redundant way. The non-redundant distribution requires a partition of the data, like horizontally or vertically partition in relational terms. Redundant storage of data, i.e. the replication of data, requires mechanisms in the information system to ensure the consistency of all replicas of a data item, when data is modified locally.

In a system of autonomous data sources the distribution of data is mainly introduced in an uncontrolled and unintended way. The data sources are often designed autonomously with regard to the local needs and requirements but store data that is interrelated to other sources. The main problems arising in an information system due to the distribution of data are the planning, scheduling, and execution of global read and write operations, allocation of data, maintenance of global integrity constraints, and replication management. Distributed databases address these problems in a homogeneous environment with a strong centralized control without any autonomy of the data nodes. Now, the addition of the dimensions autonomy and heterogeneity to the existing problems of data distribution imposes the main challenges of federated information systems.

2.3.2 Autonomy

The organizational structure of an information system generally reflects the organizational structure of the collaborating partners themselves. In many companies

the departments retain a high level of autonomy, which means that they are allowed to organize, execute, and monitor their tasks on their own. This autonomy directly effects the data sources managed by the department which means that data sources are often separately and independently controlled by the departments. In order to design and compose an information system that comprises several autonomous data sources, we have to understand and address the problems arising from the autonomy of the component databases. The autonomy of a component database of an information system is also termed as *local autonomy* [33]. A widely accepted classification of local autonomy is summarized in [102] which distinguishes between three types of local autonomy: design, communication, and execution autonomy.

Design autonomy is the ability of a local database to choose its own database design independently from the design of other component databases. This particularly means that they retain their local schemas during the integration process and that they cannot be forced to change their designs.

In particular, design autonomy allows a CDBS to freely choose

- the data it manages,
- the naming of data elements and the data representation including the data model and query language,
- the conceptual modeling of real world objects and the semantic interpretation of the data,
- the constraints to ensure consistency of the data,
- the set of supported operations for data access and manipulation, and
- the concrete implementation of the system.

The design autonomy is the main reason for heterogeneity in an information system. Especially the ability of a CDBS to choose its own conceptualization of real world objects leads to the problem field of semantic heterogeneity, which will be discussed later (see 2.3.3).

Execution autonomy enables a CDBS to execute local transactions without interference from external transactions. A system with execution autonomy cannot be forced to execute transactions according to a certain schedule by an external component. Operations may also be rejected at any time, for example, if they violate local integrity constraints. A data source with execution autonomy does not have to inform external components about the execution order of external transactions. Basically, the CBDS treats external transactions like local transactions. The problems arising from execution autonomy are mainly concerning global transaction management and consistency. Since a global component is unable to manipulate the scheduling and execution of transactions at a CDBS

with execution autonomy, it is unable to ensure a global atomic commit [77]. In particular, they cannot be forced to provide a prepare-to-commit state as required for multi phase commit protocols (e.g. 2PC). The third type of autonomy is *communication autonomy*. It allows a CDBS to decide when and how to communicate with other components. This includes the ability of a CDBS to join or leave the information system at any time. This particularly enables a CDBS to go offline at any time to rejoin the system again later on. The problem of volatile data nodes has especially to be addressed in information system architectures resembling peer-to-peer concepts. Further taxonomies and types of autonomy can be found in the literature, like operational autonomy and service autonomy [33], naming autonomy and transaction-control autonomy [40], or association autonomy [5]. They basically define subsets or combinations of the autonomy types listed above and can thus be described using the given classification.

As stated by Heimbigner and McLeod [52], the aim of an information system that integrates autonomous data sources is to achieve a feasible trade-off between local autonomy of the CDBSs and a reasonable degree of information sharing. Without the constraint of a central authority, information sharing is realized by cooperating component databases, which can communicate in three ways:

- *Data communication:* A component database provides access to its data or a subset of its data to other components directly. The information system thus has to provide mechanisms to support data sharing among the participating CDBSs.

- *Transaction sharing:* A component database may not allow other components to directly access its data. It rather provides a set of operations that can be executed upon its data stock. This requires components to be able to define transactions.

- *Cooperative activities:* Without the constraint of a central authority, the autonomous components need to cooperate to share information. They must be able to initiate, monitor, and control a series of actions that involve cooperation with other components using appropriate protocols (negotiated data sharing).

A cooperation certainly demands for agreements among the partners and in most of the cases it means restrictions to their local autonomy. Agreements among autonomous component databases for information sharing concern the data they are willing to share, the set of operations they support, but also additional cooperation parameters such as uptime guarantees.

2.3.3 Heterogeneity

The third dimension of an information system architecture is heterogeneity. As mentioned above, heterogeneity is mainly caused by the design autonomy of dis-

tributed, collaborating component data source. The ability of an administrator of a CDBS to choose the type of database including its data model and query language, as well as the individual conceptualization of the data leads to heterogeneity on the global system level. This heterogeneity has to be addressed during the integration of the CDBSs to provide a consistent global view on the partitioned data.

Heterogeneity can basically be divided into two classes, which are heterogeneity due to differences in the data sources and heterogeneity due to the semantic interpretation of the data. Both classes are described in the following sections.

Differences in data sources

Departments with a high level of local autonomy can choose their own database system depending on their specific environment and requirements, which leads to differences at the system level and in data models. Heterogeneity on the system level includes transaction management primitives and techniques like concurrency control, commit protocols, and recovery mechanisms. Furthermore, the hardware and operating system on which the CDBS resides may induce heterogeneity concerning file systems, data formats and representation, transaction support, or communication capabilities. System aspects are especially important during the integration of data sources without database management system, since the integration layer of the information system has to provide system-specific solutions for required operations, like file access or transaction management.

Heterogeneity in the data model describes the differences in the structures, constraints, and query languages. Different CDBSs can use different data models like relational, object-oriented, or semi-structured. Each data model provides different modeling constructs (e.g. inheritance and generalization in the object-oriented model) which will lead to different structures on the schema level. Even if two CDBSs use the same data model, the probability that a real world object will be modeled differently in the data sources will rise with the spectrum of available modeling constructs.

Besides differences in the structure, the data models may support different integrity constraints. Some integrity constraints might be inherent in one data model, but must be explicitly formulated in another one. For example, a specialization or generalization constraint could be expressed inherently in an object-oriented data model using an inheritance relationship, whereas in the relational model it must explicitly be expressed by a referential integrity constraint. Furthermore, active databases may use triggers to check complex constraints which cannot be expressed in a passive database although they might both be relational data sources.

Heterogeneity can also be caused by differences in the query languages that are used to manipulate data represented in different data models. Two CDBSs with the same data model might use different query languages or support different

versions or functionalities of the same query language.

Semantic Heterogeneity

A database can be considered as an image of the real world. During the database design, an administrator models real world objects in the database using the modeling constructs of the data model. To be more concrete, modeling an real world object means to name the conceptual image of the object (entity), to choose a set of attributes which describe the object regarding the specific requirements of the database application, and to assign a domain to each attribute. Furthermore, since real world objects can be related to each other, the administrator has to model relationships between entities in the database.

In a collection of autonomous databases each administrator may have different views on the same real world objects depending on its own understanding of things and the local needs regarding the data. This individual semantic interpretation of the data and its usage is the main reason for semantic heterogeneity. Unlike differences in data sources, semantic heterogeneity is harder to detect and to address. Differences can occur due to the selection and naming of entities and attributes as well as the selection of attribute domains or interpretation of attribute values.

2.4 Integration Challenges

As described in the previous section, the three main characteristics of an information system architecture are distribution, heterogeneity, and autonomy. These dimensions impose several problem fields during the integration of autonomous and heterogeneous data sources into an information system. This section describes the main integration challenges which have to be addressed by information system architects to create a reliable system assuring a high level of data quality and consistency. A more detailed overview can be found in [30].

2.4.1 Schema and Data Integration

The first and probably most difficult problem is integration conflicts resulting from the heterogeneity of the integrated data sources. Differences in data sources and semantic heterogeneity lead to different views and models of the same real world objects in the CDBSs. Database designers might have individual information needs and use their own tools to satisfy them. During the integration process, these differences must be dissolved to provide a uniform global view (*global schema*) on the entire data and to enable system interoperability and data sharing. The basic approach to build a global schema is to select several independently developed schemas from component databases with interdependent

data (*local schema*), resolve syntactic and semantic conflicts among them, and create an integrated schema comprising all their information.

A model-independent classification of integration conflicts is presented in [105]. This taxonomy distinguishes four conflict classes: semantic, descriptive, heterogeneity, and structural conflicts. They are briefly discussed in the following.

Semantic Conflicts: Two database designer might have different perceptions of a set of real world objects (entities). An object class `employees` might in one CDBS be used to represent employees of the entire company, but in another CDBS it might represent only employees of a single department. Although the object classes could be semantically equivalent in both CDBSs, they represent different sets of real world objects. The extension of two object classes can be disjunct, equivalent, overlapping, or a class extension can be strictly included in another.

Descriptive Conflicts: Descriptive conflicts arise from different conceptualizations of the same set of real world objects. Two database designer might be interested in different properties of the same object and thus create schemas with different sets of attributes. Descriptive conflicts also include naming conflicts due to homonyms and synonyms, attribute scale, domain, constraints, and operations.

Heterogeneity Conflicts: Database designers could use different data models for their databases which results in heterogeneity conflicts. In general, the integration of heterogeneous data models also implies structural conflicts since the data models provide different constructs to model real world objects.

Structural Conflicts: Even if designers use the same data model, there might be structural differences between the schemas. The same real world objects can be modeled using different modeling constructs. The more constructs are available, the more possibilities the designers have to represent the same object. As an example for a structural conflict consider the star and snowflake schema as relational representations of a multidimensional data model for data warehouses. Although both schemas store the same information, the snowflake schema uses normalized tables to reflect hierarchies in the dimensions whereas the star schema has a single non-normalized table for each dimension, but with redundant storage of information.

A well-known approach for schema integration uses assertions as the main concept. Assertions (or *mappings*) express correspondences between the schemas or parts of the schemas to be integrated. They define dependencies and integration rules for the schemas as well as transformation rules for the corresponding data instances. Thereby the mappings can be defined between a local and a

global schema or between two local schemas as required for information systems without a global schema (see 2.5). For detailed information on integration using assertions we refer to [104, 105]. From the exhaustive list of work in this area, we only want to present a small selection. [42], for instance, discuss a method for schema integration that detects class similarities by comparing previously enriched schemas along the generalization, specialization, and aggregation dimension. Similarly, [34] proposes a simple unified language for the specification of three fragmentation conflict types (classification, decomposition, and aggregation conflicts) together with techniques to solve them. Schema integration using a global data structure is presented in [18]. They propose the Summary Schemas Model to aid semantic identification. Users access local data via imprecise queries on the global schema whereas the system matches the user's terms to the semantically closest system terms. A similar approach is discussed in [36], where a semantically expressive common data model is used to capture the intended meanings of conceptual schemas. This Kernel Object Data Model describes structures, constraints, and operations on the shared data. An approach for the integration of integrity constraints is presented in [31]. The authors apply rules to a set of elementary operations for schema integration and restructuring. Finally, [75] present an algorithm that discovers mappings between schema elements based on their names, data types, constraints, and schema structure using linguistic, structural, and context-dependent matching techniques.

2.4.2 Entity Resolution

Semantic heterogeneity in federated information systems imposes multiple challenges considering the different representations of real world objects in the local databases. If two database designers model overlapping views on the same real world entity, the resulting schemas will store redundant information concerning this entity. The problem field of entity resolution (also referred to as record linkage or deduplication [14]) in the context of federated information systems deals with the identification of corresponding records referring to the same real world entity in multiple databases, possibly with different schemas.

The aim is to merge corresponding records into one record with more complete information. For example, two departments of a company could store customer information in their autonomous local databases. During the integration of these customer databases into a company wide information system with one global customer database, corresponding records that refer to the same customer have to be merged and duplicates have to be removed to ensure a consistent customer data stock. This join can already be problematic if the customers are globally identified by a company wide key (e.g. a customer number), but most often there will not be a unique key that can be used to join the records.

The basic mathematical model for entity resolution was introduced by Fellegi and Sunter [37]. Suppose the records are stored in the sources A and B. Fur-

thermore, an individual real world entity is assumed to be identified by multiple attributes (key attributes) of a record in A and B, like name, address, date of birth, and gender. Two disjoint sets M and D are defined from the cross-product $A \times B$ and denote the sets of record pairs (a, b) from A and B that could be matched ($(a, b) \in M$) and those pairs that could not be matched ($(a, b) \in U$). The record linkage process tries to determine if a pair belongs to either M or U. One of the standard algorithms for computing this task uses a probabilistic model with expectation-maximization to calculate probabilities for a match or non-match of a record pair by comparing the values of the key attributes [124, 48]. A comparison or agreement vector γ represents the level of agreement between a and b by calculating the matching weights of their key attributes. Attributes can be weighted in the comparison depending on their importance or value distribution. The composite weight (or score) for a comparison vector γ is calculated using the conditional probabilities for a match $m(\gamma)$ and a non-match $u(\gamma)$ as defined as:

$$m(\gamma) = P(\gamma \mid (a, b) \in M) \qquad \text{and} \qquad u(\gamma) = P(\gamma \mid (a, b) \in U)$$

Given two threshold values T_μ and T_λ, a record pair (a, b) is classified using its score $S = \frac{m(\gamma)}{u(\gamma)}$ as follows:

$$\begin{array}{lll} (a,b) & \text{is a match if} & T_\mu \leq S \\ (a,b) & \text{is a potential match if} & T_\lambda < S < T_\mu \\ (a,b) & \text{is a non-match if} & S \leq T_\lambda \end{array}$$

The algorithm performs multiple blocking passes for non-match record pairs in which it selects one or more blocking attributes to calculate the matching weight. If an attribute value distribution for a field is not uniform, the value can be weighted. The following overview based on [48] briefly concretizes the challenges that arise during the record linkage process:

Standardization: Without standardization, many records could be wrongly classified as non-matches due to typographical errors (e.g. 'stret' instead of 'street'), homonyms and synonyms (e.g. 'name','lastname','fullname'), or alternating representations for the same concept (e.g. 'M/F' or '0/1' for a 'gender' attribute). During the data cleaning process, attribute values are transformed into a standardized representation and spelling or combined with other values to satisfy global standards.

Attribute Selection: This problem field concerns the selection of common attributes on which the matching weight is calculated. This requires to identify common attributes or the optimal subset of common attributes that have sufficient information content to support the linkage quality.

Comparison: The actual comparison of two attributes is mainly based on distance-based metrics to compute matches. Since text or string values are most commonly used as matching attributes, this problem field deals with the development of efficient string comparators. Well-known comparison techniques are based on the edit-distance, N-gram distance, vector space representation of fields, or adaptive comparator functions.

Decision Model: After matching weights of individual attributes are calculated they have to be combined to a composite score to determine if the record pair is a match, non-match, or possible match. This classification is performed using a decision model. Besides the probability model described above, other models are proposed, like statistical models that compute statistical characteristics of errors or predictive models for learning threshold values and attribute weights.

The linkage of corresponding records is essential for the consistency of the overall system. Only if two records can be classified as a match or non-match the system is able to perform global integrity checks which are essential for the data quality in the information system.

2.4.3 Global Integrity

The integration of autonomous and heterogeneous information system components into a single federated information system inevitably raises the question of how to ensure consistency of the data from a global point of view. For example, two CDBSs might maintain locally consistent data stocks but might store controversial information about a real world entity from a global point of view. These conflicts must be resolved during integration and prevented in the future by the federated information system to ensure high data quality. Consistency in a federated information system can only be violated by write operations on interdependent data. Data on a local database that is not interrelated to remote data on another CDBS can be modified in accordance with local integrity constraints and does not compromise global consistency. Thus, we restrict our further considerations to operations that modify interrelated data locally and globally:

Local data modifications: A local data modification is a local transaction that inserts, deletes, or updates objects in a local database which are represented by a local schema. An update operation modifies one or more attribute values of a local object.

Global data modifications: A global data modification is a global transaction that is issued against a global schema. The insertion, update, or deletion of a global object results in a sequence of local write operations executed on the affected local databases.

To ensure global consistency, the information system has to monitor and check local and global write operations against global integrity constraints that express the dependencies among the interrelated data. While global transactions can easily be monitored by the federation layer, the detection of constraint violations caused by local transactions and their compensation is a major problem of information system engineering. On the global level, we distinguish between the following three types of integrity constraints that result from different situations and impose different challenges for integrity maintenance in an information system:

Constraints resulting from schema integration: The global integrity constraints result from the integration of local schemas. The CDBSs might model equivalent, overlapping, or disjunct views of a real world entity. The information system has to ensure that objects are stored correctly in their corresponding extensions. For example, if an object is inserted into an extension that is semantically equivalent to an extension on another CDBS, the object must be inserted into all affected extensions. If the insertion fails in one extension, then the overall operation must be rolled back. Semantically equivalent and overlapping local extensions require the system to maintain replicas of objects to ensure consistency. Update operations on equivalent or overlapping extensions must be executed concurrently on all replicated objects.

Constraints derived from local constraints: Local integrity constraints belong to the semantics of the data and must also be enforced on the global level. They can be separated into *explicit* and *implicit* constraints. Explicit constraints are formulated and managed separately by the DBMS whereas implicit constraints are inherent properties of the data model. If the implicit constraints are not supported in the global data model, then they have to be translated and managed explicitly by the federation layer (see [31] for details).

Additional global constraints: Additionally to the constraints resulting from schema integration and the integration of local constraints, there might be further constraints that are formulated explicitly on the global schema to enforce certain rules on the interdependent data. This could be global key or aggregate constraints that express specific business rules or existence dependencies between local extensions. The information system must be able to monitor and enforce explicit integrity constraints to ensure consistency.

As already mentioned, an essential task of global integrity maintenance is the detection of local write operations on interdependent data. The concept we propose later in this thesis (see Chapter 5) addresses this problem using the functionality of Enhanced Active Database systems.

2.4.4 Global Transaction Management

The last challenge we discuss is the planning and execution of global transactions. Global transactions are issued against a global schema and affect data of at least two different component databases of the federated information system. In general, a global transaction is decomposed and executed as a sequence of local read and write operations (subtransactions) for each affected local databases (*nested distributed transactions* [84]). A transaction management component in the federation layer has the responsibility to generate this decomposition and to monitor the execution of the individual local transactions. Like transactions in a single information system, global transactions in a federated information system should apply to the following ACID properties to ensure a consistent and reliable system [84]:

Atomicity: A transaction is always executed as a single unit of operations. Either all actions of a transaction are completed or none of them. If one action cannot be completed successfully then all other actions of that transactions must be taken back. In particular, intermediate results during the execution of a transaction that is not yet completed successfully must not be visible to other transactions.

Consistency: A transaction must map one consistent database state to another and may not violate the consistency of the data stock. Therefore, a transaction is checked against existing integrity constraints and aborted if one of them is violated.

Isolation: This property requires a transaction to see a consistent database at all times. A transaction cannot reveal its result to concurrent transactions before its commitment. This ensures that a transaction does not access data that is concurrently updated by another transaction.

Durability: Once a transaction commits, its results are stored permanently in the database and can be processed by subsequent transactions. The durability of a transaction also refers to the ability of a system to recover the last consistent state after a system failure.

While these properties are well-understood and guaranteed in centralized or homogeneous distributed database systems, their implementation impose great challenges in an environment of heterogeneous data sources with high level autonomy. In the following we describe the three major problem fields concerning transaction management in federated systems based on [17]:

Global Serializability: Since each participating component database may use its individual concurrency control protocol, existing serialization protocols

for homogeneous distributed databases cannot be used. If the global transaction manager is unaware of local transactions then it can only guarantee a serial execution of global transactions which does not automatically guarantee global serializability due to indirect conflicts caused by local transactions.

Global Atomicity: The atomicity property of a transaction dictates that all subtransactions of that transaction are either committed or aborted. In a homogeneous distributed environment this is ensured using an atomic commit protocol such as 2PC (two phase commit protocol). It requires that the participating data sources provide a *prepare-to-commit* state for each subtransaction and guarantee to remain in this state until a coordinator sends a global commit or abort. Obviously, this strongly limits the execution autonomy of the CDBS and not all data sources are able to provide a prepare-to-commit state. So, if execution autonomy should be preserved, we cannot force a CDBS to export a prepare-to-commit state. However, this will allow a CDBS to abort a subtransaction at any time before it is committed resulting in non-atomic global transactions and incorrect global schedules. As stated in [77] an atomic commit in an environment of autonomous and heterogeneous environments is impossible without either violate local autonomy, limit the types of transactions allowed, or using a new or relaxed transaction/correctness model.

Global Deadlocks: The third major problem field concerns the detection and prevention of global deadlocks. In an autonomous environment where the participating component databases use locking mechanisms to ensure local serializability, there can be a sequence of subtransactions that leads to a global wait-for-cycle that results in a global deadlock. To detect and prevent deadlocks, the global transaction manager needs information about local transactions and locks. On the other hand, since CDBSs are unwilling to exchange their control information with the federation layer (design autonomy) they will be unaware of global deadlocks.

Solutions to global integrity and global transaction management require information on the state of the participating data sources. As the main contribution of this thesis, we describe how the functionality of Enhanced Active Databases can be used to interact with other component databases or the federation layer to signal changes of their states to coordinate their actions. They particularly support immediate notifications perfectly suitable for real-time information systems.

2.5 Common Integration Architectures

The basic architecture presented in Section 2.2 gives only a general overview of the composition of a federated information system. In detail, the design and

implementation of the federation layer including the specific interoperation strategy, the supported operations and transactions, the integration strategy, as well as the wrapper functionalities and supported data sources may vary strongly with the intended application field. An important criteria for the classification of a federated information system is the composition of its federation layer. In the following we present three architectures for building federated information systems with different levels of distribution of their federation layer: centralized, modular, and fully decentralized.

2.5.1 Federated Database Systems

A *Federated Database System* as defined by [102] is a collection of autonomous but cooperating database systems that are integrated into the federation and controlled by a federated database management system (FDBMS). Components give up parts of their autonomy to participate in the federation depending on the needs and desires of federation users and administrators. The FDBMS implements DBMS functionality of centralized or distributed database systems and controls access and manipulation of the data on the integrated component databases. It provides location, distribution, and replication transparency and commonly supports query language access to the data. Supporting read-write operations it contains a query processor and optimizer, as well as a global transaction manager to ensure global consistency while allowing concurrent updates across multiple databases. The FDBS maintains either one single or multiple federated schemas which are mapped to the local schemas of the structured sources.

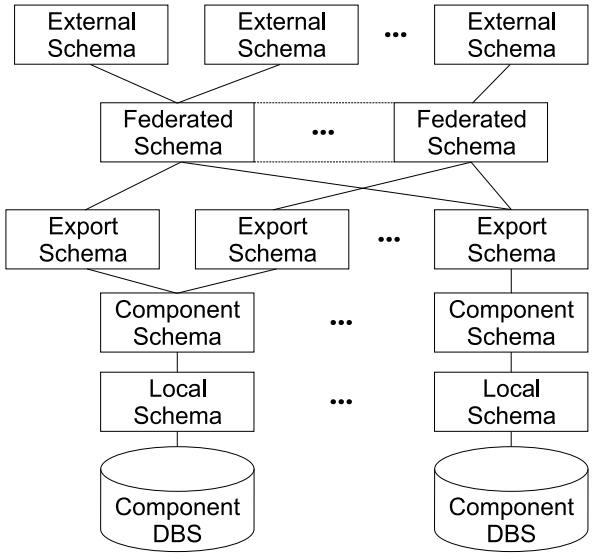

Figure 2.2: Five-level schema architecture of an FDBS [102]

Figure 2.2 depicts the well-known five-level schema architecture of an FDBS. The dashed lines between the federated schemas symbolize the option of a single or multiple federated schemas. Referring to the general architecture of federated information system (Figure 2.1) the component and export schemas are defined and maintained by wrapper component providing the federation layer with a common data model and query language. The global federated schema(s) and the export schemas reside in the federation layer. The federation layer of an FDBS typically consists of a non-distributed, monolithic federation service implementing the DBMS-like functionality. The static structure of FDBSs makes it harder to add or remove components or to react on changes in their schemas.

2.5.2 Mediator-based Information Systems

One of the first descriptions of mediator systems was introduced in [103] as a "pseudo intelligent software controller which [...] mediates between an Information Retrieval System and its end-user". This basic three-layer architecture consisting of users, mediators, and data sources is also reflected in [120] where mediators are defined as small and simple active software modules that implement dynamic interface functions between users workstations and database servers. Typical tasks performed by mediators are

- transformation and subsetting of databases,
- abstraction and generalization of underlaying data,
- providing intelligent directories to information bases, and
- providing access and merge data from multiple source.

Each mediator implements a specific set of mediation functions using one or a few databases. A user task will most likely need multiple distinct mediators to accomplish. Figure 2.3 depicts the basic architecture of a mediator-based information system.

A data source is typically encapsulated by a wrapper component to provide a homogeneous interface to the mediation layer. They convert query results into a common (or canonical) data model before they are sent to the mediators. The mediators communicate with one or a few wrappers to accomplish a specific mediation task that is offered to the application layer. Mediators can also use the functionality of other mediators as a data source, thus supporting a hierarchical composition of the mediation layer. Each mediator maintains its own federated schema which is ideally represented in the common data model and supports a common query language. The users now access the mediators that offer the required functionality to access the sources. The data is preprocessed using the mediation functions and returned for individual processing.

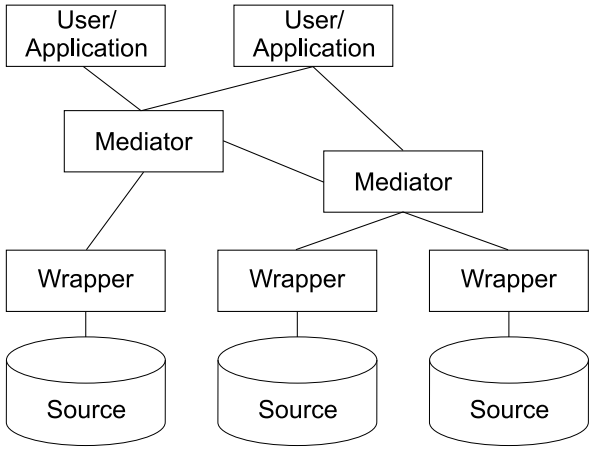

Figure 2.3: Architecture of mediator-based information systems [41, 20]

Contrary to federated database systems, mediator-based systems incorporate a modular structure and abstain from centralization. A data source can be accessed by multiple mediators whereas new mediators and sources can be added to the system at any time making the system more dynamic than FDBSs. On the other hand, mediators typically provide read-only access to their data sources, since decentralization complicates global transaction management and concurrency control.

2.5.3 Peer Data Management Systems

Peer-to-Peer (P2P) information systems or *Peer Data Management Systems* [49, 112] resemble concepts and mechanisms for fully decentralized sharing and administration of data. P2P systems are highly dynamic and scalable allowing autonomous and heterogeneous network nodes (peers) to join or leave the network at any time. The system gets more flexible than mediator-based systems, since there are no central global components that have to be maintained by administrators. Peers store data that the users are willing to share with other participants. Although not necessarily required, many P2P network topologies use *super-peers* to increase network performance (see Figure 2.4). Super-peers are used for peer aggregation, query routing, or query mediation [80].

Since P2P systems are decentralized there exists no global schema but a collection of pairwise mappings between peer schemas that are typically created using schema mapping languages (e.g. [50]). Contrary to FDBSs or mediator-based systems, the schema mediation does not follow a tree-like integration hierarchy with source schemas at the leaves and mediated schemas as inner nodes but an arbitrary graph of interconnected schemas. The set of mappings defines the semantic network (or topology) of the system. Queries are reformulated using the

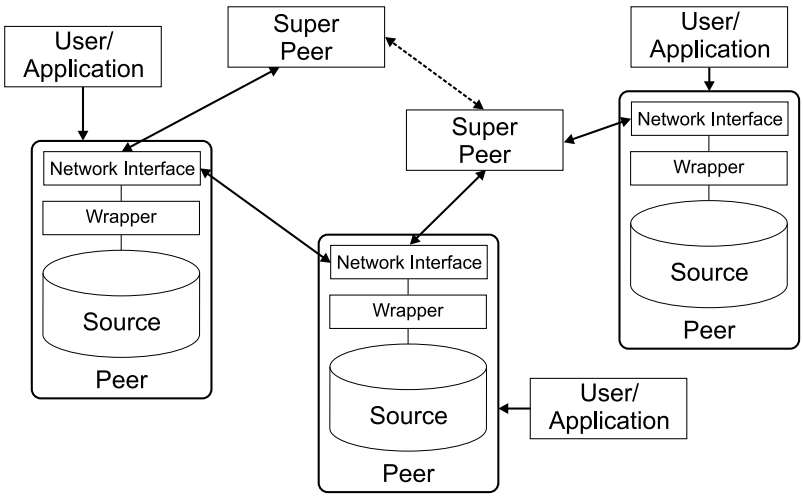

Figure 2.4: Architecture of Peer-to-peer information systems

schema mappings and routed to all the peers that might have answers. In turn, results are converted along the schema mappings from the remote into the local representation. Semantic related results from multiple peers are integrated either completely at the peer that issued the query or using intermediate integration results created by super-peers.

Referring to our basic architecture for federated information systems, the federation layer is fully decentralized and distributed over the participating peers. Each peer data source is wrapped to provide a homogeneous query interface whereas pairwise connections between peers are established via P2P network interfaces. Each peer decides autonomously which data it is willing to share and maintains its own set of schema mappings. In general, a P2P system supports read-only operations with the option to cache query results locally. However, in Chapter 7 we describe an architecture for a P2P-based information system that implements a push-based replication strategy among autonomous and heterogeneous peer databases.

Chapter 3

Enhanced Active Database Systems

Enhanced Active Databases build the basis for the concepts we propose in this thesis. In this chapter we introduce Enhanced Active Database systems and present their specific functionality that contributes to solutions to common problems in federated information systems. After a definition of Enhanced Active Databases in Section 3.1, we describe the new functionalities that can be added to component databases using External Program Calls. We introduce remote state queries, injected transactions, and external notifications as three new operations of component databases that enable the interaction of component databases within a federation. A detailed description of an External Program Call is subject to Section 3.3. We present the required components and explain the basic steps to communicate with external components. Section 3.4 discusses the effects of External Program Calls on the local autonomy of the component databases. Finally, the chapter is closed with an overview of current Enhanced Active Databases that are widely used in practice to confirm the practicability of our concepts.

3.1 Definition

Traditionally, database systems have been regarded as passive data providers that manage the storage of data and response to read and write requests issued by the users. More complex requirements regarding the integrity and consistency of the data had to be implemented in the applications themselves. But with the association of databases to highly complex information processing scenarios, with huge amounts of data or high performance requirements, database systems were extended by more comprehensive facilities to model structural and behavioral aspects of data to support the applications. Active database systems were introduced that assist applications by migrating reactive behavior from the application to the DBMS. They are able to observe special requirements of ap-

plications and react in a convenient way if necessary to preserve data consistency and integrity. The integration of active behavior in relational database systems is not particularly new and currently most commercial database systems support ECA rules, whereas the execution of triggers is mainly activated by operations on the structure of the database (e.g. insert or update a tuple) than by user-defined operations [86]. Unfortunately, the ability to react on events, especially from within the scope of trigger conditions and actions, has until recently been limited to the isolated databases they were defined at. Subsequent developments integrated special purpose programming languages (e.g. PL/SQL [74]) into the database management system to overcome some limitations of the query language and to provide a more complex programming solution for critical applications. But again, the scope of these extensions was strictly limited to the system borders of the database system, so an interaction with its environment was impossible. However, the support of ECA rules in form of triggers is necessary, but not sufficient for the concepts we propose here.

The significant improvement, on which this work is based on, is the ability of modern active database systems to execute programs written in a standalone programming language as user-defined functions or stored procedures (also referred to as external routines) from within their database management systems. This enhancement takes the functionality of active databases beyond former limits. Thus, we define a new subclass of active databases as follows:

Definition 1 *The ability of a database system to execute programs or methods from within its DBMS to interact with software or hardware components beyond its system border shall be called* enhanced activity. *A database with enhanced activity is a* Enhanced Active Database System *(EADBS). The execution of a program or method in this context shall be called an* External Program Call *(EPC).*

The execution of external programs (EPs) from inside the DBMS offers new perspectives to data management and processing in a federated information system. The database has herewith access to the entire functionality of the programming language including user-created libraries and extensions. EADBSs are active databases that are actively able to invoke methods or programs from within their database management system. An Enhanced Active Database that participates in a federation as a component database can offer its enhanced activity to improve interoperability in the federation. Which particular functionality can be provided by such component databases is presented in the next section.

3.2 Enhanced Activity

The enhanced activity allows Enhanced Active Database systems to execute external programs to interact with hardware or software components beyond the

system borders of the database. In the context of federated information systems, this functionality allows component databases with enhanced activity to communicate with specific components of the federation, like, for instance, a wrapper component, a constraint or transaction manager, an event broker, or another component database or additional data source. Communication is established using the APIs and libraries provided by the programming language that is used to code the external program. In particular, we focus on the database connectivity and client-server APIs for sockets or remote procedure calls (RPC) to add the following functionalities to a component database that participates in a federated information system:

Query the state of a remote database: The main functionality which is elementary for our approach is the ability of an CDBS to query a remote data source *directly* during the execution of a database trigger. After a connection has been established by the EP, we can perform any read operation on the remote schema items we are allowed to access. Depending on the query language we can formulate complex queries with group and aggregate functions (e.g. like in SQL). The query result of the remote database is used locally to evaluate conditions of ECA rules. We call this kind of query a *remote state query*.

Manipulating a remote database: After a connection is set up by the program, a CDBS is basically able to modify the data stock of the remote database *directly* during the execution of a database trigger. Assuming the appropriate permissions, any operation supported by the query language can be executed including data insertions, updates, and deletions. Depending on the query language, a CDBS is thus basically able to modify even the schema of a remote database using for example `ALTER TABLE` statements in SQL. In the following, a manipulation of remote data or schema items from within a database trigger shall be called an *injected transaction*, since its execution depends on a triggering transaction on a local relation. From the point of view of the remote database, a remote state query or injected transaction is handled like a request of an ordinary application.

Notification of external components: Besides the database connectivity we use client-server APIs of the programming language to establish a connection to a remote server component of an arbitrary software application. The database acts as a client and opens a communication channel via sockets or remote procedure calls. Thus, it is able to interact with remote applications and use their services during the execution of triggers or stored procedures. In particular, those connections are used to send notifications to external components via *External Notification Programs* (ENP), to either simply signal the manipulation of local data or to actually propagate the modified data items themselves.

Within recent commercial database systems a commonly supported programming language that provides the technology we need to implement these enhanced functionalities is Java. It contains JDBC, a common database connectivity framework, that provides a standardized interface for a multitude of different data sources like relational databases or even flat files. JDBC is part of the Java core API since version 1.1 and is supported by all major database manufacturers [119]. Furthermore, it comprises APIs to establish client-server connections via sockets or Remote Method Invocations (RMI). Its platform independence is particularly useful in an heterogeneous environment such as a federated information system. Java functions can be migrated between component databases without much code rewriting. Remote state queries and injected transactions are executed via JDBC using SQL as the standard query language bridging the heterogeneity. Although we cast the remainder of this work in the context of Java UDFs using JDBC and RMI the concepts certainly adapt to Enhanced Active Databases supporting other programming languages that meet the requirements just mentioned.

3.3 External Program Calls

The enhanced functionalities like remote state queries, injected transactions, and update notifications are realized using external program calls, which are described in detail in the following. Although an external program could also be explicitly executed by a user as a stored procedure, we focus on external program calls from within database triggers as part of a database transaction. The de-facto standard for managing and querying databases is SQL, currently in the version SQL-2003. Its predecessor SQL-1999 has already defined the concept of SQL-invoked routines in the form of stored procedures and user-defined functions (UDF) [6]. The standard allows both types to be defined as external routines in an external programming language like C or Java. Such routines are already supported by major database systems (e.g. Java Stored Procedures or Java UDFs [73, 78]). They can be called from triggers during their execution as part of a database transaction.

Figure 3.1 displays a schematic overview of an external program call. In general, triggers are activated by transactions that execute write operations (updates) on the data stock. In our example, an update on a relation R fires a trigger that in turn sequentially calls one or more external programs. The EPs interact with external hardware or software components, making requests and eventually waiting for responses. After the EPs terminate, the trigger returns its call and results in a *commit* or *abort* of the corresponding transaction. The execution of a trigger including the external programs is typically synchronous, i.e. the DBMS holds a lock on the affected data until the trigger terminates. The concrete locking mechanism strongly depends on the implementation of the concurrency control protocol and thus varies with the database management system.

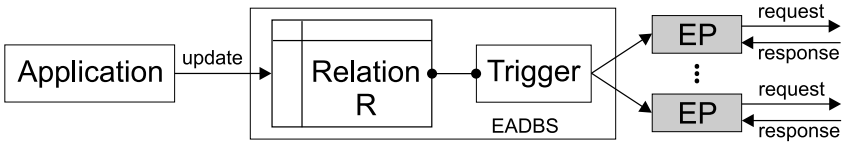

Figure 3.1: Schematic overview of an external program call

Obviously, to use an external program practically, we must be able to pass parameters to the EP and to access the corresponding program output from inside the trigger. This output can be used to evaluate trigger conditions or to determine subsequent trigger actions. Since the EPCs are embedded straight inside the DBMS of the local system, we are able to delay or abort transactions depending on the result of an external program call. Just like common triggers that exclusively use local data to evaluate their trigger conditions, the DBMS autonomously schedules the execution of the trigger that encapsulates the EP. In particular, we do not force a component database to provide an atomic commitment protocol like 2PC (see 3.4 for a discussion).

We now illustrate the call of an external program using a simple example. Unfortunately, the concrete statements and mechanisms to load and register external programs in a database vary among different database products. Thus, we use Java and the database DB2 as concrete representatives to give an example for an EPC. Before an external program can be called from a trigger it has to be loaded into the database and registered as a user-defined function or stored procedure. The program must define a specific method, procedure, or function that should be callable by the database and execute the required operations. As an example, we assume that the following Java function `someClass.someFunction` shall be registered in the database:

```
public class someClass {
  public static int someFunction (String arg1, int arg2) {
    int result=0;
    // do something with param1, param2, and result
    return result;
  }
}
```

The function takes two arguments `arg1` and `arg2` with the given data types. It calculates an integer as a function value of the parameters. Depending on the database system, the compiled class `someClass.class` can be loaded directly into the database or it must be added to a Java archive first. We assume the class to be included into a jar file `ep.jar`, that is loaded into the database using a database-specific mechanism and registered as archive `ep`. Thereafter, the external function has to be registered as a Java UDF in the database. The following

statement exemplarily creates a new UDF in a DB2 database and maps it to the `someFunction` function:

```
CREATE FUNCTION someUDF (arg1 VARCHAR(255), arg2 INT)
RETURNS INTEGER LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'ep:someClass.someFunction'
EXTERNAL ACTION DETERMINISTIC FENCED NO SQL
```

The statement specifies the signature of the function (name, arguments, return type) and additional parameters required by the database system to successfully register the function as UDF, e.g. the language type security settings. The UDF is mapped to the external function in the registered jar archive `ep`. It is now accessible from within the database and can particularly be called by a trigger. The following trigger is executed *before* an update occurs on the relation $R(A, B)$, with A and B being string and integer attributes respectively:

```
CREATE TRIGGER onUpdate
BEFORE UPDATE ON R
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
IF ( someUDF(n.A, n.B) = -1 ) THEN
  SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT='Error';
```

The trigger calls the external program `someUDF` for each updated row in R, while the update transaction is blocked by the DBMS and waits for the trigger to return. The update operation only commits, if the function call for the updated values of A and B does not yield -1. Otherwise an error is raised and the corresponding update transaction is aborted.

3.4 Discussion

We now discuss the restrictions to local autonomy that are induced by the execution of external programs. We refer to the well-known classification of local autonomy summarized in 2.3.2.

The execution of external program clearly violates local design autonomy, since the creation of triggers and user-defined functions requires changes to the database system. Communication autonomy, as it is defined in [102], allows the CDBS to decide when and how to respond to external requests. According to this definition, EPCs do not impose restrictions to communication autonomy, since we do not force to answer requests immediately. As we will see later (see Chapter 5), a component database may go offline during a global integrity check without compromising global integrity, if pessimistic checks are implemented.

Furthermore, a request via an EPC is initiated by the database itself without being forced to execute, which allows a CDBS to retain a high level of execution autonomy. The DBMS decides when to schedule a local transaction, the execution of a trigger or an external program. EPs do not interfere with local serialization of transactions or concurrency control. Like local constraint checks implemented by triggers, the DBMS waits for the termination of the EP that returns a value to commit or abort a transaction. A timeout value typically limits the time that a trigger, function, or procedure may take to execute. The decision to abort the execution of a trigger belongs exclusively to the DBMS.

As already motivated in the previous chapter, reasonable information sharing in a heterogeneous and autonomous environment demands for certain arrangements and assurances among the partners. The enhanced activity allows a component database to interact with other components of a federated information system while retaining a high level of local autonomy. Using triggers we are able to commit or abort a local transaction depending on the state of a remote data source. In particular, using Java with JDBC and SQL we can implement portable solutions to overcome heterogeneity in different types of data sources. As we will see in the next section, there is a comprehensive list of current EADBSs that support Java as a standard language for coding external procedures and functions.

3.5 Current EADBS

In this section we give a brief overview over current Enhanced Active Database systems and their supported programming languages. Commonly supported programming languages meeting the requirements of data connectivity and client-server connections are C, C++, and Java. Like Java comprising JDBC as a database connectivity framework, C and C++ support data connections via the ODBC (Open Database Connectivity) interface using SQL as query language. Besides, we are able to implement client-server connections which makes C and C++ perfectly suitable for the concepts we propose in this work. In general, the external programs are packed into shared libraries (e.g. jar, so, dll) and loaded into the database using a product-specific installation routine. The following list contains database products that support triggers and user-defined functions written in at least one of the languages C, C++, or Java and can thus be classified as Enhanced Active Database systems. The list is ordered according to their market share as presented in [43] and does not claim to be complete.

Oracle Database: The object-relational database *Oracle* runs on various platforms and provides a comprehensive set of tools for data management [7, 73]. As an active database it supports triggers on row, statement, schema, and database level. The basic version of Oracle was developed in 1979 and since then constantly enhanced [82]. Since version 6.0 it comprises PL/SQL, a procedural

language for database scripting, to provide a more complete programming solution for database applications. The first, although very limited possibility of interaction with its execution environment was introduced on version 7.3 with the `UTL_FILE` package allowing PL/SQL scripts to read and write external files sequentially. Access to operating system operations could be provided using the `DBMS_PIPE` package that allows a PL/SQL script to put a request in a database pipe from which it could be picked up and processed by a listener written in Perl or the Oracle Call Interface (OCI). In the current version 10g, Oracle supports stored procedures and user-defined functions written in C and Java that are callable directly from within triggers. Java Stored Procedures were introduced in version 8i which was released in 1999. It comprises its own J2SE 1.4.x compliant Java Virtual Machine as well as a couple of extensions to the JDBC connectivity framework like JDBC Thin and server-side internal drivers.

IBM DB2 UDB: The *IBM Universal Database 2 (DB2)* was originally released in 1983 and is now available in the version 8.2 [4, 11, 59]. Like Oracle it runs on various operating system platforms and has active capabilities in the form of triggers, although only supported on row and statement level. In the current version, stored procedures and functions can be written as SQL stored procedures based on procedural extensions to the SQL language (similar to Oracle PL/SQL), or based on high-level languages on the host system, such as RPG (Report Program Generator), COBOL, C, or Java. Non SQL procedures like C and COBOL were introduced with the DB2 version 6 released in 1999, whereas Java was not supported prior to version 7 from 2001. COBOL is a high-level language suitable for data processing in business applications. Initially designed for the handling of huge amounts of data stored in a specific record format, COBOL is also able to access databases directly via specific COBOL database bridges like [28].

Microsoft SQL Server: The *MS SQL Server* is currently in the version 2005 and like Oracle and DB2 supports triggers and external procedures and functions [76, 106]. The first version of the SQL Server was developed for the OS/2 platform by Sybase and released in 1988. In 1994 Microsoft ended the marketing partnership with Sybase and bought a copy of the source code to independently develop their own database server designed for Windows NT. After the release of the first SQL Server version 4.3, the versions 6.0 (in 1995), 6.5 (in 1996), 7 (in 1998), and 2000 followed. The database uses its own SQL dialect Transact-SQL to manipulate the data. Stored procedures can either be a collection of Transact-SQL statements or a reference to a Microsoft .NET Framework common language runtime (CLR) method. The CLR component was integrated into SQL Server 2005 and allows the execution of stored procedures, triggers, or functions written in a compatible .NET programming language like Visual Basic .NET or Visual C#.

Connections to remote databases are established using the ADO.NET framework based on the ActiveX Data Object (ADO) technology. Furthermore, Visual Basic and C# support remote procedure calls. In the previous versions (since 6.5) SQL Server supported extended stored procedures to load and execute a function within a dynamic-link library (DLL). The development of such extended stored procedures is treated as any other DLL development. DLLs are shared object written in C or C# that can be accessed by multiple threads at the same time. They can be called from within trigger like common Transact-SQL statements using the data connectivity of the host language to connect to remote data sources.

Informix Dynamic Server: The *Informix Dynamic Server (IDS)* is the database system of the Informix company which was taken over by IBM in 2001 [60]. The current version 10.0 runs on various platforms and is designed for online transactional processing (OLTP) applications. Like Sybase ASE, IDS descends from the Ingres relational database developed at the University of California, Berkeley. The database supports triggers on row and statement level. External functions and stored procedures in the languages C and Java is supported by the database since its version 9.2 released in 1999. C programs are loaded into the database as DLL or shared libraries depending on the operating system. The support of Java requires the J/Foundation extensions which contains the JVM of Sun. The mechanisms and syntax to load, register, and execute external functions and procedures is similar to DB2.

Sybase ASE: *Sybase Adaptive Server Enterprise (ASE)* 15.0 is the current version of the Sybase relational database which was first released in 1984 as *Sybase SQL Server* and has its name since version 11.5, released in 1997 [110, 111]. Like Informix, the database is a descendant of the Ingres database and was developed by Sybase in cooperation with Microsoft until 1994. After their marketing partnership ended in 1994, both databases were further developed independently. Due to their common history, the products share many basic foundations, particularly the SQL dialect Transact-SQL, although now in slightly different versions. The database runs on various platforms and implements active capabilities in the form of triggers. Since version 12.0, which was released in 1999, Sybase ASE supports external stored procedures that are registered in the database as common procedures but implemented by an Open Server application called XP Server. The procedural functions written in C or a language capable of calling C functions are loaded into the database as shared libraries like in Oracle, DB2, or Informix Dynamic Server. Also since version 12.0 the database comprises an internal Java Virtual Machine to execute Java methods as functions and stored procedures.

PostgreSQL: *PostgreSQL* was initially developed, again, as a successor of the Ingres database at the University of California at Berkeley [96]. In 1996 it started as an open source project and was soon replaced by a radically transformed and enhanced version known under its current name PostgreSQL. It is currently in the version 8.1 and supports per-row and per-statement triggers as well as external stored procedures and functions. They can be written in procedural languages like PL/pgSQL, PL/Tcl, PL/Perl, PL/Python, as well as C and, since version 8.0, also Java. The trigger definition in PostgreSQL strongly differs from other DBMSs. Trigger events are specified in SQL but the actual trigger action is implemented as an external trigger function, one for each trigger. The trigger is executed by a trigger manager that passes arguments to the trigger via specific trigger data structures.

This list shows that most database vendors already support external program calls and implies that the technology will be most likely included into most database products in the near future. In the following chapters we show how the enhanced functionality of Enhanced Active Databases could contribute to common problems in federated information systems.

Chapter 4

Active Event Notification

The integration of data sources into federated information systems is a difficult task, especially when they shall be loosely coupled to the system and retain the highest possible degree of autonomy. According to the basic architecture described in Section 2.2, a database is generally integrated using a wrapper component which encapsulates a source and provides a common interface to the federation layer. A difficult problem in such an environment with autonomous component databases is the detection of events in the integrated data sources in order to react to that event on the global level. Particularly in event-based systems of event producers and consumers, we need a mechanism to detect events in the attached sources and propagate those changes to corresponding event processing components. A special application scenario is real-time (or zero latency [81]) data warehousing, where updates are propagated and integrated into the warehouse immediately after the update occurred. The implementation of such types of data warehouses demands for real-time event delivery mechanisms for the integrated sources.

A common approach to event detection in databases is monitoring. The source is scanned at specific intervals to poll events using change extraction algorithms based on, for example, snapshots or log files. Although this method is widely used in practice, it is only able to provide periodic or deferred updates to an information system, since the Event Monitor does not exactly *know* the time an updated occurred. A truly immediate update notification requires the notification process to be integrated into the update process at the data source itself. This requires firstly that the database is able to detect and react on local events, which is the case for active databases, and secondly that the database is able to actively notify an external component about the local update. The latter functionality is provided by the external program calls, which can use client-server APIs of the external programming language to open the required connections.

In this chapter we present a concept that allows Enhanced Active Databases to actively notify an external notification interface about updates in its local data stock. The concept fully exploits the enhanced activity of the database to pro-

vide an information system with immediate update notifications. We describe the interaction of the database with a Notification Interface, which is specifically designed to support *Active Event Notifications* invoked by Enhanced Active Databases. The concept is particularly suitable for real-time data warehousing and global integrity maintenance supporting both, asynchronous and synchronous event delivery to a monitoring component. Active Event Notifications build the basis for the concept of global integrity maintenance and the tightly coupled wrapper component introduced later in this thesis.

We start with a general overview of monitoring concepts including a description of event detection phases, concrete change capture methods, and additional change delivery options in Section 4.1. Section 4.2 summarizes related work presenting solutions to event detection in research projects and major commercial database products. Our concept of Active Event Notification for immediate synchronous and asynchronous update delivery is described in Section 4.3.

4.1 Monitoring Concepts

Before we describe our concept of Active Event Notification using Enhanced Active Databases, we shortly summarize popular monitoring techniques and properties, mainly developed in the context of data warehouses for incremental view maintenance. Updates are extracted from the sources (base relations) and sent to the data warehouse where they are incrementally integrated and stored. However, since a data warehouse can be considered as a federated information system with read-only operations on autonomous operational sources, the techniques are also applicable to other forms of federations, where the federation layer must be aware of local updates.

Which monitoring techniques are applicable to a data source strongly depends on the type and activity class of the data source. With the definition of Enhanced Active Database systems as an extended type of active databases we distinguish between three data source activity classes:

Passive Data Sources: Passive data sources are still widely used in practice and a lot of important data is stored in flat files (CSV) or spreadsheets. With the uprise of XML and the semantic web, the amount of semistructured information sources steadily grows enormously. Unstructured and semistructured information is typically stored in text files without being managed by a database management system (except XML databases like [61]). Thus, such flat files, spreadsheets, or XML files do not provide transaction management and integrity checks. This activity class also includes passive database systems which in fact comprise a database management system but do not support triggers to react on local events.

Active Databases (ADBS): Active databases comprise an integrated active

mechanism to react on local events and execute integrity checks on the local data to ensure consistency. They commonly support triggers based on ECA rules which can be set up to fire on a certain event (insert, update, or delete), evaluate a trigger condition, and determine subsequent trigger actions. With triggers we are able to implement constraint checks that involve more than one entity in the database.

Enhanced Active Databases (EADBS): Enhanced Active Databases as introduced in Chapter 3 are active databases with enhanced activity. The main difference is their ability to execute external routines written in external programming languages, which enables them to actively interact with external components in a complex way, like calling remote procedures or querying remote databases directly.

The active capabilities determine the capture methods and data delivery options that can be used to monitor the data source. Changes in data sources are typically captured using an event monitor that implements a concrete monitoring concept suitable for the underlaying data source activity class.

4.1.1 The Event Monitor

Figure 4.1 depicts an Event Monitor and its interaction with other architectural components in an event-based information system. the Event Monitor can be implemented in any component that has access to the data source and wants to stay informed about events in the underlaying data source. In a federated information system, the monitor is usually part of the wrapper or federation layer, depending on the concrete design and intended functionality of the system. If implemented in the wrappers, the changes are extracted from the sources directly by the wrappers and propagated to the federation layer. Otherwise, updates are extracted from the sources by the federation layer via the wrapper components.

An event monitor basically consists of a *Change Capturer* component and a clock to trigger periodic data extractions. The capturer knows how to access the data source and implements a specific capture method suitable for the data model and activity class of the underlaying source. The captured changes are propagated to an event processor where they could, for example, be integrated into an existing data stock (like a data warehouses), replicated to another data source, or checked against integrity constraints. In many real-time scenarios, the Event Processor is a messaging system that distributes updates in a publish-subscribe fashion. Besides invoked by the internal clock, the capture process can also be triggered by an external user or application or directly by the data source itself. To better understand the sequential execution of the event detection process using an Event Monitor, we distinguish between the following three phases:

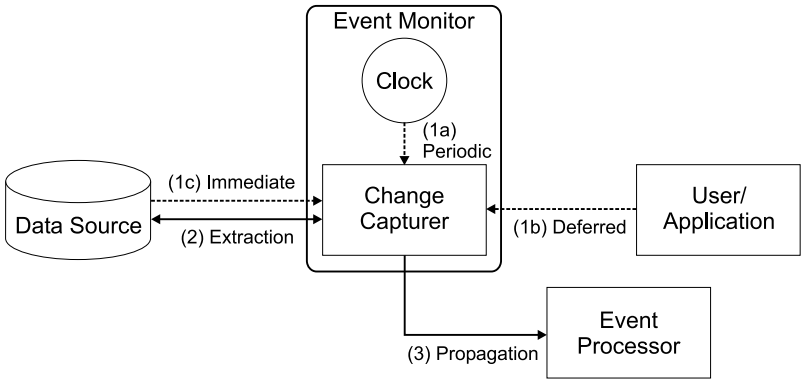

Figure 4.1: Interaction of the Event Monitor.

1. **Notification:** During the notification phase, the change capturer receives a notification that changes should be extracted from the data source. Thereupon, the capturer executes the subsequent change extraction phase. The invocation can either be triggered by a user or application, a clock, or the data source itself. The type of notification is determined by the data delivery schedule that is discussed in detail in Section 4.1.3.

2. **Change Extraction:** During the change extraction phase, the change capturer uses a specific capture method to identify and retrieve updates from the database. The concrete change capture technique depends on the data model and the activity class of the source, i.e. if the source is a passive, active, or enhanced active data sources. A list of popular change capture methods are presented in Section 4.1.2.

   There are several approaches for change extraction in different kinds of data sources available. For example, well-known solutions for passive relational or XML data sources are presented in [69] and [118] respectively, whereas updates are detected by comparing the updated data with a snapshot created previously. Other approaches use log files or timestamps to identify updates data items.

3. **Change Propagation:** After updates have been extracted from the source, they are sent to an event processor component for further processing. In a data warehouse scenario this component would most likely be a data integrator that loads the updates into the staging area of the data warehouse. In the context of other information system architectures this could be a constraint manager, a replication manager, or an event broker that processes the updates in an application dependent way.

Before we discuss the data delivery options including the delivery schedule

that determines the notification phase, we give an overview of commonly used approaches for the extraction of changes during the change extraction phase.

4.1.2 Change Capture Methods

There is a number of concepts for event detection in various types of data sources. They can basically be divided in static and incremental capture methods. While static capture is usually associated with taking snapshots of the data periodically, the incremental capture methods do consider only the updated data items and not the entire data stock. Both classes have different requirements towards the capabilities of the data sources and thus entail different advantages and disadvantages. In the context of federated information systems, we assume that monitor components of the federation layer monitor the source to detect and extract local updates.

Static Capture Methods

Static capture methods are intended for monitoring sources that store a manageable amount of data. They are less suitable for sources with high data load, since the data capture process could significantly influence their performance, especially if they store large data sets. However, except for the file comparison capture, the techniques are rather simple and therewith easy to implement.

Static Capture: The idea of this simple monitoring technique is to periodically take a snapshot of the entire base relation in the source and load it into the data warehouse. The information in the warehouse can either be replaced by the snapshot or the data can be appended to an existing table. Thus, the warehouse either holds a replica of the base relation at a given time or an accumulation of all data items over a certain time period. Unless the federation layer needs to maintain copies of the data as required for data warehousing, this capture method is rather inappropriate. In general, the federation layer needs to be provided with the updated records only rather than the entire data stock. These changes can be computed using a snapshot copy maintained by the federation layer (see file comparison capture below).

Timestamp Capture: The timestamp capture method assumes that every record contains information about the time at which it was last updated. These temporal information can now be used to select updates from the base relations. The monitors select only those records which have changed since the last scan. This capture method is independent of the database type but does not capture all changes of state of the records that occur in the time period between two scans. Furthermore, deleted records are not considered unless they are marked as deleted in the base relation and

purged from the database after the capturing process. Thus, the timestamp capture is not applicable in federated information system, if the federation layer must be aware of all changes of state (as required for global integrity maintenance) or if the sources do not implement a *marked-as-deleted* status for deleted records.

File Comparison Capture: The file comparison capture (also known as snapshot differential method [69]) detects updates in the base relation by comparing it with a previously taken snapshot. The records in the base relation are compared to the entries in the snapshot to reveal inserted tuples, that are not present in the snapshot, and deleted tuples that only exist in the snapshot. Records existing in both, the base relation and the snapshot, have to be scanned for changed attribute values to identify data updates. Typically, to speed up this comparison operation, the snapshot does only store the key attributes together with a hash value that is calculated from the attribute values of the record. If the hash value calculated from the base relation does not match the corresponding hash value in the snapshot then the record has changed and is identified as an update. The snapshot management and record comparison has to be implemented in the federation layer, whereas the snapshot should ideally be stored in an additional repository maintained by the federation layer to preserve design autonomy of the source. Popular representatives of file comparison methods are the comparison method for unstructured strings [57], relational and hierarchically structured data as presented in [69] and [23] respectively, or algorithms for the comparison of semi-structured data like XML files as described in [27, 118].

Incremental Capture Methods

Incremental capture methods provide the integration layer with updates without taking a look at the entire basis relation. Updates are stored in a persistent area (*delta sets* [46]) where they are captured by the monitor and deleted after processing. Thus, unlike using static capture methods, all changes of state of the records are recorded and accessible by the monitor. Incremental methods are closely tied to the capabilities of the source (DBMS) and more complex than static methods. They are particularly suitable for data sets where the amount of changes in a given update window is significantly smaller than the size of the entire base relation.

Application-assisted Capture: This incremental capture mode is implemented in the applications that modify the local data sources. Every time an update operation is executed by the application on the local database, it concurrently writes the changes in a persistent *delta set* (file, database table, etc.) for further processing. The updates can be polled from the storage

area immediately. The method inherits several problems: all applications that modify the data source have to be recoded to write the changes in delta sets. Furthermore, an application might only maintain key information of the records whereas additional information is added by the database (e.g. a timestamp or other default values). This requires the application to fetch the entire record from the source before writing it to the delta set. In the context of federated information systems, an implementation of the application-assisted capture method requires the recoding of all local applications that modify data on the component databases to maintain delta sets. These can then be retrieved periodically by a monitor in the federation layer to capture the data updates. Like the timestamp technique, this method is independent from the database type but is difficult to apply to legacy systems.

Transaction Log Capture: This capture method exploits the logging and recovery capabilities of database management systems to capture updates. The DBMSs maintain those log files for transaction management and system recovery so they particularly store all the information about write operations on the database. The files can now be monitored to periodically extract changes of interest without limiting source performance. The main drawbacks of this method are that the monitor must be able to identify and extract only that information that is already committed by the database and that the transaction logs must be available until all changes of interest have been captured. Obviously, this method is only applicable to source with DBMS and the log files must be accessible by the federation layer. This could cause security problems, since the log file access enables the federation layer to basically read all local transactions including those that should not be visible to the federation. The transaction log-based capture method is also the basis for popular database replication techniques, so replication-based monitoring approaches are basically represented by this method.

Trigger-based Capture: The last capture method we discuss is based on the active capabilities of active databases and uses triggers to maintain the delta sets which are typically stored directly in the database. A trigger is invoked by a certain condition or event and writes a copy of the changes of interest to the delta set. The monitor can now retrieve the updates from the delta sets similar to the application-assisted approach. Since the invocations of the triggers significantly affect the system performance, this method should only be applied to sources that are capable of processing the expected number of events. The use of triggers implies a significant limitation of design and execution autonomy. The database administrators must agree to set up the triggers and store delta sets in their database.

The main advantage is that the delta sets are maintained directly by the database independent of local and global applications and contain only changes that are already committed by the database.

The capture methods do only describe the requirements and algorithms to retrieve updated records from the sources. The interaction of the Event Monitor with the source and other event consuming components is defined by the delivery options described next.

4.1.3 Data Delivery Options

Besides the capture methods that define the techniques to identify and extract updated information from the sources, there are additional delivery options that must be considered during source monitoring. Data delivery options define the interaction of the Event Monitor with the data source and the Event Processor, and significantly influence the behavior of the entire information system. The option we discuss here are the *delivery schedule* that determines the notification phase and therewith the freshness of the system, the *delivery mode* that differentiates between push and pull-based data delivery, and the *coupling mode* of the notification process and the local write operation. The descriptions in the following subsections are mainly based on [38].

Coupling Mode

The first delivery option that significantly influences the behavior of the entire system is the coupling mode. It defines the way of interaction between a local transaction process that modifies data, and the change extraction process executed by the data capturer. The coupling between these two processes can either be asynchronous or synchronous:

Asynchronous: The change extraction process is completely detached from the update operation process. The update operation is not blocked while changes are extracted and transfered to the Event Processor. Changes must be committed to the source before the change extraction is started. Since the extraction is detached from the update, there is typically a latency between the time the changes are committed and the time they are extracted and propagated to the Event Processor.

Synchronous: The change capture process is executed as part of the update process. The DBMS blocks the local transaction during the change extraction and event processing phase. After that, locks on the local transactions are released again.

Event detection is commonly implemented using the asynchronous coupling mode. Changes are extracted after changes are written to the source and subsequently processed in an separate process. In general, events are propagated to a message queue from where they are processed chronologically by an event consumer. Synchronous event detection in a federated information system allows the federation layer to react on local events as part of the update operation in the source. This is essential in highly reliable systems with strict consistency. As we will show in Chapter 5, synchronous notification can be used for global constraint maintenance in federated information systems.

Delivery Schedule

Another delivery option we discuss here is the delivery schedule that is implemented by a monitor. It determines the time a notification is sent to the Event Monitor to extract updates from the source. A widely accepted classification for centralized and distributed systems is presented in [29, 127], where updates are extracted according to a *periodic*, *deferred*, or *immediate* delivery schedule, depending on who is triggering the change extraction phase. The three types of notification connections are depicted as dashed arrow lines in Fig. 4.1:

Periodic Delivery (1a): The probably most common delivery schedule is the periodic update extraction. The source is scanned for changes at certain time intervals, commonly triggered by an internal clock of the Event Monitor. The polling rate is significantly affecting data freshness and performance within the entire system. Although a scan should ideally only be performed if changes occurred in the source, the optimal polling strategy depends on the application field. For example, in a data warehouse scenario it could be sufficient to load new data from the sources once a day, whereas a more time-critical scenario might require updates of the view every hour or even every minute. However, too many scans in a short period of time unnecessarily stress the underlaying data source, especially when only a few changes occurred in the same time. On the other hand, if there are too few scans on a source with frequent updates, the data in the system will always be out of date. Periodic updates can, for example, be optimized using the Slacker coherence protocol presented in [109], which dynamically adjusts the polling rate to the update frequency observed at the source. Furthermore, periodic update extraction can be scheduled into off-peak hours (e.g. during the night), not to interfere with local applications.

Deferred Delivery (1b): A deferred delivery schedule captures updates from the sources as needed, for example initiated by a user or an application, whereupon the change capturer executes the change extraction phase. For instance, deferred delivery can be used to refresh a materialized view in a data warehouse whenever a query is issued against that view. The system

will refresh the view by importing updates from the sources before results are sent to the user. Deferred delivery will always provide fresh data, but entails a couple of drawbacks. Since updates are extracted on demand, it takes some time to calculate all updates and transfer them to the Event Processor. Furthermore, if an application triggers the extraction during time with high load on the sources, this can significantly slow down the systems, especially if a static capture method is applied.

Immediate (Real Time) Delivery (1c): The immediate delivery schedule requires updates to be extracted *directly* after the changes occurred in the data source. The source notifies the change capturer which thereupon starts the change extraction process. Immediate delivery produces significant additional workload to a data source, since each local transaction entails the change delivery. However, immediate updates are crucial for real-time applications like real-time data warehousing or the maintenance of strict consistency and data replication in federations. We like to point out that although an immediate schedule can be approximated by a periodic (near real time) schedule using a time interval small enough to report changes almost instantly (e.g. every second), it cannot substitute a real time (immediate) schedule. A *truly immediate* schedule will adapt automatically to the (aperiodic) update frequency of the source and report changes only when they occur, whereas a near real time periodic schedule will stress the data source due to high frequent scans even if the source was not modified.

With regard to the coupling mode we can state that a synchronous event monitoring mechanism will always implement an immediate delivery schedule, since the capture process is part of the update operation and therefore executed at the time the update occurs. Although discussed and required as a fundamental functionality of various research projects and applications, a truly immediate update notification mechanism has not yet been described in the literature (see Section 4.2 for details). Almost all the projects and applications that claim to perform real-time change capture operate in batch mode assuming that the changes can be polled from delta sets, log files, or event queues on a certain schedule (periodic delivery) [70]. So one of our contributions as presented in this chapter is a concept for truly immediate data delivery in federated information systems.

Immediate and deferred delivery are synonymously referred to as *aperiodic* delivery [38]. As we will see in Section 4.3, Enhanced Active Database systems are able to actively notify an event monitor about changes in their data, so that inefficient empty scans, i.e. scans on an unchanged data stock, are avoided. Even more, a scan is performed in the second the source was updated providing an event processor with up-to-date information.

Delivery Mode

Another important delivery option of monitoring techniques is the delivery mode. Changes can either be *pulled* from the source from an event processor or actively *pushed* by the source. Whereas polling the changes from the sources is a commonly used technique in almost every monitoring implementation, truly pushing mechanisms for immediate updates initiated by the data source are currently not available (see 4.2).

Pull: This is the traditional request/response mechanism over a unicast connection. The data capturer receives a notification from a component to initiate the change extraction phase and subsequently pulls changed records from the source. In case of a static capture method, it will compute the changes, for example, using snapshots or timestamps directly from the base relation. Using an incremental capture method, changes are polled from a persistent area, such as delta sets, log files, or other kinds of message pipes implementing the producer-consumer pattern. The source acts as a passive data provider and responses to requests from external components. It does not actively participate in the event detection process. Data can be pulled from the sources periodically (*periodic pull*) or aperiodically (*aperiodic pull*).

Push: This data delivery mode is typical for publish/subscribe systems, where the data flow is initiated by the data source. Interested components subscribe to specific information and receive an update automatically whenever changes occur in the data source on that portion of information. Updates can be pushed to subscribers using different delivery schedules (*periodic push* and *aperiodic push*). Update messages are actively created and transmitted by the data source. In particular, we do not consider a monitor component running at the database that extracts changes from the source and sends them to an event broker as a truly push-based delivery mechanism, since changes are again polled by an external component.

The data delivery options determine the behavior of the entire event-based system. Like the change capture method, the supported delivery options also depend on the capabilities of the underlaying data source. Table 4.1 lists the delivery options and capture methods supported by different data source activity classes. As can be seen, there are three delivery options that are uniquely supported by EADBSs (marked with $\oplus$). As the only data source activity class, EADBSs are able to provide immediate (real time) update delivery with pull and push using asynchronous and synchronous coupling modes.

	Passive Sources		ADBS	EADBS
	without DBMS	with DBMS		
Change Capture Method				
Static	+	+	+	+
Timestamp	+	+	+	+
File Comparison	+	+	+	+
Appl.-assisted	+	+	+	+
Transaction log	-	+	+	+
Trigger-based	-	-	+	+
Delivery Schedule				
periodic	+	+	+	+
deferred	+	+	+	+
immediate	-	-	-	⊕
Delivery Mode				
pull	+	+	+	+
push	-	-	-	⊕
Coupling Mode				
asynchronous	+	+	+	+
synchronous	-	-	-	⊕

Table 4.1: Data source activity classes and their supported monitoring options

4.2 Related Work

4.2.1 Research Projects

Most of the work concerning the monitoring of sources deal with the incremental view maintenance problem for single and multiple view using incremental and parallel processing of updates in a data warehouse environment [71, 126, 127]. A common characteristic of all these approaches is the computation of updated views within the data warehouse using the sets of changes provided by distributed sources. Updates are extracted from the sources using wrapper and monitor components, but only few details are available on the monitoring process and the extraction of updates itself. [126] presents a data warehouse architecture with wrappers for various distributed data sources supporting only deferred updates initiated by the data warehouse manager. There are no details given about the particular change detection process within the wrapper components. In [71] the authors describe an experiment to evaluate their view maintenance algorithm using an Oracle 8i database, but again there are no details about the implementation of the change detection provided. The sample scenario used in [127] has also no information about the functionality of its event monitors. The source simply *sends updates to the data warehouse*.

A rather related concept is described in [121] where wrappers and monitors

are realized as separate modules within the Whips architecture. While the wrappers are responsible for the conversion of queries and query results, the monitors extract updates from the heterogeneous sources. The monitors should also take advantage of sources that are willing to support the event detection process. Besides periodic updates using snapshots and database logs, they claim to have implemented trigger-based monitors for relational sources, but there are no further details available. In particular, the system does not support immediate updates.

Another related approach for distributed events in a heterogeneous environment is presented in [66, 117]. CORBA-based, distributed, and heterogeneous systems are enhanced by Active DBMS-style active functionality. The architecture uses wrappers with event monitors to detect data modifications in the data source based on polling and triggers. The trigger-based mechanism uses a pipe concept provided by the database to send updates as database messages to a local message buffer where they can be accessed by the wrapper. Thus, the wrapper has to poll events from the message buffer instead of the database itself.

The approach presented in [67] uses a database gateway solution to detect events in the local database. The authors state that triggers show too many restrictions and impose a substantial loss of autonomy. Thus, they propose a database wrapping approach that analyzes SQL statements to identify operations that modify the database. The gateway is created on top of the DBMS between the database and the application and eavesdrop the entire communication. However, this approach requires the DBMS to provide an interface where a gateway can be implemented to monitor the communication. Otherwise, especially all the local applications must be adjusted to use the gateway instead of the database server directly.

4.2.2 Commercial Change Capture Products

Change detection is an essential task in the popular application field of Data Warehousing. Since the major database vendors offer data warehouse add-ons to their database flagships, they provide specific solutions for event monitoring implemented as part of their ETL tools.

Since version 9i the Oracle database includes the *Change Data Capture* (CDC) framework to provide data warehouses with updates [83]. The CDC architecture uses rules and trigger to identify data that has been changed since the last extraction. The capture process works according to the publisher-subscriber paradigm that typically consists of one publisher and many subscribers. A trigger copies updated data to a specially created change table where they can be accessed by subscribers using individual views to access the information it is interested in and allowed to read. This trigger-based delta set approach is similar to the management of materialized views and is intended for asynchronous periodical pull of changes.

Oracle also offers *Oracle Streams* intended to continuously capture changes at a database and send them to remote sources as required for data warehouses or replication. The process is asynchronous using a source and a destination queue. Changes are captured at the source according to user-defined rules from the transaction logs, converted into the logical change record (LCR) information model, and enqueued in the source queue. Like in the CDC framework changes are polled from the queue using a propagation process that asynchronously polls LCRs from the source queue at a certain time interval. If immediate change delivery is required the interval must be set as small as possible resulting in queue lookups even if the queue is empty. Both concepts, CDC and streams, are unable to provide truly immediate change delivery or synchronous message processing.

In the IBM DB2 Universal Database the event detection basically works similar to the concepts implemented in Oracle. The database provides *SQL replication*, a delta set approach where changes are stored in change tables (cp. Oracle CDC), and *Q replication/Event Publishing* where changes are queue in a message queue before sending them to subscribers (cp. Oracle Streams) [58]. When using SQL Replication, a Capture program identifies changed source data using the DB2 recovery log files and saves the committed changes into staging tables. Sources in SQL Replication can be DB2 tables and views, or tables on non-DB2 relational databases. In the latter case, changes are captured using triggers (if provided by the source system). Corresponding to the Capture program on the source, there exists an Apply program on the target that supports three methods for replicating a table: full-refresh replication and two types of change capture replication. The full-refresh replication is a static capture method that frequently replaces the target tables with a snapshot of the source tables. The remaining two options captures a row (1) whenever a value changes in a column that is registered for replication, or (2) whenever a value in any column of the table changes. As already mentioned, changes in non-DB2 relational databases are captured using a trigger-based delta set capture method. Triggers are created on the source table to populate staging tables. For the delivery schedule the user can choose between interval timing (periodic), continuous (immediate), and event timing (deferred) whereas the capture method is asynchronous and pull-based, since changes are polled from the staging tables by the Apply program. Thus, applying continuous replication the Apply program replicates data "as frequently as it is able, depending on its workload and available resources" [58].

Q Replication also uses a Q Capture Program and a Q Apply program to capture changes on the source tables and to apply the changes on the target tables respectively. Contrary to the SQL replication, the programs directly communicate using messages sent over WebSphere MQ queues according to the publish-subscribe paradigm. The Capture program identifies committed changes using the transaction logs. They are converted into messages for the WebSphere MQ system and placed in the queue. The Q Apply program accesses those changes

from the queue and applies them on the target tables. Q Replication supports single-master (unidirectional) and multi-master replication between two tables (bidirectional) or multiple tables (Peer-to-peer replication). Q Replication is a DB2-specific replication technique and does not support non-DB2 databases. Event Publishing is an extension to Q Replication insofar as changes are converted to an XML representation format before they are placed in the message queue. There they can be accessed by user applications that are capable of processing XML documents. Event Publishing also uses the Q Capture program to detect updates based on log files. Again, like any other pipe mechanism, the publish-subscribe paradigm means asynchronous pull-based message transfer between the publisher and the subscriber.

To summarize we state that the capture methods provided by commercial database vendors are in general very source-specific solutions. They either are only applicable for their own database product or require additional software to execute. Data Warehousing requires the database companies to reluctantly provide methods and techniques to access databases of competing vendors, thus dealing with heterogeneity in their systems. The event detection concepts offered use asynchronous, pull-based publish-subscribe messaging systems that cannot provide truly immediate update notifications.

4.3 Active Event Notification

As we have shown in the previous section, EADBSs are able to provide truly immediate (real time) event delivery to an event processor without the need to stress the source with high frequent periodical scans. We now present the concepts of Active Event Notification for immediate push and pull-based change delivery with synchronous and asynchronous coupling mode. In all the presented concepts the communication between the EADBS and the Event Monitor is established via external programs and remote procedure calls. A trigger detects a local update event and executes an External Notification Program (see Section 3.2) that signals the event to the monitor. Since the most common events in databases are modifications of data, i.e. insertion, updates, and deletions, we limit our considerations to these operations. The ENPs connect to a specifically designed *Notification Interface (NI)* that is, in this context, implemented as a part of the Change Capture component. Please note that a Notification Interface can basically reside in any component of the federated information system that wants to be informed about updates.

Our description shall be clarified using a simple example. Consider an enhanced active relational data source DB that stores a base relation R with schema $\mathcal{R} = \{A, B, C\}$. An event monitor with a Notification Interface is set up in a wrapper component to receive active notifications from DB. Although we cast the following considerations on the relational model, our concept certainly adapts

to EADBSs with other data models by adjusting the change capture method. Furthermore, since Java is commonly supported by recent database systems, we describe our approach using Java implementations of the ENP and the Notification Interface. The concepts have been evaluated in the context of our tightly coupled wrapper architecture which is described in detail in Chapter 6.

4.3.1 Pull-based Asynchronous Notification

The first concept we present is the pull-based asynchronous notification suitable for most event-based information systems that require real time event delivery without blocking local transactions. The change extraction is initiated asynchronously *after* the corresponding local transaction is committed.

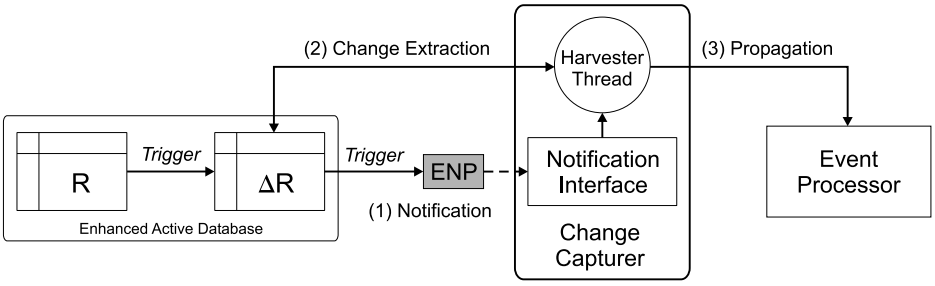

Figure 4.2: Pull-based asynchronous Active Event Notification.

Our approach fully exploits the active capabilities of the underlaying Enhanced Active Database. We use a trigger-based capture method that maintains a delta set in a separate relation which directly contains the updated tuples. Figure 4.2 depicts a schematic overview of the component interaction referring to our example. After a local operation modified the base relation R, a trigger copies the affected tuples to the delta set, where they are enriched with additional information. Subsequently, a trigger on ΔR executes the ENP to notify the Change Capturer about updates in R (notification phase). Thereupon it starts a *Harvester Thread* which continuously polls the tuples from ΔR until it is empty (extraction phase). Finally, the updates are converted into a common information model and sent to an event processor for further processing (propagation phase).

Creation of ΔR

For the base relation R in our example, we create a delta set ΔR with the attributes A, B, C, OP, T, with A, B, C being the original attributes of R. OP and T are additional attributes storing the type of operation and the time the operation occurred respectively. T is automatically set by the database using

a default value expression during the creation of ΔR. The value of attribute OP codes the type of data modification, like for example `I` for an insert, `U` for an update, or `D` for a delete. By default we set OP to $NULL$ to differentiate between newly inserted tuples and tuples in the delta set that still have to be processed by the Harvester Thread. The current operation type is set by the trigger that we use to maintain the delta set according to the corresponding operation on R.

Creation of the triggers

Changes to the base relation R are detected and processed using the active capabilities of the database. The following trigger example in SQL monitors insert operations on R and copies inserted tuples to the delta set ΔR:

```
CREATE TRIGGER R_insert
AFTER INSERT ON R
REFERENCING NEW_TABLE AS N
FOR EACH STATEMENT BEGIN ATOMIC
  INSERT INTO R_delta (A,B,C)
    SELECT * FROM N;
  UPDATE R_delta set OP='I'
    WHERE OP IS NULL;
END
```

In the first step, we copy new tuples from the temporary relation `NEW_TABLE` to ΔR and in the second step, since we have detected an insertion, we update ΔR and set OP to `I` where OP is not already set, i.e. only tuples that were added in the previous statement within the trigger are marked as inserts. Thus, for any number of affected tuples in R, we execute exactly one insert and one update statement inside the trigger. Please note that updates and deletions of tuples in R will also result in an insert operation in ΔR. Thus, ΔR reflects a chronological list of operations on R together with the operation type and the time the operation occurred. Now, we define a second trigger on the delta set `R_delta`, which exclusively fires on *update* events, as follows:

```
CREATE TRIGGER R_delta_notify
AFTER UPDATE ON R_delta
FOR EACH STATEMENT
  sendnotify('R');
END
```

The delta set `R_delta` shall, of course, exclusively be updated by the trigger `R_insert`. Thus, `R_delta_ notify` is executed as a cascading trigger directly after `R_insert`. It is triggered by the update statement in `R_insert` which sets the operation type in ΔR. This trigger calls the ENP `sendnotify` which is

registered in the database as a Java user defined function. The ENP connects to the Notification Interface of the wrapper and informs it about an update in relation R during the notification phase (see 4.1.1). Please note that each trigger is executed *once* for each `insert`, `update`, or `delete` statement on R. Thus, each update operation on R results in one call of the external notification program.

The External Notification Program

The external notification program `sendnotify` is used during the change notification phase to notify the Event Monitor about updates in the local database. A challenge here is the communication between the ENP and the Notification Interface of the Change Capturer (see Figure 4.2). During the implementation we face the problem that the Event Monitor in the wrapper component and the ENP are two different processes which are initiated and executed independently from each other. A wrapper component is installed and started once for each data source as a standalone application. An ENP is initialized and executed by the database system whenever a corresponding trigger is activated by a database transaction. After the notification, the external program process terminates and resources are freed again. In fact, we have one instance of the NI and many instances of the ENP. Thus, the communication between the ENP and the Notification Interface has to be established via sockets, a distributed object protocol such as CORBA, or, as described in this example, the Java Remote Method Invocation protocol. The NI registers a notification method `notify` to the local RMI registry. The ENP implements a call of this remote method and sends as a parameter the name of the updated relation `'R'`. The ENP is loaded into the database and mapped to a local function or stored procedure which can be called during the execution of a trigger (see 3.3 for details).

The Notification Interface

The NI is started by the wrapper component and listens for incoming RMI request from a databases via ENP calls. It is registered at the local RMI registry and offers a notification method `notify(relationname)` to get informed about updates of the database. The task of the `notify` method is to initialize the change extraction phase by waking up the Harvester Thread which is responsible for the relation `relationname`. If the harvester is already running due to a previous notification call, no action is performed by `notify`. The implementation of the NI determines the coupling mode of the event detection mechanisms. In an asynchronous implementation, as discussed in this section, the call of `notify` returns *immediately* after the harvester has been woken up, without waiting for a result. Thus, the ENP call returns and subsequently also the triggers `R_delta_notify` and `R_insert`, so locks are released and changes are committed before the harvester starts reading `R_delta`. Contrary, if the NI calls a method of another

component synchronously, the entire system gets blocked until the last function call has returned. For more details on synchronous notification we refer to Sections 4.3.2 and 4.3.4.

The Harvester Thread

The Harvester Thread (see Figure 4.2) is responsible for the extraction of updates from the database during the change extraction phase. It connects to the database and reads the updates directly from the delta set that corresponds to the notification call. Since tuples in ΔR are enriched with timestamps, the harvester is able to read the events that modified R chronologically as follows:

while runs
  while no entries exist in ΔR
    wait for notification from NI;
  end
  read all entries in ΔR ordered by T ascending;
  process entries chronologically;
  set $T_{max} = T$ of last update in current result set;
  delete entries in ΔR with $T \leq T_{max}$;
end

The monitor uses a single Harvester Thread for each delta set, which is executed as long as the wrapper is running or the Event Monitor is shut down. While the delta set is empty, the harvester waits for a notification from the NI. When it gets notified, it wakes up to read and process the tuples from ΔR chronologically. During the extraction of updates in the delta set, new updates can concurrently be executed on R and thus new tuples are added to ΔR. In that case, ΔR consists of a set of tuples ΔR^p, which are currently processed by the harvester, and a set of newly added tuples ΔR^n. Thus, the harvester deletes the set ΔR^p from ΔR after each read operation using the time of the last operation in ΔR^p as the split point. The harvester starts over again as long as unprocessed tuples ΔR^n exist in ΔR. Otherwise it goes to sleep until the next notification.

4.3.2 Push-based Synchronous Notification

Enhanced Active Databases are basically able to synchronously notify an external component about local updates. Synchronous notification is, for instance, required for strict global integrity maintenance in database federations or for synchronous replication. A local transaction is blocked until an external constraint manager is notified by the ENP and has evaluated a global constraint on interdependent data of remote component databases. Depending on the result of the global constraint check, the blocked transaction is committed or rejected. A

detailed description of the concept for global integrity maintenance in federated information systems using Enhanced Active Databases is given in Chapter 5.

We have evaluated an approach with a combined notification and extraction phase, where updates are transferred directly via the ENP as parameters. The main problem herein is the adaptability of the ENP to different schemas in the databases. The changed records must somehow be accessed and send by the trigger, no matter what schema the records have and which data types are used for the attributes. Since it is not reasonable to create a schema-specific ENP for each relation in the database, where each attribute in the relation matches a parameter of the ENP, we have to find a way to pass schema and data of an arbitrary relation to the ENP.

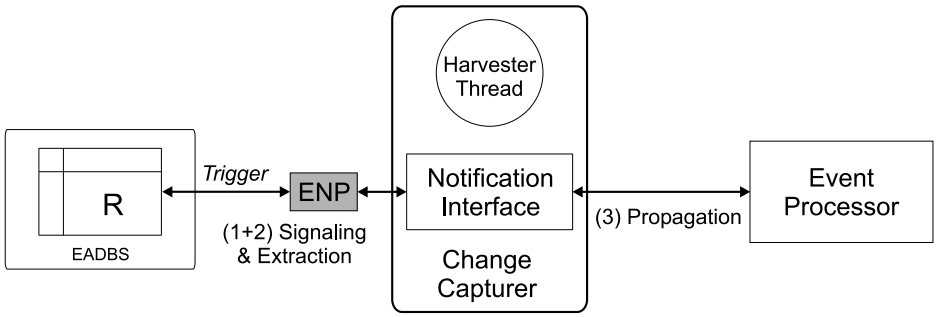

Figure 4.3: Push-based synchronous event notification.

Figure 4.3 depicts the general process of the synchronous push-based notification. Firstly, triggers must be defined for all relations that shall be monitored. A trigger is schema-specific and must be created individually for each relation. It fires on a local event and executes an ENP to notify the Notification Interface in the Change Capturer about the data modification. Contrary to the pull-based approach, this mechanism directly pushes the changes to the Notification Interface as part of the event notification via the ENP. From there they are synchronously propagated to the Event Processor. The coupling of the ENP, the Notification Interface, and the Event Processor is synchronous, i.e. the local transaction is blocked until the ENP returns from its call after the Event Processor and NI have finished processing (indicated by the bidirectional arrows). As mentioned above, the main problem arising in this concept is the handover of the changed records from the trigger to the ENP, which must be able to transfer the updates to the Notification Interface regardless of the specific structure of the data. Like in the asynchronous mechanism, the synchronous notification uses the delta sets of changed tuples in the temporary relations `NEW_TABLE` and `OLD_TABLE` storing new values (for inserted and updated records) and old values (for updated and deleted records) respectively. In general, due to the isolation property of database transactions, these temporary delta sets are not accessible from outside a trigger,

especially not from within an external program. Thus, the trigger has to loop through the delta set and calls an ENP for each inserted, updated, or deleted tuple passing their values as arguments to the ENP. Obviously, the arguments of the ENP must somehow match the attributes of the updated records. Therefore, we propose the following options to solve this problem.

Schema-specific ENPs

Records can be transfered to the Notification Interface using an individual ENP for each relation that is monitored (see Figure 4.4).

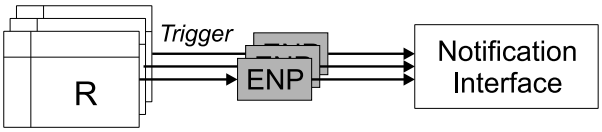

Figure 4.4: Notification via schema-specific ENPs.

For each attribute in the schema there is a corresponding argument, denoted as *schema argument*, in the signature of the ENP to take the values of records in R. Referring to the example of 4.3.1 the ENP for $R(A, B, C)$ takes four additional parameters:

```
notifyR ('R', operationtype, A, B, C)
```

`operationtype` marks the type of update operation (i.e. insertion, update, deletion), whereas the arguments `A`, `B`, and `C` take the values of the attributes A,B, and C respectively. Consider another relation $S(D, E, F, G)$ that shall be monitored, then we need an additional ENP specifically designed for the schema of S:

```
notifyS ('S', operationtype, D, E, F, G)
```

Obviously, `notifyR` cannot be used to process a record from S since the number of schema attributes does not match the number of attributes in S. All ENPs communicate with a single instance of the Notification Interface. Before the updates are sent to the NI, the records are converted into an object-based internal representation format comprising all the information required to propagate updates to an event processor: the name of the affected relation, the type of event, and the schema and values of the changed records. The benefit of this approach is obviously its robustness, since an ENP can specifically be coded to meet the requirements regarding the corresponding schema. On the other hand, to monitor a relation we must create a new schema-specific ENP, load it into the database, and register it as an additional user defined function. Thus, each

relation requires, besides the trigger definitions, a corresponding UDF and ENP to be present in the database. Furthermore, if the schema of a monitored relation is altered, it consequently requires the UDF and ENP to be changed, making the concept less suitable for systems with frequent schema modifications.

Delimiter Approach

Instead of individual ENPs for each monitored relation, the delimiter approach uses only a single ENP for all updates (see Figure 4.5). The main advantage of this concept is its universality, since an ENP must be loaded and registered in the database only once.

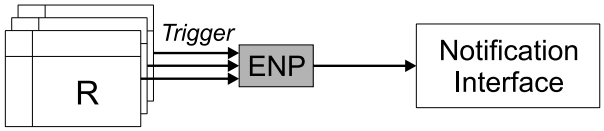

Figure 4.5: Active notification with a single ENP.

Due to the limited capabilities of the trigger language, we are unable to pass a sophisticated universal data structure to the ENP. The delimiter approach uses *host variables* `attributes` and `values` to store schema and data information to hand them over to the ENP as arguments. The variables are concatenated within the trigger using a unique sequence of characters as delimiter. For example, to propagate a tuple $r(1, 3, 4)$ of the relation $R(A, B, C)$, we create the host variables

```
attributes = "A#B#C";
values     = "1#3#4";
```

using the character `#` as delimiter. After R is updated, the trigger fires the ENP for each affected tuple passing the schema and the values of the tuple as arguments to the NI. To receive the schema and data information from the triggers, the ENP takes three additional parameters besides the name of the affected relation:

```
notify (relationname, operationtype, attributes, values)
```

The argument `operationtype` stores the operation type of the update whereas `attributes` and `values` match the concatenated host variables. The Notification Interface receives the data via the ENP and decomposes the attribute and value strings back into their atomic values for further processing and propagation. The following example shows an update trigger definition for R with host variable concatenation in DB2 syntax:

```
CREATE TRIGGER R_upd
AFTER UPDATE ON R
REFERENCING NEW AS n
FOR EACH ROW mode db2sql
BEGIN ATOMIC
  DECLARE attributes VARCHAR(32672)
  DECLARE values VARCHAR(32672)
  SET attributes = 'A#B#C';
  SET values = n.A||'#'||n.B||'#'||n.C;
  EXEC( notify('R', 'U', attributes, values) );
end
```

The delimiter approach works fine under several assumptions;

- updates may only affect a small number of tuples, since we experienced that the transmission of too many tuples caused a transaction rollback due to a trigger timeout;
- the length of the concatenated string of a tuple may not exceed the maximum length of the string host variable;
- the delimiter must be universally unique for any entry in the database.

Due to these assumptions, the delimiter approach does not seem to be applicable to relations with many attributes or attributes storing large values or binary data. Furthermore, if multiple tuples are affected by a single update statement and one of the tuples causes the transaction to be rejected due to local constraint, we have to execute compensating actions, since all previous updates have already been sent to the Notification Interface. However, it seems to be quite reasonable for global integrity maintenance, where an attribute check does only require the transmission of only a few attribute values. Chapter 5 presents such a concept using synchronous push-based notifications for global integrity checks. Both concepts for push-based event notification have their limitations concerning their applicability resulting mainly from limitations of the trigger language. An optimal solution would require the ENP to directly access the temporary delta sets `NEW_TABLE` and `OLD_TABLE` as part of a local transaction. This could be realized using references on the delta sets that are passed to the ENP as arguments.

4.3.3 Push-based Asynchronous Notification

The push-based notification mechanism can easily be changed to implement the asynchronous coupling mode. Therefore, the Change Capturer implements an event queue in which the changes pushed by the data source are placed chronologically. The call of the `notify` method of the NI is returned immediately after

an event has been queued, so the locks are released directly after all changes are propagated to the NI. Figure 4.6 displays a schematic overview of the process. Synchronous messages are indicated by bidirectional arrows, asynchronous calls by unidirectional arrows. The changes can either be polled from the event queue or pushed to the Event Processor (as depicted) independently from further notifications.

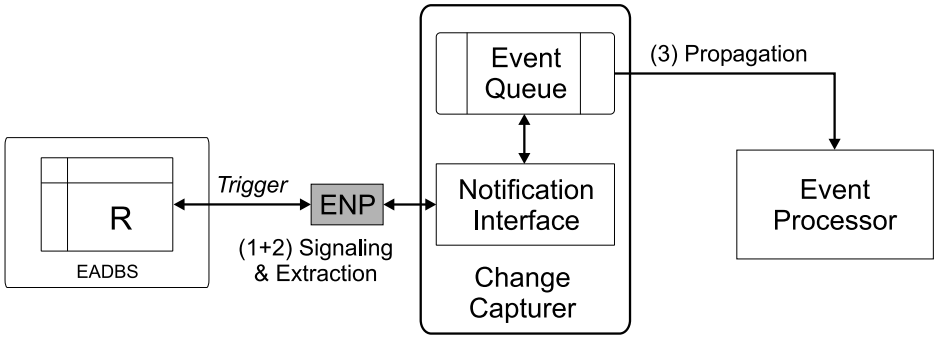

Figure 4.6: Push-based asynchronous event notification.

4.3.4 Pull-based Synchronous Notification

The adjustment to synchronously pull changes from the source in in no way straightforward. The main challenge in the synchronous notification lies in the change extraction phase. While the notification imposes no problem, the transmission of updates is rather difficult. For the evaluation of the delta set concept as presented in the previous section, we adjusted the Harvester Thread to return a boolean value when it finished extracting the updates from ΔR. The NI was recoded to wait for the return value of the harvester after its notification via the ENP. The result was a sequence of cascading calls of the triggers, the ENP, the NI, and the harvester, which resulted in a deadlock, since the harvester could not access ΔR because it was locked by the triggers. Even if the DBMS would allow read access to the delta set, the harvester would not be allowed to delete the processed tuples ΔR^p from ΔR. In general, the ACID properties of local transactions ensured by the DBMS prevent any external processes from accessing the changed records before they are committed. This basically applies to both, static and incremental change capture methods (except application-assisted), since the extraction of changes requires access to the entire updated data stock or materialized delta set. However, specific solutions for pull-based synchronous notification may be provided depending on the concrete implementation of concurrency control mechanisms of the databases. The Active Event Notification mechanisms have been evaluated as part of the tightly coupled wrapper architecture that is described in Chapter 6.

Chapter 5

Global Integrity Maintenance

The maintenance of global integrity constraints in federated information systems is still a challenge since traditional integrity constraint management techniques cannot be applied to such a distributed management of data. In this chapter we present a concept of global integrity maintenance by migrating the concepts of active database systems to a collection of Enhanced Active Databases. These *Active Component Systems* are able to interact with each other using direct connections established from within their database management systems to actively participate in global integrity maintenance. Global integrity constraints are decomposed into sets of *partial integrity constraints*, which are enforced directly by the affected Active Component Systems. The content of this chapter is based on the work originally published in [92].

We start with a definition of Active Component Database Systems based on Enhanced Active Databases in Section 5.1. In Section 5.2 we introduce partial integrity constraints as a new type of constraints to define dependencies on interrelated data managed by ACDBS. Based on partial constraints, Section 5.3 provides a detailed explanation of the constraint checking mechanism for a set of commonly used classes of constraints. A discussion of properties of our approach and related constraint checking protocols is covered in Section 5.4. To overcome some limitations of the basic concept, we present the COMICS constraint management extension in Section 5.5. Partial integrity checks are initiated by the Active Component Systems but executed and evaluated using an external constraint manager. The last section of this chapter provides an overview of related work.

5.1 Active Component Database Systems

Within the classical notion of federated databases [102], the component databases do only have passive functionality with regard to the federation. Like repositories, they provide access to their data and respond to data requests initiated

by the clients. Such passive component databases, with respect to the federation, operate isolated and do not have any knowledge of other CDBSs within the federation to which their data is related. Now, the enhanced functionality of Enhanced Active Database enables them to interact with external hardware or software components beyond their system borders. An EADBS that is integrated into a federated information system as a component database can use this enhanced activity to coordinate its actions with other components of the federation. In particular, the extended functionality can be used to ensure consistency of interdependent data and to enforce business rules in the form of global integrity constraints.

Definition 2 *An* Active Component Database System (ACDBS) *is an Enhanced Active Database which actively participates in maintaining global integrity constraints in a federated information system. It is able to directly communicate with other component databases, to which its data is semantically related, and implements constraint checks to maintain consistency among this interdependent data.*

Constraint checks are executed on ACDBSs using *remote state queries* from within triggers as proposed in 3.2. Local events are processed according to the *push-based synchronous notification* mechanisms proposed in 4.3.2. After a local event has been detected, the trigger executes a remote state query that uses the database connectivity of the programming language to connect to remote databases. Since we use synchronous notification, the local transaction is blocked while the EP is executed. After the connection is established, we execute queries upon the remote stock to determine the state of interrelated remote data items. The local transaction is aborted or committed depending on the state of the remote data sources. Such constraint checks performed by ACDBSs are triggered and executed on local CDBSs, but require access to remote data. Since these checks cannot be expressed by neither local nor global constraints, we introduce *partial integrity constraints* as a new type of integrity constraints for ACDBSs.

5.2 Partial Integrity Constraints

In this section, we present partial integrity constraints as a new type of constraints for ACDBS participating in a federated information system. Our work is developed in the context of relational databases, since this type of data source is widely used for data storage in practice and currently most EADBSs support the relational model (see 3.5). We assume an information system which comprises a collection of autonomous relational sources of various vendors running on different platforms. The databases store interdependent data that is accessed by local and global applications. Each ACDBS in this federation has to meet two

requirements concerning the programming language for encoding EPs for remote state queries:

- the DBMS must be able to connect to other component databases of the federation using the database connectivity of the programming language and
- the DBMS must support a query language understood by the other component databases to execute at least read operations on the remote data stock.

In practice, two widely used standard database connectivity interfaces are JDBC and ODBC, which support a multitude of relational databases. An established query language for relational databases certainly is SQL. This enhanced functionality is used to implement constraint checking algorithms for partial integrity constraints, which are defined next.

5.2.1 Definition of Partial Integrity Constraints

We start with the following definitions similar to [46]:

Definition 3 *A federation F of relational component databases is a set of n interconnected database systems $\{S_1, ..S_n\}$. The database systems do not necessarily have to be located on physically different nodes of the network. Each system $S_i \in F$ manages a local database D_i. A local schema $\mathcal{D}_i$ of a database D_i comprises the schemas $\mathcal{R}^i_1, ...\mathcal{R}^i_{n_i}$ of the relations $R^i_1, ..., R^i_{n_i}$ stored in the database. The global database schema $\mathcal{G}$ of F is the set of all relational schemas $\mathcal{R}^i_j$ in F.*

We assume that a real-world object, that is modeled in a component database of F, is globally identified by a set of key attributes, i.e. a real-world object will have the same key attribute values when stored in different CDBSs. Otherwise we assume mapping functions to match real-world objects in different sources.

Definition 4 *A* local integrity constraint $I_L^{D_i}$ *is a boolean function over a local database schema $\mathcal{D}_i$, i.e. $I_L^{D_i} : \mathcal{D}_i \rightarrow \{true, false\}$. A* global integrity constraint I_G *is a boolean function over the global schema $\mathcal{G}$, i.e. $I_G : \mathcal{G} \rightarrow \{true, false\}$. It cannot be expressed over a local database schema $\mathcal{D}_i \in \mathcal{G}$.* Constraint checks *for $I_L^{D_i}$ and I_G are algorithms for evaluating $I_L^{D_i}$ and I_G respectively.*

After a global constraint I_G has been defined over $\mathcal{G}$, we identify a non-empty set $\mathcal{C} \subseteq F$ of component databases $c \in \mathcal{C}$ whose local schemas $\mathcal{R}_c$ are affected by I_G, i.e. data stored in the relations R_c on the component databases is semantically related. Thus, from the point of view of each component database c, I_G affects a relation managed locally and at least one remote relation managed by another

component database. For example, if a key constraint is defined on a global attribute that is derived from multiple sources, each of the sources has to ensure the global uniqueness of the key attribute, when a new tuple is inserted locally. This means that I_G consists of a set of *partial integrity constraints*, which we define as follows:

Definition 5 *A* partial integrity constraint I_{R_c} *on an ACDBS* $c \in \mathcal{C}$ *is a boolean function, which is expressed over the local schema* $\mathcal{R}_c$ *and related schemas* $\mathcal{R}_{k_u}$ *for* $k_u \in \mathcal{C} \setminus \{c\}$*, i.e.* $I_{R_c} : \mathcal{R}_c \times \mathcal{R}_{k_1} \times \cdots \times \mathcal{R}_{k_v} \rightarrow \{true, false\}$ *for* $v \leq |\mathcal{C}| - 1$*. A constraint check for* I_{R_c} *is an algorithm for evaluating* I_{R_c}*, which is entirely implemented on* c *using external program calls to access the remote schemas* $\mathcal{R}_{k_u}$*.*

A partial integrity constraint consists of a local constraint check and one or more remote constraint checks on interrelated remote data, depending on the type of global constraint and the number of affected databases. It is used to express a global constraint from the local view of a single component database. An ACDBS, which implements a partial integrity constraint, has to ensure consistency of its local data depending on related data stored in other component databases. This means that it is responsible for checking a specific part of the corresponding global integrity constraint concerning local write operations on the interrelated data. Thus, a global integrity constraint is assured, iff all affected component databases enforce their corresponding partial integrity constraints, expressed as

$$I_G : \bigwedge_{c \in \mathcal{C}} I_{R_c}$$

The global constraint I_G consists of a conjunction of partial integrity constraints I_{R_c}, which are formulated as

$$I_{R_c} : local_{R_c} \wedge \bigwedge_{j \in \mathcal{C}'} remote_{R_c,R_j}$$

Depending on the type of global integrity constraint, each affected ACDBS c has to check an optional local condition $local_{R_c}$ and one or more remote conditions. $remote_{R_c,R_j}$ defines a pairwise dependency between the affected local relation R_c and one interrelated relation R_j on component j. Remote conditions do not necessarily have to be defined for each pair of local and remote databases in $\mathcal{C}$. Thus, $\mathcal{C}' \subseteq \mathcal{C} \setminus \{c\}$ denotes a subset of ACDBSs, which are required to check for a specific integrity constraint. A constraint check for I_{R_c} implements tests for the local condition $local_{R_c}$ and each remote condition $remote_{R_c,R_j}$. For aggregate constraints, a test for a local or remote condition results in the successful computation of a local or remote aggregate. Detailed examples for partial constraint checks are provided in Section 5.3.

5.2.2 Partial Integrity Constraints as ECA Rules

Since our concept of global integrity maintenance is based on ACDBS with enhanced activity, we use database rules following the event-condition-action (ECA) paradigm to specify a partial integrity constraint I_{R_c} on a component database c as follows:

define rule *PartialIntegrityRule* I_{R_c}
on *event which modifies* R_c
if *a test for* $local_{R_c}$ *yields false or*
a test for $remote_{R_c,R_j}$ *yields false*
do *local and/or remote action(s) to ensure or*
restore a consistent global state

Such integrity rules can precisely define both: events that potentially violate the integrity of local and remote data, and corresponding reactions on these events to ensure or restore consistency in the entire system. The relevant events concerning data consistency are modifications of the data stock, i.e. insertions, updates, and deletions. According to the definition of partial constraints, each rule condition and rule action of a partial integrity rule can consist of a local and one or more remote checks. The local check $local_{R_c}$ exclusively uses and accesses local data, while the remote checks $remote_{R_c,R_j}$ exclusively process data stored on remote systems. The remote checks of a partial integrity rule are implemented using remote state queries (or injected transactions) provided by the ACDBS. Thus, we have the following options to call an external program during an integrity check:

During the evaluation of a trigger condition: An EPC during the evaluation of a trigger condition allows a DBMS to determine subsequent actions depending on the result of a remote state query or the result of an injected transaction. Thus, a locally executed constraint check can be conditioned by the state and behavior of a remote data source. In our concept, most of the constraint checks are implemented using remote state queries from within trigger conditions. The part of the trigger condition, which evaluates a condition using remote data shall be called *remote condition*.

During the execution of (a) trigger action(s): Besides the remote condition, an external program can be executed as a trigger action. A local transaction can thus trigger an injected transaction to manipulate a remote data source. This can be used to execute consistency restoration actions or to implement special constraints like cascading referential integrity. This part of the trigger action, which manipulates remote data using injected transactions shall be called *remote action*.

The specific combination of the time the corresponding EP is executed (i.e. during remote condition or remote action) and the time a partial integrity rule is evaluated (i.e. **before** or **after** a local transaction is committed) significantly affects the behavior of the entire system. Please note, that we are certainly not limited to exclusively one of these combinations. During the evaluation of a partial integrity rule, external programs can be called from both, the remote condition and the remote action, before or after a local transaction is committed.

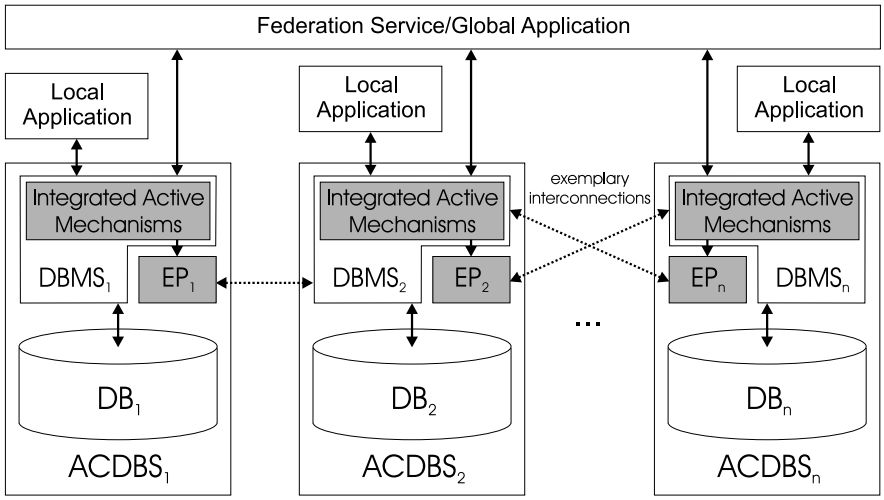

Figure 5.1: Global integrity maintenance with Active Component Systems

Putting it all together, we present the architecture for global integrity checking in federated information systems using active component databases depicted in Fig. 5.1. The partial integrity checks are defined and implemented directly in the ACDBSs, building up an application independent communication layer to jointly ensure global consistency of interdependent data. The ACDBSs uses push-based Active Event Notification in synchronous coupling mode to initiate and execute the constraint check process. Each transaction is checked according to local and partial integrity constraints, no matter if submitted by local or global applications. The maintenance of global integrity is thus migrated from a global application or federation layer to the underlaying active component databases.

5.2.3 System Interaction

We now give a schematic description of the interaction process between two ACDBSs during the execution of a partial integrity check. The concept relies on the push-based synchronous event notification mechanisms presented in Section 4.3.2. The synchronous execution blocks the local transaction until the partial constraint is evaluated using the remote data sources.

Consider two relations R and S on active component databases $ACDBS_1$ and $ACDBS_2$, which store interdependent data. To enforce a global constraint, we have to define and implement partial constraint checks on both component databases. Therefore we create the following objects on each ACDBS (see Fig. 5.2):

- An **external program (EP)** (here a Java method) to execute queries upon the remote data stock,
- a **user defined function (UDF)**, which is mapped to the external Java method, and
- a **trigger** which executes the UDF when relevant write operations occur on the relation.

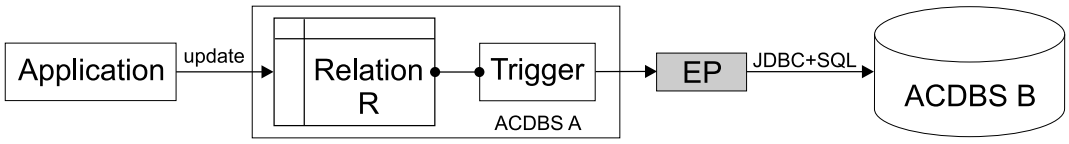

Figure 5.2: Interaction between two ACDBSs during a partial constraint check

We assume a global key constraint to be enforced on R and S, i.e. whenever a tuple is inserted or modified in R or S, we have to check the global uniqueness of the key attributes of the newly inserted or modified tuple in both relations. Our description focuses on data modifications in R, since modifications in S would be processed analogously. When an application inserts or updates data items ΔR in R, the corresponding trigger is executed by the database system *before* the transaction is completed. The trigger has access to ΔR via temporary tables provided by the DBMS. For each new tuple r in ΔR, the trigger first performs a check on the local data and afterwards, if necessary, on the remote data. If the local test fails, the key constraint is already violated and the remote test is omitted. If the local test succeeds, the trigger calls a UDF to check for conflicts in the remote data. The UDF is mapped to a Java function and receives r as a parameter from the trigger.

The Java function now bridges the gap between the two component databases. Using JDBC it connects to the remote database and executes an SQL query to check for the existence of r in S. The function returns $true$ or $false$ depending on the query result. The trigger receives this result and is now able to determine subsequent actions. Please keep in mind that we execute local and remote checks during the evaluation of the trigger condition before the transaction is completed. Thus, the transaction is blocked as long as the trigger is executed. If a corresponding tuple already exists in S, then we reject the data modifications on R. An SQL error is raised to signal the global key constraint violation.

Department	Database	Relations
A	DB_A	$resA(R_A, N_A, S_A)$ $projA(P_A, T_A, B_A)$ $proresA(R_A, P_A)$
B	DB_B	$resB(R_B, N_B, S_B)$ $projB(P_B, T_B, B_B)$ $proresB(R_B, P_B)$

Table 5.1: Example relations in the research departments A and B

5.3 Checking Global Integrity Constraints

We now concretize our concept of global integrity maintenance explaining how partial integrity constraints are expressed and implemented for different constraint types. Therefore we use a simplified scenario with two relational data sources. A generalization to more than two sites is discussed in Section 5.4.

Consider a company with two research departments A and B, which both manage their own autonomous relational database DB_A and DB_B. The company wants to integrate these standalone sources into an information system and define integrity constraints to ensure global data consistency within the entire company. We assume that the sources are relational databases with enhanced activity, which host the relations shown in Table 5.1. Each department stores information about researchers in relation *res* (researcher number R, name N, salary S) and their corresponding projects in relation *proj* (project number P, title T, budget B). Researchers are related to projects using the *prores* relation (m:n).

According to the classification presented in [115, 44], we consider four commonly used classes of global integrity constraints which can be defined on the global schema: attribute constraints, key constraints, referential integrity constraints, and aggregate constraints. In the following, we provide an example for each non-trivial class of global integrity constraints, followed by a rule definition for corresponding partial integrity constraint on the affected ACDBSs.

5.3.1 Attribute Constraints

The company could define a constraint saying that the budget of each project may not exceed a certain value. Since this global attribute constraint is expressed over a single attribute, it can be translated into a local attribute constraint and thus be enforced by local integrity mechanisms on DB_A and DB_B, e.g. by an additional **check** clause in both project relations. The global constraint can be enforced by a local constraint check on each ACDBS, so no EPCs are required.

5.3.2 Key Constraints

The company may want to ensure that each project is globally identified by a unique identifier, i.e. the values stored in P_A and P_B are globally unique. Thus, each time a project is added in one of the research departments, we have to ensure that the new project number does neither already exist locally nor in the project database of the other research department.

Since partial constraint checks are entirely implemented on a participating ACDBS using EPCs to access remote data, we decompose Key_G into a set of partial integrity constraints Key_{projA} and Key_{projB} for DB_A and DB_B respectively:

$$Key_G : Key_{projA} \wedge Key_{projB}$$

The partial constraints are in turn formulated as

$$Key_{projA} : local_{projA} \wedge remote_{projA,projB}$$
$$Key_{projB} : local_{projB} \wedge remote_{projB,projA}$$

A partial constraint consist of a local condition and a remote condition, which are defined for Key_{projA} as follows (Key_{projB} and $remote_{projB,projA}$ are defined analogously):

$$\begin{aligned} local_{projA} &: \forall P, T, B, P', T', B' : \\ &\quad [projA(P, T, B) \wedge projA(P', T', B') \Rightarrow \neg(P = P')] \\ remote_{projA,projB} &: \forall P, T, B, P', T', B' : \\ &\quad [projA(P, T, B) \wedge projB(P', T', B') \Rightarrow \neg(P = P')] \end{aligned}$$

Suppose the tuple $projA(p, t, b)$ is inserted. According to Key_{projA} we have to check the existence of the key locally and in $projB$, stored on DB_B, with the following tests for $local_{projA}$ and $remote_{projA,projB}$:

$$\begin{aligned} localtest_{Key_{projA}} &: \nexists P, T, B : [projA(P, T, B) \wedge (P = p)] \\ remotetest_{Key_{projA}} &: \nexists P, T, B : [projB(P, T, B) \wedge (P = p)] \end{aligned}$$

Both tests are evaluated by performing queries on the relevant relations for a tuple that has p as its project number. Therefore, we need two boolean functions

$$checklocalkey : schema(projA) \rightarrow \{true, false\}$$
$$checkremotekey : schema(projB) \rightarrow \{true, false\}$$

for $localtest_{Key_{projA}}$ and $remotetest_{Key_{projA}}$ respectively. *checklocalkey* should always be evaluated first to avoid cost-intensive remote data access where possible. The *checkremotekey* function is implemented using a remote state query, which queries database DB_B to find tuples with the values to be inserted. If the query result is not empty or the remote source is not reachable by the external program, then the function is evaluated to $false$, i.e. the corresponding transaction in database DB_A is rejected. The condition is evaluated *before* the triggering operation is committed at DB_A. The corresponding partial rule for Key_{projA} is expressed as:

define rule *PartialKeyConstraint*
on *creation of a new object in projA*
if *checklocalkey yields false* **or** *checkremotekey yields false*
do *reject transaction*

A partial constraint is herewith realized on an ACDBS with an implementation of the ECA rule using two functions *checklocalkey* and *checkremotekey* with one remote state query. Having implemented both partial constraints Key_{projA} and Key_{projB} on both systems, we are able to verify Key_G each time a modifying transaction is committed locally on DB_A and DB_B.

5.3.3 Referential Integrity Constraints

A widely spread constraint is the definition of referential integrity on relations to specify existence dependencies between two database objects. Referring to our scenario, the company could allow researchers of department A to cooperate on shared projects of department B. Thus, we have to ensure that a researcher in DB_A is related to an existing project in DB_B and vice versa. As already mentioned, researchers are related to projects via the *prores* relation referencing the relevant primary keys of the local project and researcher relations. Now, to reflect the global referential integrity constraint in our exemplary relational model, we allow R_B in $proresB$ to reference both, local researchers using R_B in res_B and cooperating researchers in res_A using R_A. In the scope of this thesis we only consider referential integrity without cascading, although our concept of ACDBSs basically supports cascading. An outlook on cascading referential integrity can be found later in this section.

Referential Integrity Without Cascading

In the following, we focus on the referential integrity concerning the researcher number R_B in $proresB$. Referential integrity for P_B in $proresB$ is handled analogously. Similar to global key constraints, a global referential constraint is first decomposed into a set of partial constraints:

$$RefInt_G : RefInt_{resA} \wedge RefInt_{proresB}$$

The existence dependency between the local parent relation $resA$ and the local dependent relation $proresA$ can be expressed as follows:

$$\begin{aligned} local_{proresA} : \forall R, P \exists R', N, S : \\ [proresA(R, P) \wedge (R = R') \Rightarrow resA(R', N, S)] \end{aligned}$$

Furthermore we formulate a remote constraint $remote_{proresB,resA}$ as

$$\begin{aligned} remote_{proresB,resA} : \forall R, P \exists R', N, S : \\ [proresB(R, P) \wedge (R = R') \Rightarrow resA(R', N, S) \vee resB(R', N, S)] \end{aligned}$$

Using these definitions, we express the partial constraints for $RefInt_G$ as

$$\begin{aligned} RefInt_{resA} : local_{proresA} \wedge remote_{proresB,resA} \\ RefInt_{proresB} : remote_{proresB,resA} \end{aligned}$$

A project can only be inserted into $proresB$, if a corresponding researcher exists in either $resB$ (locally) or $resA$ (remote). Contrary, a researcher in the parent relations $resA$ and $resB$ may not be deleted, as long as depending projects exist in the dependent relation $proresB$. Thus, we have to distinguish between constraint checks for insertions and deletions on the dependent and parent relations respectively.

Insertion check: Suppose the tuple $proresB(r, p)$ is inserted, whereas p refers to an existing project in $projB$. According to $RefInt_{proresB}$ we have to check the existence of r locally and remote using the following tests for $remote_{proresB,resA}$:

$$localtest_{RefInt_{proresB}} : \exists R, N, S : [resB(R, N, S) \wedge (R = r)]$$
$$remotetest_{RefInt_{proresB}} : \exists R, N, S : [resA(R, N, S) \wedge (R = r)]$$

If one of the tests yields *true*, then there exists a corresponding entry in either the local or remote parent relation and the tuple $proresB(r, p)$ can be inserted. Otherwise the insertion has to be rejected. For the implementation of these tests, we use the functions *checklocalkey* and *checkremotekey* as introduced in Section 5.3.2. The remote test is evaluated using a remote state query on DB_A. A corresponding ECA rule for this partial constraint can be expressed as follows:

define rule *PartialReferentialConstraint*
on *creation of a new object in proresB*
if *checklocalkey yields false* **and** *checkremotekey yields false*
do *reject transaction*

Deletion check: Suppose the tuple resA(r,n,s) shall be deleted from DB_A. According to $RefInt_{resA}$ we have to ensure that there are no depending objects in the *prores* relations on DB_A and DB_B before we delete this item. Thus, we formulate the following tests for $local_{proresA}$ and $remote_{resA,proresB}$:

$$localtest_{RefInt_{resA}} : \nexists R, P : [proresA(R, P) \wedge (R = r)]$$
$$remotetest_{RefInt_{resA}} : \nexists R, P : [proresB(R, P) \wedge (R = r)]$$

The deletion check succeeds, i.e. $resA(r, n, s)$ can be deleted, if there are no dependent objects in $proresA$ and $proresB$. This partial constraint is represented by the following ECA rule:

define rule *PartialReferentialConstraint*
on *deletion of an object in resA*
if *checklocalkey yields true* **or** *checkremotekey yields true*
do *reject transaction*

Of course, we have to ensure that an entry in the parent table exists either in $resA$ or $resB$. This is realized using a key constraint on the researcher id as presented in Section 5.3.2.

Cascading Referential Integrity Constraints

With the extended functionality of Active Component Systems, we are basically able to realize cascading referential integrity on updates or deletions of tuples. Injected transactions can be executed during the evaluation of a partial integrity constraint to modify remote data stocks including even deletions, before or after the modifying operation is committed locally. If a tuple is deleted in the parent relation, we execute an injected transaction to delete all corresponding tuples in the dependent relation. Analogously, if a key value is updated in the parent relation, we cascade this update to the dependent relation by modifying the relevant entries in the remote database via injected transactions.

The corresponding partial integrity constraint for the parent relation is expressed similar to the partial rule without cascading presented in 5.3.3. We extend the rule to delete dependent objects from within the rule condition or action depending on the intended system behavior. Thus, we are able to delete entries in the dependent relation from within a *remote condition* or a *remote action*, *before* or *after* the local entry is deleted. Of course, if we modify data on more than one autonomous database system, we face the problem of atomic commitment in a multidatabase environment [77, 79]. A distributed update may lead the federation into a (temporary) inconsistent state in case of a failure or constraint violation. An analysis of potential consistency violations using injected transaction can be found in [108]. We need a recovery mechanism based on the concept proposed to detect inconsistencies and restore global consistency after an injected transaction has not committed globally.

Referring to our example, consider a third department C that maintains a database DB_C similar to the databases in departments A and B. Researchers in $resB$ and $resC$ are related to projects in $projA$ via the relations $proresB$ and $proresC$ respectively. A referential constraint with cascading would require the deletion of corresponding entries in $proresB$ and $proresC$ if the related parent entry in $projA$ is deleted. Now, during the partial referential constraint check with cascading triggered by a deletion of project (p_A, t_A, b_A) from $projA$ we have to delete the depending tuples (r_B, p_A) and (r_C, p_A) from $proresB$ and $proresC$ using injected transactions. It is possible that the deletion fails at one remote site due to conflicts with local transactions or if the database is not reachable. In that case we face an inconsistent global state. The literature proposes several approaches to react on such violations of atomic commitment, which are summarized in [17]:

Redo: Redo a subtransaction that only *writes* local data. If a redo transaction also fails then repeatedly resubmit it until it commits.

Retry: If a global transaction *reads and writes* local data, an aborted subtransaction can either be retried or compensated. In the retry approach, the failed subtransaction is resubmitted as a new subtransaction, thus possible

reading an writing different values. This approach requires a transaction to be retriable, that is that it will eventually commit after a sufficient number of times from *any database state*.

Compensate: The compensation approach tries to perform an undo of a subtransaction from the semantic point of view. A compensating transaction is a regular transaction and may not only consist of an inverse function of the original subtransaction. Another local or global transaction may see the state written by the subtransaction before it is compensated.

For the injected transactions as part of partial constraint checks the redo or compensate approach seem to be applicable. In our example, the trigger blocks the delete operation T_A on DB_A during the partial constraint check. The partial constraint check submits two injected transactions T_B and T_C to delete the depending tuples in DB_B and DB_C respectively. If T_B commits but T_C aborts then we can either redo T_C and commit T_A or compensate T_B and abort T_A.

5.3.4 Aggregated Constraints

As a representative for this type of constraint let us assume that the company has a budget limit for all research projects. Thus, it must be checked whenever a project is created or updated in DB_A and DB_B that the sum of all project budgets B_A and B_B does not exceed a certain value ε. We restrict our further considerations on the standard aggregate functions *min*, *max*, *sum*, and *count*. The average function *avg* must be calculated during a partial constraint check using *sum* and *count*. Furthermore, we assume that $agg(T, w)$ is an aggregate function that calculates the aggregate of an attribute w of a relation T. The function $totalagg(agg(R_{m_1}, w_{m_1}), ..., agg(R_{m_s}, w_{m_s}))$ computes the overall aggregate of partial aggregates for $m_u \in \mathcal{C}$ and $s = |\mathcal{C}|$. Please note that *count* is a semi additive aggregate function and the overall aggregate must be calculated as the sum of partial *count* aggregates.

These preliminaries provided, we can now formulate a global aggregated constraint for our example as

$$Sum_G : Sum_{projA} \wedge Sum_{projB}$$

with the partial constraints defined as

$$\begin{aligned} Sum_{projA} &: totalsum(localsum_{projA}, remotesum_{projB}) \ \leq \ \varepsilon \\ Sum_{projB} &: totalsum(localsum_{projB}, remotesum_{projA}) \ \leq \ \varepsilon \end{aligned}$$

Both databases have to check the total sum whenever an insertion or update occurs on B_A or B_B. Therefore, DB_A calculates its corresponding local and remote aggregate as

$$localsum_{projA} = sum(projA, B_A) \quad \text{and} \quad remotesum_{projB} = sum(projB, B_B)$$

using two functions

$$agglocal : schema(projA) \rightarrow \mathbb{R} \quad \text{and} \quad aggremote : schema(projB) \rightarrow \mathbb{R}$$

The calculation of the remote aggregate is realized using a remote state query on DB_B. The aggregates for DB_B are calculated analogously. Now suppose the tuple $projA(p, t, b)$ is inserted. According to Sum_{projA} we first compute $localsum_{projA}$ including the new value b and $remotesum_{projB}$ on DB_B. After we receive the result from the remote aggregation, we calculate $totalsum$ and compare the overall aggregate to ε. If the comparison yields $false$ then the insertion of $projA(p, t, b)$ is rejected. A corresponding ECA rule for this partial constraint can be expressed as:

define rule *PartialAggregatedConstraint*
on *update of B_A in $proj_A$* **or** *insertion of a new object in $proj_A$*
if $totalsum(localsum_{projA}, remotesum_{projB}) > \varepsilon$
do *reject transaction*

5.4 Discussion

An evaluation of our concept of global integrity maintenance can be found in [53] using a simple scenario. Besides the basic functionality, the main focus was on the portability of external notification programs among different databases. Thus, the concept was evaluated for various Enhanced Active Databases using Java and JDBC for Oracle and DB2, and C# and ODBC for the MS SQL Server.

The checking mechanism presented in this chapter is closely related to the Local Test Transaction Protocol (LTT) presented by Grefen and Widom in [46]. They state that LTT requires the database system to be "capable of performing a notification and waiting for an acknowledgment, all within a single transaction". Furthermore, to behave correctly, the local test must be evaluated within the same transaction in which the update occurred. Using Enhanced Active Database we are able to provide synchronous update delivery with local transaction blocking during a constraint check as required by LTT. Referring to the related work presented in Section 4.2 we state that an implementation of constraint checks using LTT and databases without enhanced activity has so far been impossible, since common event detection techniques implement asynchronous update delivery.

LTT exploits transaction capabilities provided by the local database system to perform a notification and wait for acknowledgment within a single transaction. Using the LTT we try to avoid remote checks by evaluating local tests first. If a local test has already failed, then we do not have to evaluate the cost intensive remote check using remote state queries. The implementation of a transaction-based protocol like LTT has to evaluate a constraint check *before* the triggering operation is committed. The external program is executed as part of the remote condition of a partial integrity rule. Our implementation certainly adopts all advantages and drawbacks of the applied LTT protocol. Thus, the implementation proposed is safe and accurate, which means that it detects all constraint violations and that, whenever an alarm is raised, there is a state in which the constraint is violated. On the other hand, since the relation is locked until the external program returns a result, the local ACDBS looses autonomy and the risk of deadlocks is relatively high, if relations in DB_R and DB_S are updated concurrently.

Due to the flexibility of our architecture, we are basically able to implement the entire set of protocols described by Grefen et al. Thus, to overcome the drawbacks of the LTT, we can modify the partial integrity constraints to implement the Materialized Delta Set Protocol (MDS), which increases autonomy and reduces the risk of deadlocks. Therefore, we maintain an additional relation ΔR, which stores an accumulated set of updates of the original relation R. The constraint checking mechanism is then evaluated using ΔR instead, so the original relation is not locked during the check. This enables at least concurrent read access to R while updates must still be delayed until the lock is released. In our architecture ΔR is maintained using the active capabilities of the DBMS. We define a local rule on R to copy all updated items to ΔR. The partial integrity rule including the remote checks as presented above is then expressed over the Materialized Delta Set ΔR.

A generalization of the constraint checking mechanism to more than two sites is tightly corresponding to the implemented integrity checking protocols. As already described in [46], most of the constraints that involve more than one site can be decomposed into a couple of constraints, which are expressed over exactly two databases. If there is the need for multi-site constraints, the authors propose to use non-transaction-based protocols like Direct Remote Query (DRQ) or Timestamped Remote Query Protocol (TRQ), which can both be implemented using our architecture. To avoid locking of the updated relation we adjust the partial integrity rule to be evaluated *after* the modifying operation is committed.

The protocols could be optimized according to the Demarcation Protocol presented in [10]. This protocol is particularly suitable for arithmetic constraints like aggregated constraints, but can also be used for key or referential integrity constraints. The Demarcation Protocol can be seen as an extension to the LTT and thus be implemented using our architecture.

5.5 Global Constraints with COMICS

To overcome the limitations of the basic concept presented in the previous section and to enable global constraint checks over more than two sites, we have developed the COMICS system, a global **Co**nstraint **M**anager for **I**nteractive **C**omponent **S**ystems. COMICS is an extension to the basic concept insofar as it introduces an external constraint manager component that is used to define interdependencies and mappings between the data sources and to execute constraint checks on remote ACDBSs of a federation. The return values of the checks determine if a local transaction shall be committed or aborted. In this section we sketch the basic architecture of COMICS and describe its functionality.

Analogously to the basic concept, a global constraint is decomposed into a set of partial integrity constraints, which are enforced by the affected ACDBSs using triggers, but the rule definition in COMICS consists of only one local and one remote integrity check clause. For example, a partial constraint I_{R_c} on relation R on a component database c can be formulated as:

define rule *PartialIntegrityRule* I_{R_c}
on *event which modifies* R_c
if *local check on* R_c *yields false* **or**
remote check performed by the constraint manager yields false
do *reject transaction*

The local check is directly performed on the local system, while the remote check is performed and supervised by the external COMICS Constraint Manager. During the remote check, the Constraint Manager (CM) queries interrelated component databases to detect violations of the corresponding global constraint. COMICS currently supports the same set of global constraints like the basic concept: global key constraints, global referential integrity constraints, and global aggregated constraints with the standard aggregate functions *min*, *max*, *sum*, and *count*. Although we are basically able to implement referential constraints with cascading using injected transactions, we restrict our considerations in the focus of this work to remote checks performing remote state queries only.

5.5.1 System Overview

The COMICS system is specifically designed for global integrity maintenance among a collection of Enhanced Active Databases and does not support global transactions itself. COMICS can be included into the federation layer of the federated information systems as a constraint management subsystem that assists the ACDBS with the checking of partial integrity constraints. The management of schema mappings is shifted from the ACDBSs to an external repository hiding semantics from the components and thus reducing complexity of the ENPs.

Figure 5.3 displays the architecture for global integrity checks enhanced by the following COMICS components:

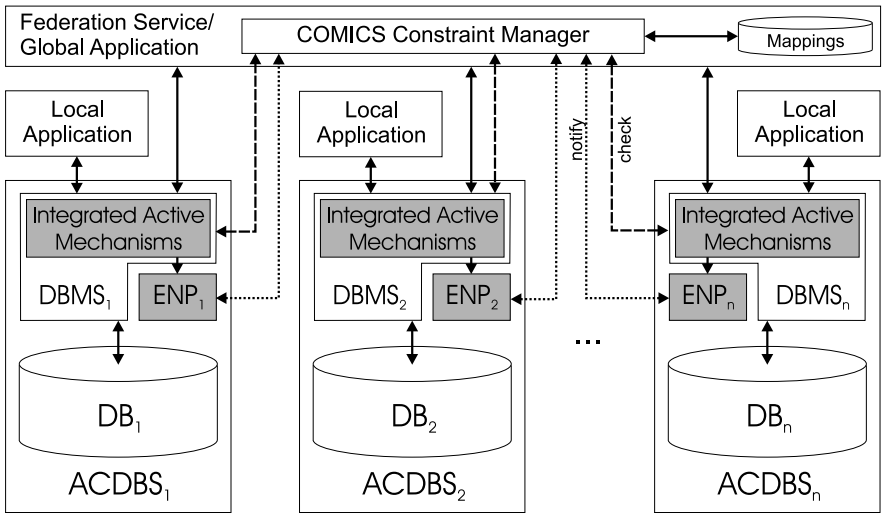

Figure 5.3: Federated Information System with COMICS Constraint Manager

Constraint Manager: The Constraint Manager enables and controls the communication between the ACDBSs. It is responsible for the execution and evaluation of remote checks on related component databases as part of partial integrity constraints. Therefore, it implements constraint check algorithms for commonly used global constraints and comprises a query interface for various EADBSs to poll the required data from the remote sources. To overcome the differences of the database schemata, the constraint manager needs access to a repository, which maintains mappings between the local schemata.

Mapping Repository: This repository stores the mappings between the local schemas of the ACDBSs. Mappings are created when a new component database is attached to the Constraint Manager for integrity maintenance. The repository mainly contains table-to-table and attribute-to-attribute mappings to overcome naming differences imposed by local design autonomy. To perform an integrity check, the Constraint Manager retrieves interrelated tables and attribute from the repository to create the concrete remote state queries.

The COMICS components can be implemented as part of the federation layer or global application as well as part of a standalone constraint management layer. If part of the federation layer, it can access the mappings created during the integration of the local schema into the global schema. If no such mappings exists, they have to be created and maintained separately.

Department	Database	Relations
A	DB_A	$resA(\underline{R_A}, N_A, S_A)$ $projA(\underline{P_A}, T_A, B_A)$ $proresA(\underline{R_A}, P_A)$
B	DB_B	$resB(\underline{R_B}, N_B, S_B)$ $projB(\underline{P_B}, T_B, B_B)$ $proresB(\underline{R_B}, P_B)$
C	DB_C	$resC(\underline{R_C}, N_C, S_C)$ $projC(\underline{P_C}, T_C, B_C)$ $proresC(\underline{R_C}, P_C)$

Table 5.2: Interrelated relations in research departments A,B, and C

5.5.2 Checking Constraints with COMICS

We now describe the constraint checking process in COMICS using the key constraint example presented in Section 5.3.2. Therefore we consider a third research department C that maintains its individual database DB_C for its projects in the relations displayed in Table 5.2.

The company wants to ensure the same global key constraint as above saying that the project number in each database must be globally unique. Checking this constraint now involves all three departments:

$$Key_G : Key_{projA} \wedge Key_{projB} \wedge Key_{projC}$$

The global key constraint Key_G is decomposed into three partial key constraints for each affected database DB_A, DB_B, and DB_C. Now, each time a new project is inserted or an existing project is updated in one of the departments, we have to check the uniqueness of the project identifier in all three databases by executing the partial integrity check on the database, on which the update occurred. In the following we exemplarily describe the checking process for the partial key constraint on the relation $projA$ in database DB_A when a new tuple $projA(p_A, t_A, b_A)$ is inserted.

Basic Process

Checking global integrity with COMICS basically involves four steps:

1. A local update on $projA$ is detected by a trigger component. It subsequently evaluates the corresponding partial key constraint including a local key check and a remote key check. If the local check fails then the local transaction is aborted, otherwise the remote check is executed.

2. The remote check for global uniqueness is performed by the Constraint manager. The trigger uses an ENP to connect to the external Constraint

Manager via a remote procedure call to initiate the process. This call is executed synchronously, i.e. the local transaction is blocked until the call returns.

3. The Constraint Manager receives the message from the ENP including the name of the database and relation that was updated. It looks up a set of schemas of remote component databases to which the updated schema is related to and schedules the query of these databases.

4. After the interrelated schema items have been identified, the Constraint Manager actually queries the remote databases to check if p_A already exists. If the value exists in at least one of the sources, then the update transaction on DB_A is rejected.

We will now take a closer look at the components performing the essential tasks: the trigger to react on local events, the ENP to connect to the CM, and the external CM that executes the partial constraint check.

Trigger Definition

The key constraint has to be checked whenever data is inserted or updated in the relation *projA*. Analogous to the basic concept, we define a trigger to react on inserts and updates as follows:

```
CREATE TRIGGER key_project
BEFORE INSERT OR UPDATE ON projA
REFERENCING NEW AS n
FOR EACH ROW
  // local test
  DECLARE localtest INTEGER;
  SET localtest =
    (SELECT COUNT(Pa) FROM project WHERE Pa = n.Pa);
  IF (localtest > 0) THEN
    RAISE LOCAL ERROR;
  // remote test
  ELSE IF
    (checkremotekey('DB_A', 'projA', 'Pa', n.Pa) = -1) THEN
      RAISE REMOTE ERROR;
END
```

When a local transaction inserts or updates data items Δp in *projA*, the trigger is executed *before* the transaction is completed. The trigger has access to Δp via a temporary table `NEW` provided by the DBMS. Corresponding to the partial integrity constraint, the trigger executes two integrity checks: a local

test on local data and a remote test on interrelated data on other component databases. First, the trigger checks if the new or updated project number already exists in the local data stock. This can be done using a local query which counts the number of tuples with the corresponding project number. If a project with this number already exists, then the transaction is rejected without executing the remote check. Otherwise, the trigger executes the remote check via the external program *checkremotekey* to check the existence of the project number in *both* related project databases DB_B and DB_C. If the remote check returns false, i.e. a tuple with the respective P_A (denoted as `Pa` in the trigger definition) was found in DB_B or DB_C, then the transaction is rejected.

The External Notification Program

Contrary to the basic concept we do not directly access the remote component databases from within the external programs, but delegate the constraint check to the COMICS Constraint Manager. Figure 5.4 shows the basic interactions during global integrity checks using COMICS. Implementing the push-based synchronous event notification mechanism (see Section 4.3.2) we use an External Notification Program to connect to the Constraint Manager to initiate the check for the remote part of the partial key constraint. The trigger blocks the local transaction, calls the ENP, and waits for the program to return the result of the remote check. The ENP performs a remote procedure call (RMI in the case of Java) to the Constraint Manager which in turn executes the actual existence check of the key value `n.Pa` in the remote database DB_B and DB_C using, for example, JDBC connections.

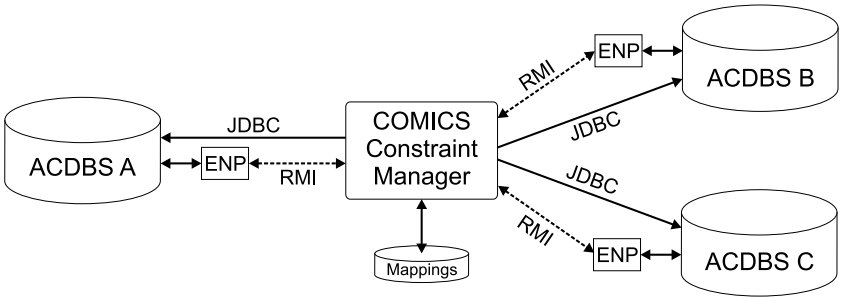

Figure 5.4: Constraint checking with COMICS

The following function is executed during a remote condition by the trigger:

$$checkremotekey(db, table, attr, value) = \begin{cases} 1 & : \text{ if key does not exist} \\ -1 & : \text{ if key exists or error} \end{cases}$$

Although the signature and codomain of the *checkremotekey* in the basic concept and COMICS do only differ slightly, they implement completely different logics. While the external program in the basic concept is a remote state

query executed directly from within the DBMS, it performs a notification of an external component in COMCIS. The function takes a database identifier *db*, the name of the relation *table*, and the name of the key attribute *attr* where the data modification was detected locally. Furthermore, the function requires the *value* of the inserted or updated project key that has to be checked against the remote databases. The function returns 1, if the key does not exist remotely or -1 if it exists or in case of a system failure or network breakdown. The return value is used by the trigger to determine if a transaction shall be committed or aborted. Since we enforce pessimistic synchronous integrity checks, the transaction has also to be rejected, if a system failure or network breakdown occurs during communication with the constraint manager or one of the remote component databases.

The Constraint Manager

In the first version of COMICS the Constraint Manager is a centralized component, that plans, initiates, and monitors the execution of remote state queries among a set of interactive component databases. It receives synchronous notifications from the databases via remote procedure calls and thereupon executes queries on interdependent component databases. The ENP calls a function *checkkey* of the Constraint Manager which can be defined as follows:

```
boolean checkkey(db,table,attr,value) {
  boolean success = true;
  db[] = mappings.lookupMappings(db,table,attr);
  for all (db[i]) {
    if (value exists in db[i].table)
      success = false;
    }
  }
  return success;
}
```

In our example, the Constraint Manager receives a synchronous update notification via the EPC `checkremotekey('DB_A','projA','P_A','p_A')` executed on DB_A with `P_A` and `p_A` being the name and value of the key attribute in $projA$. It thereupon looks up interrelated schemas from the mapping repository, in our case $projB$ and $projC$. Knowing the concrete databases, relation, and attribute name it then queries $projB$ and $projC$ for the existence of p_A. If the value cannot be found both databases then `checkkey` returns *true*, i.e. the update operation on $projA$ is committed.

In our first Java prototype of a mediator-based information system, the CM is implemented as a centralized component in the architecture. Since all remote checks in the entire information system are performed by one CM, this obviously

builds a performance bottle neck. However, it guarantees global consistency since updates on interrelated data that require partial constraint checks are globally serialized by the Constraint Manager. Furthermore it prevents global deadlocks and ensures atomic execution of partial constraint checks. Besides, semantic information is moved from the sources to the Constraint Manager which makes it easier to add or remove interdependent sources or to maintain schema mappings. In our Java prototype the Constraint Manager queries the databases using JDBC and SQL. The method *checkkey* is implemented as *synchronized* which means that it can only be called once at the same time. Future implementations could consider decentralized Constraint Managers that are assigned to only a subset of the data sources in the information system. They are able to communicate with each other during partial constraint checks if a check requires access to sources of multiple CMs.

5.6 Related Work

In the last years, research on integrity constraints in heterogeneous environments mainly considered the simplification, evolution, or reformulation of constraints rather then mechanisms or protocols for integrity checking. A closely related concept in terms of the rule structure and constraint types is presented in [44]. The authors use private and public global constraints to define dependencies between data in different databases, similar to the partial constraints presented in this chapter. One of the main differences is the use of a layered approach to support the active functionality required for event detection and rule processing. A reactive middleware based on CORBA encapsulates active and passive sources and processes rules using an external remote rule processing mechanism. Furthermore every component is assumed to have an Update Processor to execute local update requests, but the local relation cannot be locked during the evaluation of remote conditions.

The metadatabase approach [56] uses a rule-oriented programming environment to implement knowledge of information interactions among several subsystems. Each subsystem is encapsulated by a software shell, which is responsible for monitoring significant events, executing corresponding rules, and interacting with other shells. Although conditions can be evaluated in a distributed way, the rule processing itself is still centralized.

A distributed rule mechanism for multidatabases is presented in [64] as part of the Hyperion project. A distributed ECA rule language is introduced, which is mainly used to replicate relevant data among data peers in a push-based fashion. Rules are processed by a rule management system that resides in the P2P layer on top of a peer database. The X^2TS prototype [72] integrates a notification and transaction service into CORBA using a flexible event-action model. The architecture presented resembles a publish/subscribe system, whereas publishing of

events is non-blocking. Another middleware approach for distributed events in a heterogeneous environment is presented in [66]. CORBA-based, distributed, and heterogeneous systems are enhanced by Active DBMS-style active functionality. The architecture uses wrappers with event monitors to detect data modifications in the data source.

A common characteristic of the architectures just mentioned is the use of a layered approach with event monitoring to somehow notify a mediating component (e.g. a constraint manager, rule processor, or middleware component) about events occurring in the local database. If the source is not monitored, the notification mechanism is generally based on active capabilities of the underlaying database management system, but there is so far no detailed description of this interaction published.

The most distinctive characteristic of our concept is the direct usage of existing active capabilities of modern database management systems without the need for wrappers or monitoring components. Since a remote condition is evaluated during the execution of a trigger, it is irrelevant if the triggering transaction was a global or local update. We benefit from the active functionality of the DBMS in terms of transaction scheduling, locking, and atomicity, resulting in a synchronous integrity checking mechanism. Especially the ability to rollback updates depending on a remote state query makes corrective or compensative actions superfluous.

Chapter 6

Tightly coupled Wrappers

The interconnection of heterogeneous and autonomous data sources for information sharing in federated information systems demands for flexible and extensible integrative components. In this chapter we present a universal architecture for building wrapper components to access various types of data sources. Our wrapper comprises an event detection subsystem to detect modifications of the data stock and propagate them to the federation layer for further processing. The architecture proposed includes a Notification Interface to especially support push- and pull-based Active Event Notifications from Enhanced Active Databases as presented in detail in Section 4.3. Due to this bidirectional communication, the wrapper gets *tightly coupled* to its data source. The content of this chapter is based on a paper originally published as [93].

We start with a detailed description of the architectural components of the wrapper and the functionalities they provide in Section 6.1, followed by a schematic overview of the event detection subsystem in Section 6.2.

6.1 Wrapper Architecture

As already discussed in Section 2.2, component databases of a federated information system are commonly integrated via a wrapper layer. Wrappers encapsulate heterogeneous data sources to convert the source-specific interface into a source-independent interface the federation layer expects. A wrapper can act as a simple adapter for queries and corresponding query results or attach additional functionality to the data source to build up more complex infrastructures as needed for federated information sharing environments or interoperable collections of databases.

We now present a wrapper architecture for the integration of various data sources into a federated information system. The architecture comprises both, a query subsystem and an event detection subsystem to detect and propagate modifications of the data stock to the federation layer. It particularly provides

a Notification Interface to support Active Event Notifications from Enhanced Active Component Systems as described in Section 4.3. The components of the wrapper itself communicate via methods and function calls written in the wrapper programming language and return data types or objects. Figure 6.1 depicts the components of the wrapper architecture, which are described in the following.

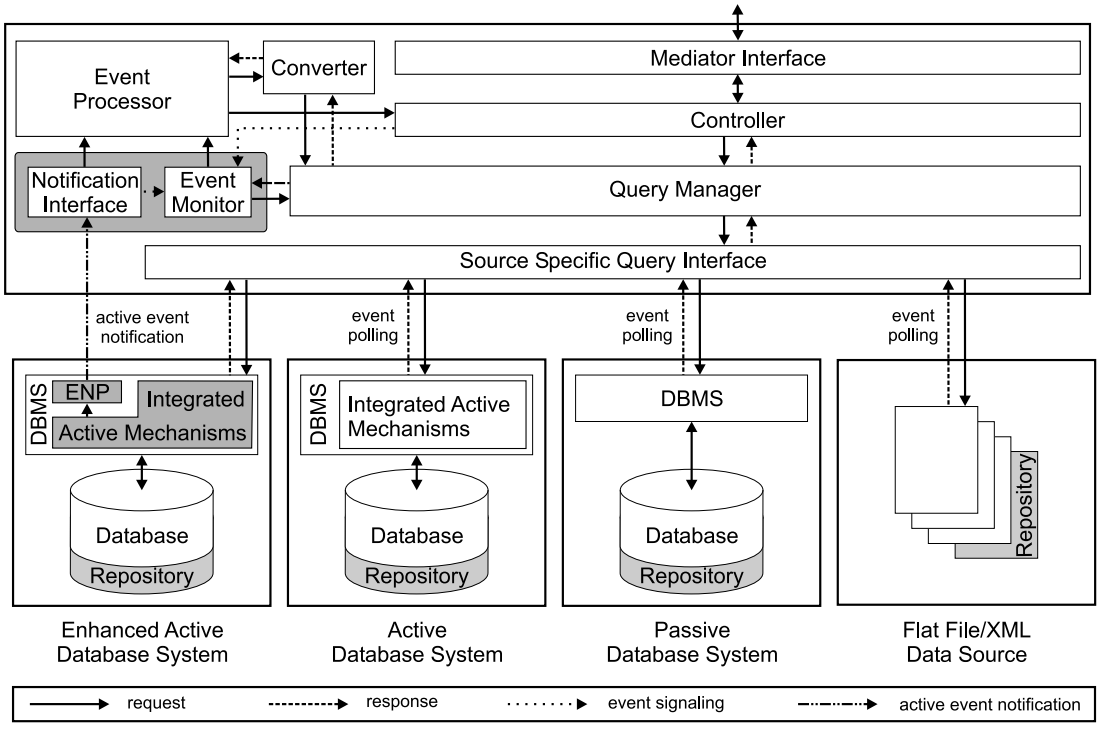

Figure 6.1: Wrapper architecture with event detection subsystem.

Heterogeneous Data Sources: Our architecture basically supports any type of data source, but the functionality that can be offered to the mediation layer by the wrapper strongly depends on the capabilities of the underlaying data source. A data source is connected to the wrapper by a source-specific query interface. Types of data sources can be, for example, unstructured or semi-structured files, relational or object-oriented database systems, or directory systems, with our without integrated active mechanisms. The architecture is designed to support event detection for all data source activity classes, but especially Enhanced Active Databases with the ability to actively signal updates to the wrapper for real-time change delivery.

Source Specific Query Interface: The source-specific query interface is a software component that provides basic operations on the data stock. It knows

how to open and close a connection to a source and what types of queries are supported. Furthermore it translates query results into a representation format of the programming language. This can be, for example, a result set, a multidimensional array, or a heterogeneous collection of basic data types. The required query language and the returned query results significantly depend on the functionality provided by the data source. Most databases with database management systems support SQL as a standardized query language and return a resource identifier which points to the result of a specific query. A well-established package which can be associated to this architectural component is the Java Database Connectivity Framework (JDBC). It supports various types of data sources including even query interfaces for flat or XML files. The Query Manager calls functions of the source-specific query interface to communicate with the data source and to read or modify its data stock.

Query Manager: The Query Manager consists of a set of functions which encapsulate the entire communication from the wrapper components to the data source. These functions are specifically designed for the desired data source functionality of the overall system. The Query Manager uses the source-specific query interface to open and close connections, send queries, and receive corresponding results. Each query function executes a single query or a parameterized family of queries to read or modify the data stock. The functions can be divided into two types of interaction: data requests and repository requests. Data request functions provide an interface to access local data in the data source, whereas repository requests are used to access meta data stored in the repository. Both, controller and event monitor, call the predefined query functions to interact with the data source. They receive the corresponding query result as data types or objects as return values from the functions. Due to the centralization of the data source access in the Query Manager and the separation of Query Manager and source-specific query interface we achieve a portable wrapper solution for federated information systems.

Converter: Since a wrapper is used to convert queries and data from one model to another, we need a component to convert source-specific data into an exchange format the mediation layer expects and vice versa. This conversion is done by the Converter component. On the one hand, it is used by the Query Manager and the Event Processor to transform local data into the desired exchange format before it is sent to the mediation layer. On the other hand it converts external data received from the mediation layer into the internal representation format understandable to the Query Manager. In the scope of this thesis we do not provide a specific exchange format, since the choice of a format depends on the properties, characteristics, and functionality of the overall system. However, a common exchange format for the use in the mediation layer of a federated

information systems could be based on RDF or OWL [88].

Controller: The Controller controls the interaction between the mediation layer and the data source. In particular, it maps external requests to internal query functions of the Query Manager, and forwards results for output. An external request may result in a sequence of database operations which are also coordinated and processed by the Controller component. Furthermore, it manages all the meta data required in the repository. The Controller also provides a set of event notification functions, which are used by the Event Processor to signal data modifications of the local source. Events can thus be reported to the mediation layer for further processing.

Mediator Interface: The Mediator Interface is the communication unit for interaction with the mediation layer. The functionality and concrete implementation of the interface depends on the type of infrastructure to be established. Possible infrastructures for mediation layers could be based on for example RMI or CORBA, Server-Client communication over specified protocols, web services, publish-subscribe, or peer-to-peer. The Mediator Interface establishes connections to remote systems and handles incoming or outgoing requests between the mediation layer and the wrapper controller.

Repository: The repository contains all meta data required for the operation of the wrapper component and the interaction with the mediation layer and the data source. This includes configurations and properties, data source descriptors, login information, and access control lists. The repository is managed exclusively by the controller of the wrapper, but it is stored directly in the local data source. Thus, requests to data and meta data in the repository can be processed in a uniform manner by the Query Manager. The Query Manager provides the Controller and event monitor with query functions for repository management and access. The main advantage is the homogeneous storage of both, data and meta data, and the centralized access via the Query Manager.

Event Monitor: The Event Monitor is the active component of the event detection subsystem. It detects data modifications in data sources that are not able to actively propagate events to the wrapper, like for example active database systems without enhanced activity. The Event Monitor obtains information about the data items to be scanned from the repository. Using this information it periodically scans relevant parts of the data stock for modifications. In the case of a relational database a *monitoring schedule* in the repository could contain the name of a relation and the time period in which the relation should be scanned. The overall meta data required for monitoring data modifications depends on the change detection algorithm implemented by the Event Monitor, which in turn

depends on the type of local data source. The Event Monitor is executed as a subprocess (thread) at startup of the wrapper and does not provide any operational interface to the remaining wrapper components. Like the Controller, the Event Monitor uses the query functions of the Query Manager to access the data source. If a relevant data modification is detected, the modified data items are extracted from the data source and reported to the Event Processor. For a detailed description of the design and functionality of the Event Monitor we refer to Section 4.1.1.

Notification Interface: The Notification Interface is the passive component of the event detection subsystem of our wrapper architecture. It receives all update notifications directly from an EADBS via Active Event Notifications (see 4.3 for details). It consists of a source-specific set of functions, which are called by the underlaying EADBS. After a change has been signaled to the Notification Interface, the modified data items are reported to the Event Processor.

Event Processor: All events, monitored by the Event Monitor or signaled by the Notification Interface, are processed by the Event Processor. It receives modified data items and converts them into the exchange format using the Converter component. Afterwards, it forwards the changes directly to the wrapper controller.

6.2 Event Detection Subsystem

Since the ability to react on events on a component database is essential for many information system architectures, we have included an event detection subsystem into the wrapper architecture. It comprises the Event Monitor, the Event Processor, and the Notification Interface described above. The event detection subsystem is adjustable to all data source activity classes and data models so that basically all monitoring technique presented in Section 4.1 can be implemented depending, of course, on the capabilities of the underlaying data source. We distinguish between two main task of the event detection subsystem: pull-based event polling for data sources without enhanced activity, and push-based Active Event Notifications for enhanced active databases. The implementation of both tasks within the wrapper is sketched in the following:

Event Polling: The Event Monitor executes the change extraction process using a source-specific change capture method. This method is implemented in a set of query functions provided by the Query Manager. According to the event detection phases we have identified in Section 4.1.1, the extraction of changes has to be triggered by a certain event during the notification phase. Referring to Figure 6.1, the pointed arrows depict the notification channels for the notification

phase in our wrapper architecture. Depending on the concrete delivery schedule this could be a clock in the monitor that triggers periodic updates, a request from an application or user via the wrapper controller, or an Active Event Notification from an EADBS for real-time change delivery. Data sources without enhanced activity are monitored using a pull-based capture method providing deferred or periodic change delivery. The connection between the wrapper and the source is unidirectional, which means that the source has no knowledge of the encapsulating wrapper component. However, our wrapper also supports immediate pull-based notifications from Enhanced Active Databases. The sequence diagram (Figure 6.2) depicts the interaction of the components during a pull-based asynchronous notification (4.3.1) for relation R starting with the execution of the ENP.

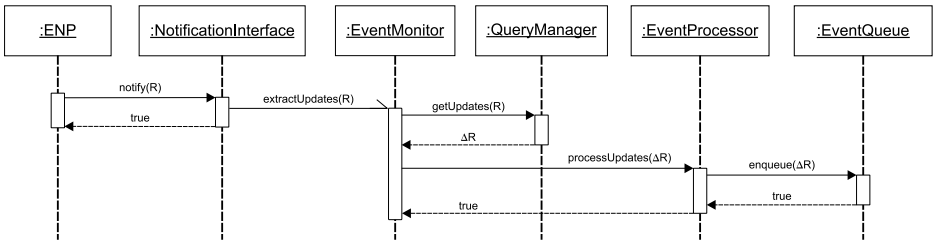

Figure 6.2: Asynchronous pull-based notification in the wrapper

The ENP connects to the NI and signals the update immediately after it occurred on R. The NI asynchronously forwards the notification to the Event Monitor which thereupon initiates the extraction phase. It uses the query functions in the Query Manager to capture the changes ΔR in the source, for example using a trigger-based incremental capture method. Thus, access to the source is controlled by the Query Manager hiding source-specific implementation details from the Event Monitor. The changes are forwarded directly to the Event Processor where they could be put into an event queue (as recommended for asynchronous updates) and converted into a common representation format using the Converter. Afterwards, the Controller decides how to react on the updates.

Push-based Notification: During the push-based notification, the data source pushes changed records to the wrapper. The ENP connects to the NI and transfers the changes using the synchronous or asynchronous notification mechanisms (4.3.2 and 4.3.3) depending on the application field. The push-based approach does not need interaction with the Event Monitor or Query Manager since the updates are already sent to the wrapper as part of the notification phase.

Figure 6.3 depicts the sequence diagram for a synchronous push-based notification process as required for global integrity maintenance. A relation R is updated and a trigger subsequently executes the ENP to push the updates (ΔR)

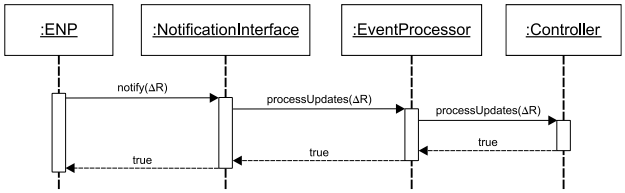

Figure 6.3: Synchronous push-based notification in the wrapper

to the Notification Interface. The messages between all components are sent synchronously. Thus, the update transaction is blocked until a return value is received from the Controller. From the NI, the updates are forwarded to the Event Processor and processed like in the pull-based approach. During synchronous notifications the Event Processor has to synchronously inform the Controller about the updates which, in turn, has to decide how to react. For example, it could use the mediator interface to contact a constraint manager in the federation layer or directly contact a remote component database to evaluate a partial integrity check. Please note that during a synchronous notification process all messages must be sent synchronously throughout the wrapper and external components.

In both examples, the data source actively communicates with its wrapper component to signal a change of state via an External Notification Program. Due to this bidirectional communication we say that the wrapper is *tightly coupled* to its data source. A tightly coupled wrapper is perfectly suitable for event-based federated information systems. They can detect changes in various data sources, convert them into a common representation format, and actively propagate them to the federation layer.

6.3 Application Fields

The wrapper architecture presented in this chapter is especially designed for federated information systems and loosely coupled database federations. The event detection subsystem allows interactions between the participating data nodes triggered by data manipulations in the sources, which is a crucial task in many autonomous information sharing environments. The architecture adapts both design patterns, an *adapter* to provide a unified interface for the mediation layer and a *decorator* which allows the extension of the functionality of the wrapped data source. Possible applications for our architecture could be, but are not limited to:

Event publishing: One application field of our wrapper architecture is the publishing of events in federated information systems. It can be used to signal data modifications to event brokers, constraint managers, or global rule

managers in the federation layer. Events are converted into a common representation format and propagated to the federation layer via the mediator interface as synchronous or asynchronous messages.

P2P-based information system: The wrapper architecture is part of a loosely coupled multidatabase architecture (see Chapter 7). The system achieves a reasonable tradeoff between autonomy and information sharing among the participating databases. Each data source decides autonomously which kind of information to share. The data sources are organized as a P2P system and communicate using push- and pull-based protocols. Data and schema modifications are actively propagated to subscribing databases, which are herewith able to maintain a local up-to-date replica of the data required.

Mobile databases: The architecture is ideal for the implementation of wrappers for mobile databases [8, 9]. As long as the mobile device is connected to the network we can use the event detection subsystem to synchronize data modifications on the mobile database with the backend. If the device is disconnected from the network, the updates can be accumulated in a cache on the mobile device to synchronize with the backend when the network connection is reestablished.

We have evaluated our wrapper architecture for a set of relational databases (DB2, Oracle, MySQL) in Java using JDBC as the source-specific query interface. A detailed description of the Event Monitor is given in [68]. The multi-threaded monitor periodically scans relations using an implementation of the snapshot differential algorithm for relational databases. The Notification Interface to support Active Event Notifications by EADBSs was implemented and evaluated in [113]. External notification programs were created as Java Stored Procedures for DB2 and Oracle whereas the communication between the ENPs and the Notification Interface was realized using Java RMI.

6.4 Related Work

In this section we give an overview of existing projects relating to wrapper architectures where the wrappers are mainly developed as part of a complex system. In this overview we focus on general wrapper architectures providing query and data translation. For a discussion of event detection architectures we refer to Section 4.2.

Two well-known wrapper-based information systems are TSIMMIS [85] and Garlic [99]. In both architectures the focus lays on the processing and optimization of queries. TSIMMIS provides a toolkit for the rapid development of wrapper

components for different data sources. Based on a *Query Description and Translation Language (QDTL)* description that is used to describe the queries that are supported by the underlaying source, a *converter* component in the architecture decides if a query is directly, logically, or indirectly supported by the source. Besides a QDTL description, a wrapper implementor provides a source driver and a definition of the source output and the information required. The wrapper is queried using a specific query language MSL and returns results in the Object Exchange Model (OEM) as common data model. The TSIMMIS wrapper architecture does not comprise an event detection module. The Garlic wrappers provide an object-oriented view on the underlaying repository to the GARLIC middleware. Actual data is received using method invocations on these Garlic objects. The wrapper furthermore participates in query planning and finally executes queries on the source. Garlic objects are defined in the Garlic Data Language (GDL), which is a variant of the ODMG's Object Description Language. Like TSIMMIS, Garlic wrappers do not implement event detection mechanisms. More recent work on wrappers focuses on the formalization, automatic generation, and maintenance of wrapper components, especially for semi-structured data [24, 98].

Contrary to the approaches just mentioned, we do not restrict our considerations to a specific query language or representation model. We rather describe the conceptual units that are required to implement the intended functionality. Query language translation and model conversion are performed by the Converter component and depend on the concrete application field and type of the encapsulated data source. However, the related concepts can be used to simplify speed up the wrapper implementation for different data sources.

Chapter 7

The DÍGAME Architecture

In this chapter we present an architecture for a federated information system that uses tightly coupled wrappers to interconnect a collection of autonomous and heterogeneous data sources based on peer-to-peer concepts. It achieves a reasonable tradeoff between autonomy and information sharing among both, permanently available and volatile data sources. Each data node decides autonomously which kind of information to share. Data availability, query performance, and up-to-dateness on each participating data node is improved using a push-based replication strategy, which propagates data modifications over multiple nodes. The content of this chapter is based on papers originally published as [89, 90, 94].

We start with a short introduction to our architecture in Section 7.1 followed by a description of its basic functionality using a motivating example. Section 7.3 presents the main architectural components with a particular focus on the location of the required schemas definitions. The main characteristics of DÍGAME are discussed in Section 7.4. Section 7.5 describes the extensions required for the basic tightly coupled wrapper to implement the DÍGAME functionality. The chapter closes with an overview of related work.

7.1 Introduction

Peer-to-peer is a promising concept for building up highly dynamic, flexible, and extensible information systems. Although designed for a huge number of users, those systems are particularly useful for dynamic information sharing among a manageable number of equal partners. The architecture of a company wide information system has to be applicable to the data policy of the company and vice versa. The question of data ownership determines the composition of the information platform, while it has to ensure a high level of consistency and fail-safety. Following this argumentation we are basically able to classify the structure of a corporation according to the classification of multidatabases in Sheth and Larson [102], i.e. in loosely coupled or tightly coupled companies. This classification is

applicable to both, intra- and inter-enterprise collaboration. The latter includes virtual corporations, fused solely for specific business purposes. Departments within a loosely coupled corporation possess a high degree of local autonomy being able to decide by themselves on their global participation level. On the other hand, tightly coupled corporations are controlled and administered by a strong centralized management, hence their departments have only a few administrative rights, but rather collect or process data. High availability and data quality build the basis for decision making, using e.g. a company wide workflow management system to control and organize processes or to check and reduce costs, thus to ensure competitiveness. Cooperating departments need shared access to consistent data to be able to increase their productivity.

In this chapter we present the DÍGAME architecture, a **D**ynamic **I**nformation **G**rid in an **A**ctive **M**ultidatabase **E**nvironment, which interconnects heterogeneous and autonomous data sources to support loosely coupled intra- and inter-enterprise collaboration. We have enhanced the multidatabase architecture of Heimbigner and McLeod [52] with Peer-to-Peer (P2P) concepts to offer a flexible information grid with high data availability to provide each participating node of this grid with all the data required. Extending the approach of Heimbigner and McLeod, our architecture enables the sharing of information among both, permanently available and volatile data sources (e.g. mobile databases [8]) without any central component. For that we have included a push-based replication mechanism into our architecture that propagates data modifications over multiple nodes. Thus, we are able to ensure high availability of data, even if the original data source is temporarily unreachable. Additionally, the replication of data increases query performance, since we do not have to query remote data sources. An information sharing environment, which comprises the information shared by interconnecting heterogeneous and autonomous data peers using our architecture, shall in the following be referred to as an information or data grid [25].

7.2 Basic Functionality

We start with the basic functionality of our enhanced multidatabase architecture using a motivating example.

Consider a worldwide operating company planning the launch of a new product (Fig. 7.1). We assume that there are three departments involved in this business process: the executive board (management), the sales office, and the product engineering department. Each department manages its own database to store the information for which it is responsible in an autonomous way. The management produces basic data of the product (A) including deadlines, descriptions, workflows, and additional objectives. This management information is substantial for the further product development and the work in the participating departments.

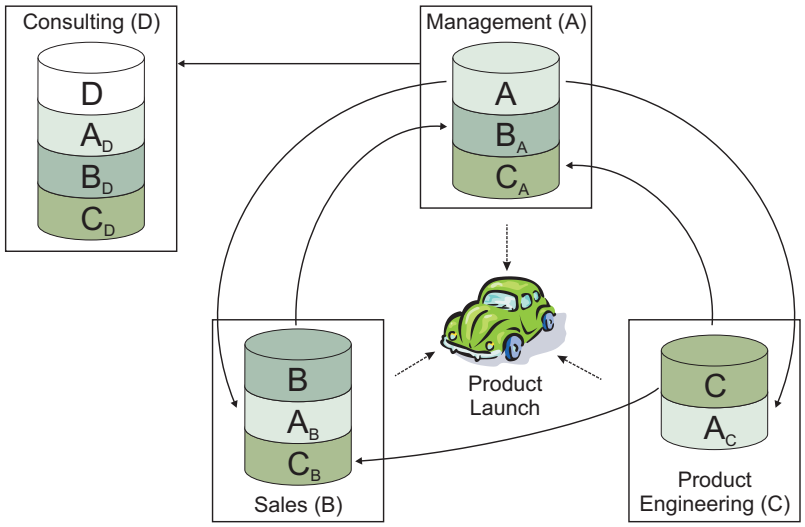

Figure 7.1: Collaborative Work with DÍGAME

The product engineering department uses a predefined part of that management data (A_C) as basic conditions for the concrete implementation and technical realization of the product. Local applications create additional data which has to be stored separately (C). According to the product engineering the sales department enriches the authoritative management data (A_B) with concrete concepts for the upcoming product launch (B). Furthermore concrete development plans of the product engineering are required to prepare sales strategies (C_B). Both, sales and product engineering departments, concretize the strategic guidelines of the management in their specific assignment. To keep track of the costs and the progress of the project, it is indispensable for the management to access the product engineering and sales department's relevant information just mentioned (B_A, C_A).

Sharing data within this company using our architecture is realized using a push-based replication strategy to improve data availability, query performance, and up-to-dateness on each participating data node. Hence, the data source actively propagates data updates to relevant peers, which are herewith able to maintain an up-to-date replica of the imported data. For example, the creation of a new replica of management data on department B is realized as follows: each data source of the departments A, B and C is wrapped by a source-specific wrapper component. These wrappers build up a communication layer, which enables the departments to interact pair-wise using a common protocol. This union adopts Peer-to-Peer concepts and operates without any central administrative instance. Due to these characteristics the combination of such a data source and its related wrapper component can be named as a (data) peer.

The administrator of peer A makes a subset of its own local data accessi-

ble using the administrative interface of the wrapper component. The export schema [52] created this way is managed by the wrapper component and specifies the information that the department is willing to share. The information concerning the access control to local data by remote peers is attached to the export schema. Peer B is now able to import the data into its local database subscribing to a specific part (A_B) of the published data, i.e. the data required by the department. During this subscription process the data target (subscriber) informs the data source (publisher), which subset of the export schema it is willing to import. The data stock A_B is then transferred to the subscriber to perform an initial filling. If a data or schema modification is detected by the wrapper of the publisher, all relevant subscribers have to be informed. To determine whether the subscribers, including peer B, have to be notified about this modification, the wrapper queries all export schemas in the repository. The modified data or schema items are then pushed actively to the relevant subscribers using a semantically rich representation format. Each data peer is herewith able to maintain an up-to-date replica of the data and schema items required by local applications.

Now the management department has decided to involve an external consulting group D to analyze and optimize the productivity within corporate workflows. Therefore the consultants need access to the entire management data, including the data of departments B and C. Instead of negotiating separate data exchanges with every single department, our architecture enables the consulting group to obtain all data required from only one data source, the management department. This can be realized, since our architecture supports the sharing of data imported from other nodes. Please note, that the export of imported data must explicitly be allowed by the administrator of the management department. After the consulting has subscribed to the entire management data, data updates in B and C are propagated to A as usual. Node A delivers updates on its own data stock and additionally those coming from nodes B and C to its subscriber D. Due to this characteristic, node A becomes a Data Hub for the consulting group according to the Link Pattern Catalog introduced in [95].

We distinguish two different types of update propagation: *direct* and *indirect updates*. After an update is detected on local data of a data source, it is propagated to the relevant subscribers. Referring to our example, B gets direct updates, whenever modifications occur on the data stock of node C. If a node explicitly shares a previously imported data stock, its modifications are in turn propagated to other subscribers. Referring to our example, node A shares previously imported data from peers B and C, which is subscribed by the consulting group node D. If an update occurs on B or C, it is first propagated to node A, which in turn propagates it to its subscriber D. This sequence of update propagation is called a *cascading update*. Further partners may join this collaboration at any time. In fact, each peer can be provided with any data concerning the product launch stored in one of the collaborating data nodes without interfering with existing data flows. The data source maintained by the partner can on the

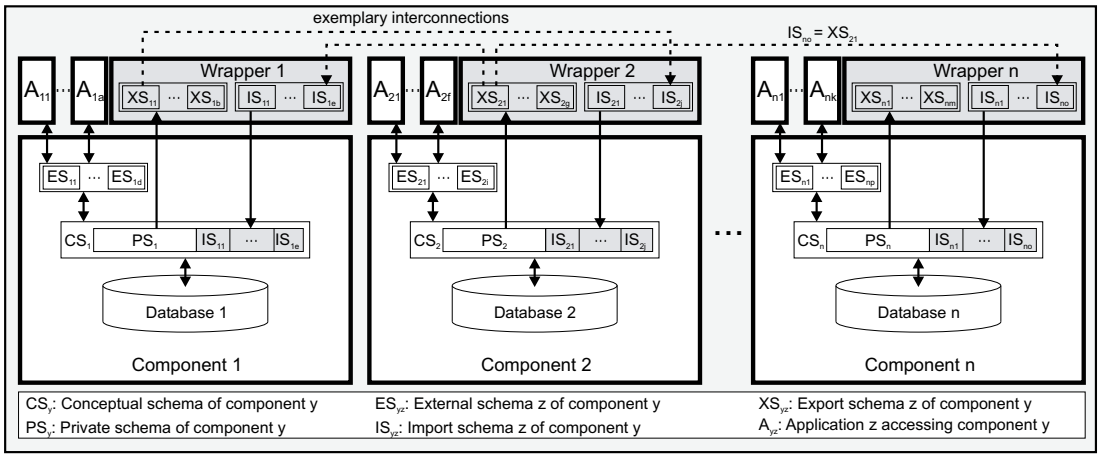

Figure 7.2: DÍGAME Architecture

other hand be easily connected to the existing data grid sharing its own data. If a peer is no longer willing to share its data, it can easily be removed from the data grid, notifying all its subscribers to remove the replicas from their local data stocks. The support of this temporary collaboration makes our DÍGAME architecture particularly suitable for virtual cooperations.

7.3 DÍGAME Architecture Components

In this section we discuss the components of our DÍGAME architecture (Fig. 7.2). The data grid $DG := (P, C)$ created by our architecture is a directed graph, which consists of a set of peers $P := \{p_1, ...p_n\}$ and a set of connections C, where a connection $c = (p_i, p_j) \in C$ links exactly two peers, representing a data flow from p_i to p_j.

As already mentioned, each peer consists of a component database and a corresponding wrapper component. These components which are both required for establishing data flows between communicating peers are described in the following:

Wrapper: The core of our data grid is the wrapper component, which provides a uniform interface to the heterogeneous component databases. It is responsible for negotiating and establishing communication among peers and coordinates the data and schema exchanges after a communication channel has been set up. Each wrapper maintains a repository in its corresponding data source to store information about subscribers, export and import schemas, access control lists, and delivery schedules.

Each wrapper has to realize two major tasks: exporting and importing data and schema items. To export local data from a peer p, a set of export

schemas $\mathcal{XS}_p := \{XS_{p1}, ..., XS_{pi}\}$ is maintained by the wrapper of p. To allow indirect updates, those export schemas have to be based on the entire conceptual schema CS_p of the database, excluding RS_p, the schema of the repository stored on p, i.e. $\forall XS \in \mathcal{XS}_p : XS \subseteq CS_p \setminus RS_p$. They are required to determine, which peers have to be informed about data modifications. To import data from a remote peer p, the wrapper on a peer q ($p \neq q$) maintains a set of import schemas $\mathcal{IS}_q := \{IS_{q1}, ..., IS_{qj}\}$, where

$$\forall q \in P \; \forall IS \in \mathcal{IS}_q \; \exists_1 p \in P \; \exists_1 XS \in \mathcal{XS}_p : IS = XS \wedge p \neq q$$

and

$$\forall p \in P \; \forall XS \in \mathcal{XS}_p \; \exists_1 q \in P \; \exists_1 IS \in \mathcal{IS}_q : XS = IS \wedge q \neq p$$

After the initial import of subscribed data, each data and schema modification propagated by remote peers is reproduced locally in the workspace of the wrapper. Since exporting peers actively propagate the data and schema to relevant subscribers, they must be able to detect modifications on their local data stock and support different data delivery schedules. This functionality is provided by a wrapper with event detection subsystem based on our tightly coupled wrapper architecture. For implementation details of the DÍGAME wrapper component we refer to Section 7.5.

Autonomous Component Databases: According to the Three Schemas Architecture and the architecture for loosely coupled multidatabases [52], each component database on a peer q contains several types of schemas (see Fig. 7.2). The private schema PS_q stores data, which is locally produced and maintained. It is controlled exclusively by the local database administrator. Other peers do not have direct access to this data. Besides the private schema, the conceptual schema CS_q comprises the disjoint union of the import schemas and the repository mentioned above, i.e.

$$CS_q := (\dot{\bigcup}_{IS \in \mathcal{IS}_q} IS) \cup PS_q \cup RS_q \qquad \text{with} \qquad IS \cap PS_q = \emptyset.$$

Local applications $A_{q1}, ..., A_{qf}$ can now access and process the data of the conceptual schema excluding the repository information as usual using a set of external schemas $\mathcal{ES}_q := \{ES_{q1}, ..., ES_{qd}\}$. The only limitation is the read-only access to data derived from the imported schema.

Please keep in mind that the imported data and the repository are exclusively managed by the wrapper component and should never be modified by the local administrator or applications, although this would be possible due to the local autonomy. In fact, future implementations could support such multi-master replication techniques.

7.4 Characteristics

We now discuss the main characteristics of our DÍGAME architecture including the advantages and limitations related to its implementation.

Autonomy and Heterogeneity: Our architecture is based on the concept of loosely coupled multidatabases of Heimbigner and McLeod [52] using import and export schemas for data exchanges. The aim of this architecture is to achieve a feasible trade-off between local autonomy and a reasonable degree of information sharing. A data source is basically free to decide on its own level and form of participation. This includes the ability to decide which data it is willing to export, which data is imported, and during which periods services are provided.Our architecture supports the integration of principally any kind of data source using a wrapper component tailored to that specific data source. The wrapper provides an uniform interface for the DÍGAME system, where communication is performed using a standardized protocol and exchange format.

No Central Authority: Any information sharing environment based on our DÍGAME architecture interconnects autonomous and previously isolated data peers. Each participating data node keeps full control over its own data, i.e. there is no central authority imposing certain restrictions. Contrary to other approaches like [125] we do not use any central component, where publications or subscriptions are managed. In our system, peers subscribe directly to data published by other nodes. The information on the data offered is not managed centrally, but stored exclusively on the corresponding peers.

Wrapper organized similar to P2P systems: We have enhanced the multidatabase architecture with P2P concepts. The wrappers in our architecture interact similar to classical P2P networks. Data exports and imports are exclusively negotiated pairwise, whereas each peer is basically able to interact with any number of data nodes. The entire communication is realized without any central authority, resulting in a network of self-responsible peers, where members are basically able to join or leave at any time.

Replication: The replication of data is one of the main features of our DÍGAME architecture. Data availability is improved in the information grid allowing a data stock to be directly or indirectly replicated over multiple peers. This means, that required data is accessible, even if the original data node is temporary unavailable. Furthermore query performance is increased, since all the required data is stored locally. The refreshment strategies for updating the replicas depend on the application field. We are basically not limited to a single delivery schedule, but able to provide specific replication strategies depending on the needs

of each subscribing peer. Generally, the preferred delivery schedule is an immediate propagation of updates, but other possible delivery schedules can be, but are not limited to periodical or even aggregated propagation. The replication is managed by the wrapper component, which holds information about each subscribing database and its corresponding delivery schedule in its corresponding repository. DÍGAME uses *lazy replication* protocols with one single master and multiple read-only replicas.

Push-based Protocol: A further central characteristic of our architecture is the push-based propagation of data and schema modifications to subscribing peers. At first a data peer subscribes to data offered by a data source, whereupon it receives once a complete copy of the requested data. Afterwards the data source pushes all relevant updates directly to the subscribers according to their specific delivery schedule. This modifications are passed on to further subscribers using indirect updates, until all replicas are updated. Each peer maintaining a replica of remote data is herewith able to access data, which is as up-to-date as possible, even if the original data source is temporary not available. If a replica can not be updated, because a subscriber is currently not reachable, we have decided to include a pull-based fallback mechanism into our architecture. After the communication has been reestablished, the data target can then actively query the data source whether data updates have occurred since their last contact. Thereupon lost updates are propagated once again to the data target.

Standardized Exchange Format: The dynamic interconnectivity of data peers requires a standardized exchange format, suitable for both, data and schema representation. Using knowledge representation techniques we can guarantee that every single data peer understands data and schema updates without explicitly arranging an exchange format. The additional integration of identifiers for data items (e.g. [87]) within the data exchange process simplifies data maintenance, especially if data is imported from multiple sources. This meta information may furthermore be useful for detecting and solving conflicts within the data. We have decided to use the Web Ontology Language OWL as the common representation format for our architecture, since it provides several advantages over classical (semi)structured exchange formats. Based on a meta representation of (relational) databases we can describe the schema of virtually any database. Thereupon the schema representation itself can be used as an OWL ontology, to base the representation of the actual data on. This flexible and powerful technique is only possible due to the possibilities given by *OWL Full* to interpret an instance of a metamodel as a novel ontology. The representation of relational data and schema with the Web Ontology Language OWL entails several advantages over classical (semi)structured exchange formats like XML.

Local Integration: As already mentioned above, each peer may subscribe to multiple data sources. For each subscription it obtains an exact copy of the relevant remote data and schema items. Since we do not have a global schema, the imported data is integrated individually following local integration strategies, which are not provided by our DÍGAME architecture. Having all required data stored in the local database, we are particularly able to associate local and *remote* data with integrity constraints provided by the database, e.g. foreign keys. Furthermore index structures can be created on imported data to optimize data access according to local query requirements.

7.5 Implementation Details

The implementation of the DÍGAME architecture demands for an intelligent and sophisticated wrapper component. Push-based event delivery requires event detection mechanisms to propagate events to interested subscribers. These requirements are perfectly met by our tightly coupled wrapper component presented in Chapter 6. With the event detection subsystem included in the wrapper architecture we are able to react on local events and specifically support real-time update delivery provided by Enhanced Active Databases. Unlike other event-based systems there is no central event broker for event publishing available. Due to the P2P nature of the architecture, each wrapper has to maintain a repository of current subscribers and event delivery queues. A subscriber shall only receive events for which it registered and according to its individual delivery schedule. The implementation of the DÍGAME functionality in the tightly coupled wrapper requires the following extensions:

Subscriber Repository: The wrapper has to implement a repository for subscribers, the events they are interested in, their individual delivery schedule (immediate, periodic), and permissions.

Access Control: The wrapper has to implement a user management to secure access to the data source. Subscribers may have different permissions on different parts of the data and therewith different parts of the update information. For example, consider one of the project databases presented in Table 5.2. One subscriber might be allowed to retrieve updates including information about the project budget B_A, whereas another subscriber must not see the budget information.

Event Processor Update Queue: All events that are detected either by the Event Monitor or Notification Interface are forwarded to the Event Processor. In an environment using asynchronous messages like DÍGAME all events are places in a chronological update queue for further processing.

The Controller sequentially pops the events from the queue and prepares them for propagation to the subscribers.

Subscriber Update Queues: From the list of all updates provided by the Event Processor Queue, the Controller maintains individual update queues for each subscriber according to their specified events of interest and delivery schedule. The subscriber queues are necessary, since the propagation process to the subscribers might fail due to system errors or network breakdowns. An update may only be removed from the subscriber queue, if it was successfully transfered to the target system. Otherwise the transmission has to be repeated.

Client/Server Interface: Since a wrapper in DÍGAME can concurrently send and retrieve messages it requires both, a server and client component in the Mediator Interface to establish communication. The server listens for incoming connections from client components of remote wrappers. Communication channels could be established using sockets, remote procedure calls, or protocols provided by communication frameworks like JXTA [63]. The size of the messages should be optimized using compression algorithms whereas the channels could be secured using public key encryption techniques.

A promising language for the implementation of the DÍGAME wrapper surely is Java. It provides all state-of-the-art solutions for the problems we encounter in this distributed and heterogeneous environment. A wide spectrum of existing libraries and frameworks like JDBC, RMI, JavaBeans, and JXTA allows us to create a robust, flexible, extensible, and most of all portable piece of wrapper software.

7.6 Related Work

The first generation of grid computing emerged in the mid 1990s with the demand for high performance applications, which could not be satisfied by single computers. De Roure et al. [100] divide the evolution of grid computing into three generations: the first generation with its primitive architecture, which tried to distribute computing onto different computers a trivial way. With the second generation of grid computing middleware systems emerged, and finally the current third generation tries to facilitate global collaboration.

Simultaneously some efforts arose to use distributed resources for information retrieval. Although the *Information Grid* of Rao et al. [97] is focused on giving an integrative user interface for distributed information, this approach can be seen as an early forerunner of the so called *Data Grid* of Chervenak et al. [25], a specialization and extension of grid computing. Its intention is to

create an architecture of integrated heterogeneous technologies in a coordinated fashion. Although Chervenak et al. act on the assumption of a heterogeneous conglomerate of data sources, they force the introduction of a centralized metadata repository, e.g. an LDAP directory [107]. This aim is quite catchy especially in a grid consisting of completely autonomous databases changing their schemas frequently. Although we admit that a global metadata repository would simplify many of the challenges, we abstain from that effort of re-centralization, as it causes many difficulties, e.g. every schema change has to be replicated to the global schema directory. The effect is a single point of failure, exactly the opposite of what we wanted to construct. We thus prefer to keep the databases as they are: autonomous, loosely coupled, and without a single point of failure.

With the raise of filesharing systems like Napster or Gnutella [22] the database community started to seriously adopt the idea of P2P Systems to the formerly known loosely coupled database systems. Contrary to the data grid, P2P database systems do not have a global control in form of a global registry, global services, or a global resource management, but multiple databases with overlapping and inconsistent data. These P2P databases resemble heterogeneous and distributed databases, also known as multidatabases [13, 47]. Currently the database community makes a great effort in investigating P2P databases. Worth mentioning is especially the *Piazza* [49] project, where a P2P system is built up with the techniques of the *Semantic Web* [12] with local point-to-point data translations rather than mapping to common mediated schemas or ontologies. Halevy et al. focus on processing and rewriting queries on XML data throughout multiple peers. Contrary to this approach, we deal mainly with relational data and do not have a global schema, since every peer may have its own import-/export-schema combination. As a result every peer has its own integrated schema as basis for queries. Beyond this, Piazza can only deal with data updates as long as the peers are online. As soon as one peer is disconnected from the network data consistency cannot be guaranteed any more. Like in most P2P approaches, peers may not have all the information required for their queries stored locally, so they have to deal with query and result rewriting. This is superfluous in our architecture, since all the data required is cached on that peer. Similar to our approach, Piazza allows only data updates in its origin. Hoschek follows a quite different approach [55], since his goal is to let the loosely coupled databases appear to be a single data source and thus has to deal with distributed query processing. For a more general glimpse on data mappings in P2P systems see [65].

Our strategy allows data to be exchanged among distributed databases connected through a lazy network. This means, that although a running network may not be guaranteed and thus some data broadcasts may be lost, the system heals itself. This challenge resembles the problems known from environments with mobile databases. Current research covers synchronous mobile client synchronization, i.e. data changes are propagated periodically (every t seconds) and not just in time of the data change. Current systems have two main problems

which arise with the synchronous replication: clients have to be contacted every t seconds, no matter if changes have occurred and in the worst-case changes have to be delayed for t seconds. For a more detailed discussion we refer to [51], as most *push-based* technologies base on the idea of *broadcast disks* [2, 3]. In contrast to the broadcast disks, our model ensures that data is only broadcasted to the clients when changes occur, unless the communication between both peers crashes. Hence our approach resembles a *push-based* system with a *pull-based* fallback, similar to [2] with the major difference that our approach is not based on broadcast disks, but on the Observer's Pattern (see below).

There has been much effort in the research of better and more efficient techniques for data propagation, caching, and replication. The evolution of these methods started with early papers like [62] for classical database systems and goes to more recent publications for mobile clients like [8, 9, 51]. For a classification of database replication techniques see [122]. As mentioned above, we have decided to use a push-based replication strategy, which resembles the software engineering's *Observer-Pattern* [39]. This pattern gives us a prototype of how to notify all interested databases about data updates [51]. This communication is only started, if a data update has occurred and a database is interested. In consequence, data broadcasts are minimized.

Following the argumentation in [45] and [25] our model provides only single-master replication, the only guarantor for data stability and clear defined data flows.

Chapter 8

Link Patterns

The development of novel information platforms like DÍGAME, demand for appropriate modeling and description techniques, especially when they resemble P2P concepts. In this chapter we propose the Link Pattern Catalog as a modeling guideline for recurring problems appearing during the design or description of information grids and P2P networks. Link Patterns are represented using the Data Link Modeling Language, a language for describing and modeling virtually any kind of data flows in information sharing environments.

After a short motivation in Section 8.1, we introduce DLML in Section 8.2. The language consists of only a few components and allows the intuitive modeling of data flows in information sharing environments like our DÍGAME architecture. Section 8.3 introduces Link Patterns by giving a description of their structure and a possible classification. Link Patterns are graphically represented in DLML. The Link Pattern Catalog presented in Section 8.4 contains a collection of elementary, data independent, and data sensitive Link Patterns together with a detailed description of a representative of each class. Section 8.5 provides an example for the usage of Link Patterns and DLML, whereas Section 8.6 concludes with an overview of related work. The content of this chapter is based on work originally published in [95].

8.1 Motivation

With the rise of filesharing systems like Napster or Gnutella the database community started to seriously adopt the idea of P2P systems to the formerly known loosely coupled databases. While the original systems were only designed to share simple files among a huge amount of peers, we are not restricted to these data sources any more. New developments allow peers to share virtually any data, no matter if it is originated from a relational, object-oriented, or XML database. In fact, the data may still come from ordinary flat files.

Apparently we have to deal with a very heterogeneous environment of data

sources sharing data, referred to as an information or data grid [25]. If we allow participants to join or leave information grids at any time (e.g. using P2P concepts [21]), we must take a constantly changing constellation of peers into account. Any information grid built up by these peers can either evolve dynamically or be planned beforehand. In both cases we need a concept in order to describe and understand the interactions among the peers involved. Having such a mechanism, we could not only detect single data exchanges, but even model and optimize complex data flows of the entire system.

We adopt commonly used methods for designing data exchanges among peers as *Link Patterns*, suitable especially for information grids and P2P networks. Analogous to the intention of the Design Pattern Catalog used for object-oriented software development [39] we want to provide modeling guidelines for engineers and database designers, engaged in understanding, remodeling, or building up an information grid. Thus information grid architects are provided with a common vocabulary for design and communication purposes. Up to now data flows in information grids were designed without having a formal background leading to individual solutions for a specific problem. These were only known to a circlet of developers involved into that project. Other designers, engaged with a similar problem would never get in contact with these results and thus make the same mistakes again. Different modeling techniques make it difficult to exchange successfully implemented solutions.

Link Patterns do not claim to introduce novel techniques for sharing, accessing, or processing data in shared environments, but a framework for being able to understand, describe, and model their data flows. They provide a description of basic interactions between data sources and operations on the data exchanged, resulting in a catalog of reusable conceptual units. A developer may choose Link Patterns to model and describe complex data flows, to identify a single point of failure, or to avoid or consciously insert redundant data exchanges. The composition of Link Patterns is an essential feature of our design method. It gives us the possibility to represent a structured visualization not only of single data linkages, but of the entire information platform.

8.2 The Data Link Modeling Language (DLML)

8.2.1 Introduction

The Data Link Modeling Language (DLML) is based on the *Unified Modeling Language* (UML) [39] notation, but slightly modifies existing components, adds additional elements, and thus extends its functionality. It is a language for modeling, visualizing, and optimizing virtually any kind of data flows in information sharing environments.

Modeling: DLML is a language, suitable for modeling, planning, and re-engi-

neering data flows in information sharing environments, e.g. information grids, systematically. A Data Link Model built up using this language reflects the logical and not the physical structure of the entire system. It enables the developer to specify the properties and the behavior of existing and novel systems, in order to describe and understand their basic functionalities.

Visualizing: Visualizing data flows is an important assistance in understanding the structure and behavior of an information platform. The impact of ER [91] and UML has proved, that a system is easier to grasp and less error-prone, if a graphical visualization technique is provided, which uses a well-defined set of graphical symbols, understood by a broad community. Especially within the analysis of systems with distributed information, it is favorable to have a method, suitable for drawing up a map of relationships between the participating peers, in order to depict global data flows.

Optimizing: Besides the modeling and visualization of an information sharing environment, DLML can be useful to optimize the whole distributed data management. Redundant data flows and data stocks can systematically be detected and removed, leading to a higher performance of the entire system. Of course, redundancy may explicitly be wanted, in order to achieve a higher fail-safety or a faster access to the data.

Due to the characteristics mentioned above, the Data Link Modeling Language is especially suitable for visualizing data flows in distributed information grids. It may furthermore be employed to model data management in enterprise information systems, data integration and migration scenarios, or data warehouses, i.e. wherever data has to be accessed across multiple different data sources.

8.2.2 DLML Components

Since DLML is based on UML, its diagrams are constructed in an analogous manner, using a well-defined set of building blocks according to specific rules. The following components may be used in DLML (Fig. 8.1) to build up a Data Link Model:

Nodes: Nodes are data sources, data targets, or applications, usually involved in a data exchange process. They may either be isolated or connected through at least one *data flow*. A data source may be a database (e.g. relational), a flat file (e.g. XML), or something similar, offering data, whereas a data target receives data and stores it locally. An application is a software unit, which accesses or generates data, without maintaining an own physical data stock. Physical data stocks are represented in DLML by *Data Nodes*, applications by *Application Nodes*.

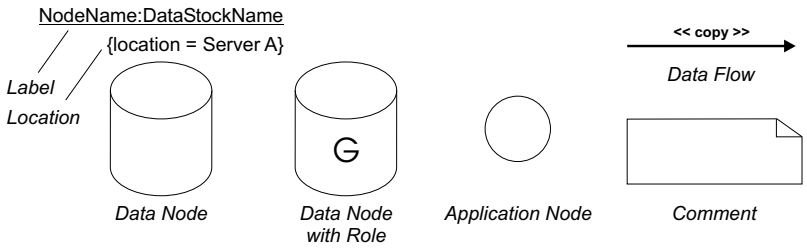

Figure 8.1: DLML Components

Label: Each node can have a label. It consists of generally two parts separated by a colon: the *node name* and the *data stock name* or *application name* respectively. The data stock name identifies the combination of data and schema information stored at this node. If this data is replicated as an exact and complete copy to another node, the data target has to use the same data stock name. The application is identified by the application name. Analogous to the data stock name, any further instances of the same application have the same application name. In both cases we use the node name to distinguish nodes with the same data stock or application name. Otherwise the node name is optional.

Location: The optional location tagged value specifies the physical location of the node. It either specifies an IP address, a server name, or a room number, helping the developer to locate the Data or Application Node.

Role: A node providing a certain functionality on the data processed, may have a functional role (e.g. filtering or integrating data). This role will usually be implemented as a kind of application, operating directly on the incoming or outgoing data. The name of the role or its abbreviation is placed directly inside the symbol of the node. This information is not only useful for increasing the readability of the model, but also for being able to identify complex relationships.

Data Flow: The data exchange between exactly one data source and one data target is called data flow. The arrow symbolizes the direction, in which data is being sent. A node may have multiple incoming and outgoing data flows. Optionally each data flow may be labeled concerning its behavior, i.e. if the data is being replicated (`<<copy>>`) to the data target or if it is just accessed (`<<access>>`). If data is being synchronized, both data flow arrows may be replaced by one single arrow with two arrowheads.

Comment: A comment may be attached to a component, in order to provide additional information about a node or a data flow. These explanations may concern a node's role, filter criteria, implementation hints, data flow

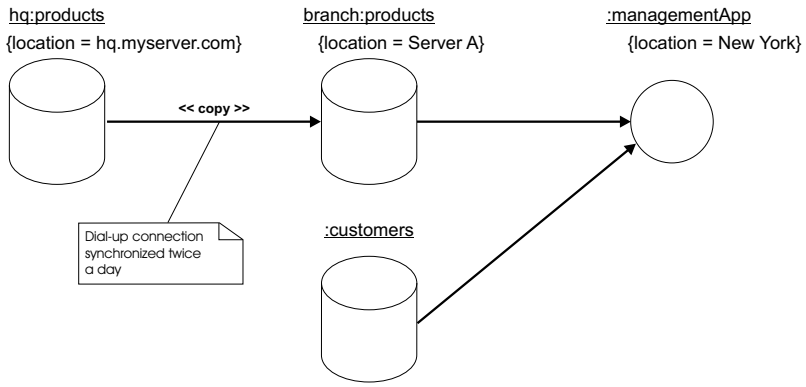

Figure 8.2: DLML Example

properties, or further annotations important for the comprehension of the model.

8.2.3 Example

We now illustrate the usage of the Data Link Modeling Language with a simplified example. Consider a worldwide operating wholesaler, with an autonomous overseas branch. The headquarters is responsible for maintaining the product catalog (`hq:products`) with its price list, while the customers database (`:customers`) is administrated by the branch itself (Fig. 8.2).

The overseas branch is connected to the headquarters by a dial-up connection, not sufficient for accessing the database permanently. For this reason, the product catalog is replicated to the branch twice a day (`branch:products`), where the data may be accessed by the local employees. The branch management uses a special application (`:managementApp`) to access both data stocks in order to generate the annual report for the headquarters.

8.3 Link Patterns

In order to be able to provide a catalog of essential Link Patterns it is necessary to understand what a Link Pattern is. Therefore we present the elements a Link Pattern is composed of, including its name, its classification, or its description. For graphical representation we use the Data Link Modeling Language, specified above.

8.3.1 Elements of a Link Pattern

In this section we present the description of the Link Pattern structure. It is based on the Design Pattern Catalog from Gamma et al. [39], which has reached great acceptance within the software engineering community. Thus a developer is able to quickly understand and adopt the main concept of each Link Pattern for his own purposes. Each Link Pattern is described by the following elements:

Name: The name of a Link Pattern is its unique identifier. It has to give a first hint on how the pattern should be used. The name is substantial for the communication between or within groups of developers.

Classification: A Link Pattern is classified according to the categories described in Section 8.3.2. The classification organizes existing and future patterns depending on their functionality.

Motivation: Motivating the usage of the pattern is very important, since it explains the developer figuratively the basic functionality. This is done using a small scenario, which illustrates a possible application field of the pattern. Therewith the developer is able to understand and follow the more detailed descriptions in the further sections.

Graphical Representation: The most important part of the pattern description is the graphical representation. It is a DLML diagram and describes the composition and intention of the pattern in an intuitive way. The developer is advised to adopt this representation, wherever he has identified the related functionality in his own information grid model.

Description: The composition of the Link Pattern is described in-depth in this section, including every single component and its detailed functionality. The explanation of the local operations on each node and data flows between the components involved, points up the intended functionality of the whole pattern described. This description shall give the user both, a guidance through the identification process and instructions for its proper usage.

Challenges: Besides the general instructions given in the prior section, this section shall give hints for sources of error in the implementation process of this pattern. The developer shall get ideas, of how to identify and avoid pitfalls, arising in a certain context (e.g. interaction with other Link Patterns).

8.3.2 Classification

A classification of the Link Pattern Catalog shall provide an organized access to all Link Patterns presented. Patterns situated in the same class have similar

structural or functional properties, depending on the complexity of their implementation. Although a categorization of a very limited number of patterns may seem superfluous, we have decided to include this into our Link Pattern Catalog, since it may help developers to allocate and evaluate the pattern required. Furthermore it should stimulate the developer to find and rate novel patterns, not yet included in the catalog.

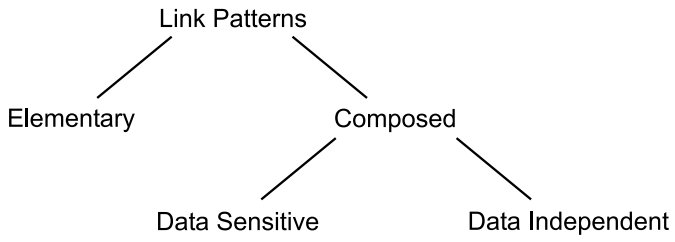

Figure 8.3: Link Pattern Catalog Classification

Figure 8.3 depicts the classification of our Link Pattern Catalog we have chosen. The patterns presented can be divided into two main categories, *Elementary Link Patterns* and *Composed Link Patterns*. In fact this classification is not completed, but shall provide a starting point for further extension.

Elementary Link Pattern

An Elementary Link Pattern is the smallest unit for building up an information grid model. It consists of exclusively one single node and at least one data flow connected to it. Each Data Link Model is composed of several Elementary Link Patterns, linked together with data flows in an appropriate way. Please note, that a single Elementary Link Pattern is not yet a reasonable Data Link Model, since any data flow must have at least one node offering data and one node receiving data.

Elementary Link Patterns are easy to understand and easy to implement, since they concern only a single node, a small set of data flows, and do not include basically any data processing logic. It must be pointed out, that the Elementary Link Patterns consist only of two main patterns, the *Basic Data Node* and the *Basic Application Node*, and its derivatives (e.g. Publisher and Generator, discussed in section 8.4).

Composed Link Pattern

Composed Link Patterns are built up by combining at least two Elementary Link patterns in a specific way, in order to realize a particular functionality. A Composed Link Pattern may hereby be composed out of both, Elementary or other Composed Link Patterns. A pattern has to represent a prototype or

solution for a recurring sort of problem. Please keep in mind, that an arbitrary combination of different patterns will not automatically lead to a reasonable Composed Link Pattern. In contrast to the Elementary Link Patterns, we have to deal in this context with a more complex kind of patterns. They do not only include more nodes, but may even represent a quite sophisticated way of linking them. Besides, each node may additionally process the data received or sent. The fact, that it may act differently depending on the data involved, is an essential property of Composed Link Patterns and justifies the creation of two subclasses:

Data Sensitive Link Pattern: As soon as a node included in a Composed Link Pattern acts depending on the data it processes, the entire pattern is called a Data Sensitive Link Pattern. This data processing logic implemented on such a node may depend on and be applied to incoming and/or outgoing data. The operations of this application can either create, alter, or filter data.

Data Independent Link Pattern: Any Composed Link Pattern, not classified as Data Sensitive, belongs to this class. In contrast to the patterns described above, data is not being modified, but sent or received as is. A rather crucial topic is the topology of the nodes and data flows involved, which is most relevant for the creation and functionality of this kind of patterns.

8.3.3 Usage

This section describes how Link Patterns can be useful to develop, maintain, analyze, or optimize both, straightforward and complex data flows in information grids. There are basically two methods, how Link Patterns can improve the work of developers:

Analyzing existing systems: Many existing information grids have arisen during the years without being planned centrally or consistently. Even if they were planned initially, they usually tend to spread in an uncontrolled way. In such an environment it is vital to have supporting tools, helping to understand and later optimize an existing system.

First of all a map or model of the existing system has to be created, e.g. with DLML presented in Section 8.2. Afterwards we examine successively smaller parts of the model, in order to match them to existing Link Patterns of the Catalog. As a result we get a revised model containing basic information on the composition and functionality of subsystems, including their data processing and data flows. With this information in mind, we are now able to derive information on data flows and interaction of nodes inside the Data Link Model. This enables us to perform optimizations like detecting and eliminating vulnerabilities or handling redundancies.

Link Patterns may thus not replace human expertise for understanding existing information grids, but give support in the process of recognizing global data flows and therewith interpret the purpose of the entire system.

Composing new models: As already mentioned a Link Pattern may not only improve the process of understanding an existing information grid, but is also a support for modeling new systems. An information grid architect needs to have a clear idea of what the system should do. Depending on the data sources available, the local requirements on the nodes, and the results he wants to achieve, he can combine nodes and data flows, according to Link Patterns, until the entire system realizes the intended functionality. Link Patterns hereby guarantee a common language, understood by other developers, not yet involved in the modeling. Each developer is thus able to quickly get a general idea of the system modeled at any time. Furthermore they accelerate the development process, since they provide well tried solutions for recurring problems, leading to an efficient system of high quality.

8.4 Link Pattern Catalog

In this section we finally give an introduction into the Link Pattern Catalog. This includes a graphical overview over the main Link Patterns in DLML, as well as a detailed description of selected patterns. As mentioned beforehand the Link Patterns can be classified according to the classification presented in section 8.3.2. Since any Composed Link Pattern either belongs to the Data Sensitive or to the Data Independent Link Patterns, we organize the catalog as follows:

Elementary Link Patterns

The Elementary Link Patterns are the basic building blocks of a Data Link Model. They consist of the two basic patterns, described below, and its derivatives. All Elementary Link Patterns are depicted in Figure 8.4.

Basic Data Node

Classification: Elementary Link Pattern

Motivation: This pattern is one of the basic building blocks of a Data Link Model. Each incoming or outgoing data flow of a Data Node is modeled using this Link Pattern.

Graphical Representation: See Figure 8.4

Description: A Basic Data Node is a DLML Data Node, which receives data through incoming data flows, stores it locally, and simultaneously propagates data, held in its own data stock. If a Basic Data Node does only have outgoing or incoming data flows, it applies the *Publisher Pattern* or the *Subscriber Pattern* respectively. If it does neither have any incoming, nor any outgoing data flows, the Data Node is called *isolated*.

Challenges: One of the main challenges to take in this pattern is the proper coordination of incoming and outgoing data flows. At first all incoming data has to be stored permanently on the local data stock, without violating any constraints, before it may be propagated again to other nodes.

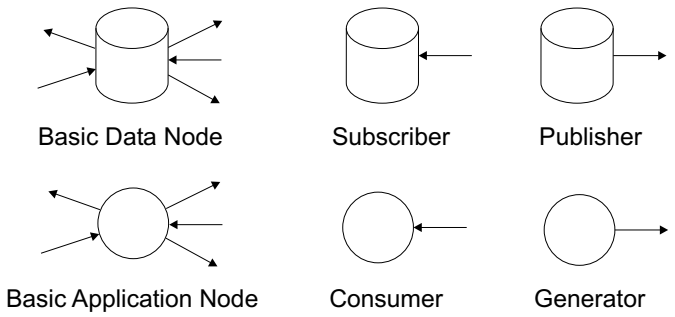

Figure 8.4: Elementary Link Patterns

Basic Application Node

Classification: Elementary Link Pattern

Motivation: This pattern is one of the basic building blocks of a Data Link Model. All applications, relevant for a Data Link Model, are based on this pattern.

Graphical Representation: See Figure 8.4

Description: An application interacting with arbitrary Data or Application Nodes, is represented by this pattern. The application does not only receive, but also propagate data. If a Basic Application Node does only have outgoing or incoming data flows, it applies the *Generator Pattern* or the *Consumer Pattern* respectively. If it does neither have any incoming nor any outgoing data flows, the Application Node is called *isolated*.

Challenges: Propagated data can either be received or generated. All data manipulations on incoming data, which have to be propagated, have to be processed in real-time, without storing data locally.

Data Independent Link Patterns

The Data Independent Link Patterns belong to the Composed Link Patterns. These patterns describe a functionality, which only depends on their structure, i.e. the way nodes and data flows are combined. A graphical overview of the patterns in this class is given in Figure 8.5, of which the *Data Backbone* is described exemplarily.

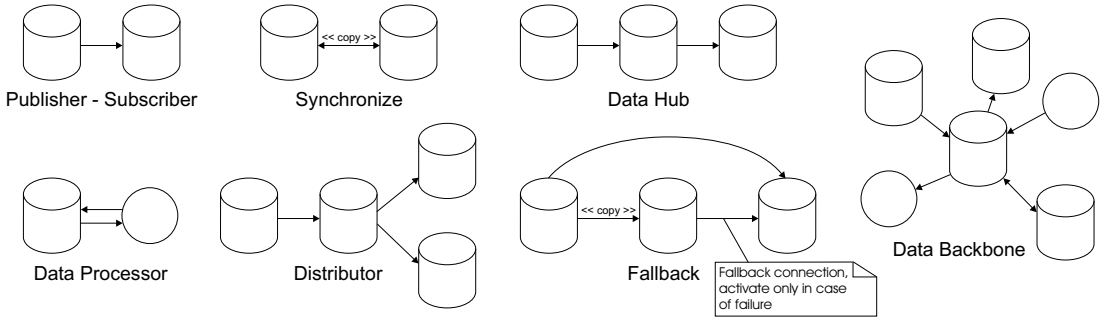

Figure 8.5: Data Independent Link Patterns

Data Backbone

Classification: Data Independent Link Pattern

Motivation: A Data Backbone is used, wherever a centralization of data sharing or data access has to be realized. This is typically required, if data stocks are re-centralized, a central authority wants to keep track on all data flows, or data exchanges have to be established among multiple data stocks and applications.

Graphical Representation: See Figure 8.5

Description: The Data Backbone Pattern consists of several nodes, linked together in a specific way. A designated node, called Data Backbone, is either data source or data target for all data flows in this pattern. All nodes, including the Data Backbone itself, can be data stocks or applications. Data is always propagated from data sources to the Data Backbone, where it may be accessed or propagated once again to other target nodes. Direct data flows between nodes, which are not the Data Backbone, are avoided.

Challenges: Since the Data Backbone is involved in all data flows, it has a crucial position in this part of the information grid. Thus, a Data Backbone node has to provide a high quality of service, concerning disk space, network

connection, and processing performance. If the quality of service required cannot be provided, the Data Backbone may easily become a bottleneck. Furthermore a breakdown of this node could lead to a collapse of the entire data sharing infrastructure, which makes it to a single point of failure.

Data Sensitive Link Patterns

Contrary to the Data Independent Link Patterns, the patterns described in this section are not only classified according to their structural properties, but particularly because of their data processing functionality. A graphical representation of these Data Sensitive Link Patterns can be found in Figure 8.6, while a detailed description is only given for the *Gatekeeper Pattern*.

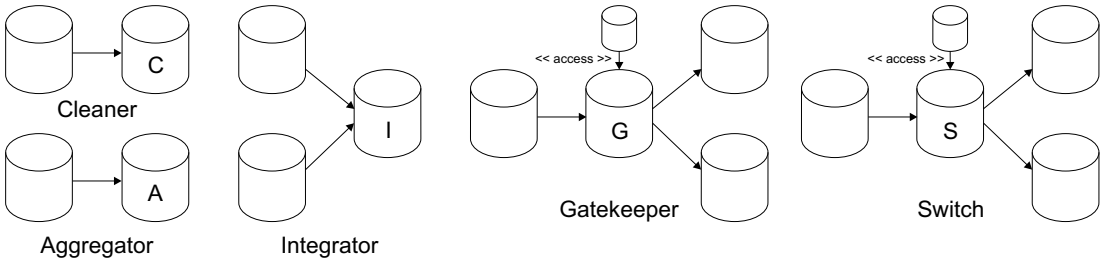

Figure 8.6: Data Sensitive Link Patterns

Gatekeeper

Classification: Data Sensitive Link Pattern

Motivation: A Gatekeeper is used to control data flows according to specific rules (e.g. Access Control Lists), stored separately from the data processed. It is responsible for providing the target nodes with the accessible data required. The application of this pattern is not limited to data security matters. It may actually be applied to any node, which has to supply different target nodes with specific (e.g. manipulated or filtered) data flows.

Graphical Representation: See Figure 8.6

Description: A Gatekeeper is a designated node, which distributes data according to specific rules, eventually stored separately. Local or incoming data of a Gatekeeper is accessed by target nodes. Before this access can be admitted, the Gatekeeper has to check the permissions. Thus, corresponding to the rules processed, neither all data stored in the Gatekeeper, nor all data requested by the target nodes has to be transmitted.

Challenges: The rules and techniques, which are used by the Gatekeeper in order to secure access to the data, have to be robust and safe. The Gatekeeper needs a mechanism to identify and authenticate the target nodes (e.g. IP address, public key, or username and password), which may be stored in a separated data stock. Due to its vital position in the exchange process, this information has to be protected from unauthorized access. The Gatekeeper must be able to rely on the correctness, authenticity, and availability of the rules required.

8.5 Example

This section provides an example of how to model a new information grid of a worldwide operating company. The headquarters of the company are located in New York. It has additionally branches in Düsseldorf (head office of the European branches), Paris, Bangalore, and Hong Kong. Each branch maintains its own database containing sales figures, collected by local applications. For backup and subsequent data analysis, this data has to be replicated to the headquarters. Additionally, the Düsseldorf branch needs to be informed about the ongoing sales activities of the Paris branch. To simplify the centralized backup, the company has decided to forbid any data exchanges between the single branches.

The central component of this infrastructure is the backup system in New York. It collects the sales data from all branches, without integrating them. Additionally it provides the Düsseldorf branch with all the information required from Paris. Since the headquarters in New York want to analyze the entire data stock of the company, a data warehouse, based on the data of the backup system, is set up. Having a certain local autonomy, the data provided by the European branches and the remaining branches have some structural differences. For this reason, the data has to be integrated prior to the aggregation required for the data warehousing analysis. Using the Link Patterns, we are now able to model the enterprise information grid as depicted in Figure 8.7.

The local applications, which maintain the local sales databases, are modeled using the *Data Processor*. This data is replicated to the backup system in New York, realized as a *Gatekeeper*. It thus controls the data flows from the branches to the data warehouse and to the Düsseldorf branch. It must be guaranteed, that the data targets get only their designated data, i.e. neither data from Bangalore, nor from Hong Kong is accessible for the European head office in Düsseldorf. The data warehouse is realized by a node, which integrates several data sources using common integration strategies (*Integrator Pattern*) and aggregates the data afterwards (*Aggregator Pattern*), in order to provide OLAP applications with a homogeneous data stock. Please keep in mind, that the Data Link Model presented in Figure 8.7 reflects the logical structure of the information platform, not the physical. This means, that the nodes of the model do not have to be

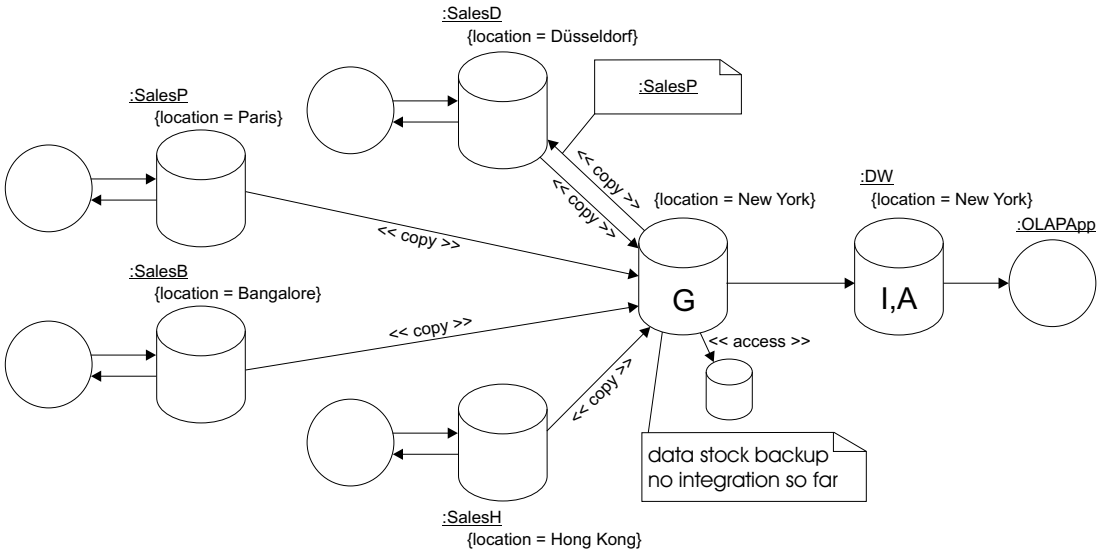

Figure 8.7: Example using Link Patterns

located on different machines.

8.6 Related Work

Data Flow analysis and modeling has been a focus of researchers for decades. Earlier work concentrates mainly on data flows in computer architectures and software components (e.g. [123, 26]). Later on, data flows were also used for query processing and optimization in database systems. For instance, Teeuw and Blanken [114] compare control versus data flow mechanisms controlling the execution of database queries on parallel database systems. Dennis and Misunas present in [32] a Basic Data-Flow language, which expresses graphically the data dependencies within a program. In this data flow graph model, instructions are represented by nodes and paths stand for data or control flows. Although this language was originally designed for software development, it may be seen as an early forerunner, in designing data flows among different data sources. A specialized data flow graph is introduced by Eich and Wells [35], which can be used for scheduling database queries within multiprocessor environments or databases distributed over a network [15]. Thus, both approaches apply data flow concepts to database processing.

The Link Patterns are tightly coupled to the Design Patterns of the object-oriented software design [39, 19] and Enterprise Application Integration (EAI) [54], since they represent prototypes or solutions for recurring problems. Contrary to these patterns, Link Patterns are not intended to solve recurring problems in software design or EAI, but to provide modeling and description guidelines for

information grids, focusing exclusively on data flows. As a possible application field of our Link Patterns we suggest modeling or visualizing information grids, i.e. heterogeneous environment of data sources sharing data or modern information infrastructures, based on P2P concepts like DÍGAME (described in Chapter 7) or Piazza [49].

Chapter 9

Conclusion and Future Work

9.1 Summary

The ability of modern active database systems to execute external programs written in a standalone programming language from within its DBMS offers new perspectives to data management in federated information systems. We have defined Enhanced Active Databases as a new subclass of active databases that provide external program calls as an enhanced active functionality. Databases of this activity class are thus able to interact with external software or hardware components, whereas communication is established using client/server socket connections, remote procedure calls, or database connections provided by the external programming language. In the context of this work we have focused on external programs that are executed from within triggers as a reaction to an update event occurring in the database. Besides the definitions and notations used throughout this thesis and a detailed description of of the execution of external routines, we have introduced remote state queries, injected transactions, and external notifications as the enhanced functionalities required for our concepts.

We have proposed several concepts and architectures that are specifically developed for Enhanced Active Databases to support information sharing in the autonomous and heterogeneous environment of federated information systems. We have shown that an Enhanced Active Database that participates in a federation as a component database can contribute to the interoperability and consistency in federated architectures.

As a basic functionality we have introduced Active Event Notifications that enable EADBSs to actively signal changes in their data stock to external components like an event broker or constraint manager. We are thus able to implement a truly immediate change delivery using a push-based delivery mode and synchronous messages. A local update transaction is blocked until the notification process has finished which makes this feature unique for Enhanced Active Databases.

One application field of synchronous update notifications is global integrity maintenance in federated information systems. We have introduced the notion of Active Component Database Systems based on EADBS which are able to communicate with other component databases to which their data is semantically related to. They are no longer just passive data providers but actively participating in global integrity maintenance. Global integrity constraints are composed of sets of partial integrity constraints for each component database that is affected by the constraint. The partial constraints are evaluated using local and remote checks which are implemented entirely on a local site. We have described the requirements and basic functionality of our architecture and provided examples for partial constraints for commonly used classes of global constraints. As an extension to this concept we have proposed an external constraint manager that is synchronously notified of updates by the component databases and performs the partial constraint checks instead.

To support Active Event Notifications for general processing in a federated information system, we have extended a wrapper component with an event detection subsystem. The wrapper architecture particularly supports Active Event Notifications from EADBS which makes him tightly coupled to the encapsulated database. Furthermore, it comprises components required for monitoring other types of data sources like passive databases or flat files. This makes the wrapper perfectly suitable for event-based federated systems with real-time data processing.

Based on our tightly coupled wrapper components we have presented an architecture for a P2P-based information system. Our DÍGAME architecture enhances the well-known multidatabase architecture with P2P concepts, in order to support dynamic intra- and inter-enterprise collaboration. Local administrators decide themselves on their level of participation, since the local autonomy is preserved. Data provided by other peers can be subscribed and integrated into the local database as needed. The data source actively propagates changes on the subscribed data and schema items to the relevant peers via a standardized exchange format resulting in a replication of the data demanded locally. Peers participating in the data grid interact pairwise without being managed by any central authority.

Finally, we have presented Link Patterns as guidelines for modeling and describing data flows between nodes in information sharing environments like DÍGAME. The Link Pattern Catalog consists of prototypes or solutions for recurring problems and therewith supports developers to model, describe, and understand complex information grids. Furthermore the Link Patterns provide a common vocabulary for design and communication purposes, enabling developers to exchange successfully implemented solutions. Additionally we have introduced the Data Link Modeling Language (DLML) for modeling, visualizing, and optimizing data flows, especially suitable for information grids. This language based on UML consists of a well-defined set of building blocks, representing data nodes,

application nodes and data flows between them. They can be combined according to specific rules, to build up the Data Link Model of an information sharing environment.

9.2 Future Work

In the context of this thesis we have focused on only a small set of possible enhanced functionalities to improve interoperability in federated information systems. We believe that the full potential of Enhanced Active Databases, that enables a tight coupling of database layer and application layer, has still to be released. Using external program calls, we are able to use the entire spectrum of the external programming language to establish communication channels and manipulate remote data. However, referring to the concepts we have presented in this work, there are several aspects that would require further investigations.

The push-based synchronous update notification mechanism has some limitations that need further considerations. Depending on the database we could develop a mechanism that fully exploits DBMS-specific capabilities to push updates to external components. The main problem herewith is the access to the delta sets from within the triggers. Possible solutions to this problem could pass the entire delta set as a parameter to the external routine.

Another problem is the usage of injected transactions during partial referential integrity checks. We have to cope with the problem of atomic commit to ensure a consistent global state. This could be realized using an external transaction manager similar to the Constraint Manager in COMICS that implements centralized transaction management according to the redo, retry, or compensate approach.

The development of tools to assist administrators and users to create, deploy, and monitor external programs for federated information systems would be another contribution. Furthermore, the concepts have to be tested for new EADBSs that will surely emerge in the near future.

The Link Patterns we have presented are ideal to generate a static model of data and application nodes with their corresponding data flows. Future work could consider dynamically changing and evolving environments, in which nodes constantly join or leave the grid. This may not only affect the Link Pattern Catalog, but also the Data Link Modeling Language. Furthermore the Catalog has to be enhanced, in order to include novel Link Patterns, not yet identified. The entire Link Pattern Catalog shall provide developers with an extensive reference guideline for modeling information sharing environments.

Bibliography

[1] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and Updating the File. In *Proceedings of the 19th International Conference on Very Large Databases*, pages 73–84. Morgan Kaufmann, 1993.

[2] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing Push and Pull for Data Broadcast. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 183–194. ACM Press, 1997.

[3] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Disseminating Updates on Broadcast Disks. In *Proceedings of 22th International Conference on Very Large Data Bases*, pages 354–365. Morgan Kaufmann, 1996.

[4] Maria Sueli Almeida, Kirk Condon, Michael Fischer, and Julian Stuhler. DB2 Java Stored Procedures - Learning by Example. http://ibm.com/redbooks, 2000.

[5] Rafael Alonso and Daniel Barbará. Negotiating Data Access in Federated Database Systems. In *Proceedings of the 5th International Conference on Data Engineering*, pages 56–65. IEEE Computer Society, 1989.

[6] ANSI/ISO/IEC 9075-2 Information Technology - Database Language SQL - Part 2: Foundation (SQL/Foundation), 1999.

[7] Lance Ashdown. Oracle 10g - Application Developer's Guide - Fundamentals. Oracle Press, 2005.

[8] Daniel Barbará. Mobile Computing and Databases - A Survey. *Transactions on Knowledge and Data Engineering*, 11(1):108–117, 1999.

[9] Daniel Barbará and Tomasz Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 1994.

[10] Daniel Barbará-Millá and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, 1994.

[11] Hernando Bedoya, Daniel Lema, Cintia Marques, and Vijay Marwaha. Stored Procedures and Triggers. http://ibm.com/redbooks, 2001.

[12] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284:35–43, 2001.

[13] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Proceedings of the 5th International Workshop on the Web and Databases*, pages 89–94, 2002.

[14] Indrajit Bhattacharya and Lise Getoor. Relational Clustering for Multi-type Entity Resolution. In *Proceedings of the 4th International Workshop on Multi-relational Mining*, pages 3–12. ACM Press, 2005.

[15] Lubomir Bic and Robert L. Hartmann. AGM: A Dataflow Database Machine. *ACM Transactions on Database Systems*, 14(1):114–146, 1989.

[16] Kenmore S. Brathwaite. Resolution of Conflicts in Data Ownership and Sharing in a Corporate Environment. *ACM SIGBDP Data Base*, 15(1):37–42, 1983.

[17] Yuri Breitbart, Hector Garcia-Molina, and Abraham Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 1(2):181–240, 1992.

[18] M. W. Bright, Ali R. Hurson, and Simin H. Pakzad. Automated Resolution of Semantic Heterogeneity in Multidatabases. *ACM Transactions on Database Systems*, 19(2):212–253, 1994.

[19] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., 1996.

[20] Susanne Busse, Ralf-Detlef Kutsche, Ulf Leser, and Herbert Weber. Federated Information Systems: Concepts, Terminology and Architectures: Forschungsbericht des Fachbereichs Informatik 99-9. Technical report, TU Berlin, Fachbereich 13 Informatik, 1999.

[21] Mario Cannataro and Domenico Talia. Semantics and Knowledge Grids: Building the Next-Generation Grid. *IEEE Intelligent Systems*, 19(1):56–63, 2004.

[22] Bengt Carlsson and Rune Gustavsson. The Rise and Fall of Napster - An Evolutionary Approach. In *Proceedings of the 6th International Computer Science Conference - Active Media Technology*, pages 347–354. Springer, 2001.

[23] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful Change Detection In Structured Data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 26–37. ACM Press, 1997.

[24] Liangyou Chen, Hasan M. Jamil, and Nan Wang. Automatic Composite Wrapper Generation for Semi-structured Biological Data based on Table Structure Identification. *SIGMOD Record*, 33(2):58–64, 2004.

[25] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.

[26] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, 1989.

[27] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering*, pages 41–52. IEEE Computer Society, 2002.

[28] COBOL ACCESS. www.rldt.fr, 2006.

[29] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Supporting Multiple View Maintenance Policies. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 405–416. ACM Press, 1997.

[30] Stefan Conrad. *Föderierte Datenbanksysteme: Konzepte der Datenintegration*. Springer, Berlin, 1997.

[31] Stefan Conrad, Ingo Schmitt, and Can Türker. Dealing with Integrity Constraints During Schema Integration. In *Proceedings of the International Workshop of Engineering Federated Database Systems*, pages 13–22. Otto-von-Guericke-Universität Magdeburg, 1997.

[32] Jack B. Dennis and David P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132. ACM Press, 1975.

[33] Weimin Du, Ahmed K. Elmagarmid, and Won Kim. Effects of Local Autonomy on Heterogeneous Distributed Database Systems. Technical Report ACT-OODS-EI-059-90, Microelectronics and Computer Technology Corp., 1990.

[34] Yann Dupont. Resolving Fragmentation Conflicts in Schema Integration. In *Proceedings of the 13th International Conference on the Entity-Relationship Approach*, pages 513–532. Springer, 1994.

[35] Margaret H. Eich and David L. Wells. Database Concurrency Control Using Data Flow Graphs. *ACM Transactions Database Systems*, 13(2):197–227, 1988.

[36] D. Fang, J. Hammer, and D. McLeod. The Identification and Resolution of Semantic Heterogeneity in Multidatabase Systems. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 136–143. IEEE Computer Society, 1994.

[37] I.P. Fellegi and A. B. Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association*, 64:1183–1210, 1969.

[38] Michael Franklin and Stan Zdonik. "Data in your face": Push Technology in Perspective. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 516–519. ACM Press, 1998.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[40] Hector Garcia-Molina and Boris Kogan. Node Autonomy in Distributed Systems. In *Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems*, pages 158–166. IEEE Computer Society, 1988.

[41] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[42] Manuel García-Solaco, Felix Saltor, and Malu Castellanos. A Structure Based Schema Integration Methodology. In *Proceedings of the 11th International Conference on Data Engineering*. IEEE Computer Society, 1995.

[43] Gartner Press Release May 2005. http://www.gartner.com/press_releases/asset_127553_11.html, 2005.

[44] Lorena G. Gomez. *An Active Approach to Constraint Maintenance In A Multidatabase Environment*. PhD thesis, Arizona State University, 2002.

[45] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, 1996. ACM Press.

[46] Paul W. P. J. Grefen and Jennifer Widom. Integrity Constraint Checking in Federated Databases. In *Proceedings of the International Conference on Cooperative Information Systems*, pages 38–47. IEEE Computer Society, 1996.

[47] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What Can Databases Do for Peer-to-Peer? In *Proceedings of the 4th International Workshop on the Web and Databases*, 2001.

[48] Lifang Gu, Rohan A. Baxter, Deanne Vickers, and Chris Rainsford. Record Linkage: Current Practice and Future Directions. Technical Report 03/83, CSIRO Mathematical and Information Sciences, Canberra, Australia, 2003.

[49] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. In *Proceedings of the 12th International Conference on World Wide Web*, pages 556–567, 2003.

[50] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema Mediation in Peer Data Management Systems. In *Proceedings of the International Conference on Data Engineering*, pages 505–517, 2003.

[51] Takahiro Hara. Cooperative Caching by Mobile Clients in Push-based Information Systems. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 186–193, 2002.

[52] Dennis Heimbigner and Dennis McLeod. A Federated Architecture for Information Management. *ACM Transactions on Information Systems (TOIS)*, 3(3):253–278, 1985.

[53] Ludmila Himmelspach. Portierbarkeit von Partiellen Integritätsbedingungen. Bachelor's Thesis. Heinrich-Heine-Universität Düsseldorf, 2005.

[54] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[55] Wolfgang Hoschek. A Unified Peer-to-Peer Database Framework for Scalable Service and Resource Discovery. In *Proceedings of the Third International Workshop on Grid Computing*, pages 126–144, 2002.

[56] Cheng Hsu and Laurie Rattner. Metadatabase Solutions for Enterprise Information Integration Problems. *ACM SIGBDP Data Base*, 24(1):23–35, 1993.

[57] James W. Hunt and Thomas G. Szymanski. A fast Algorithm for Computing Longest Common Subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[58] IBM DB2 Online Manuals: Introduction to Replication and Event Publishing. http://www.ibm.com, 2006.

[59] IBM Homepage. http://www.ibm.com, 2006.

[60] IBM Software - Informix Product Familiy. www.ibm.com/software/data/informix, 2006.

[61] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[62] Paul R. Johnson and Robert H. Thomas. RFC 677: Maintenance of Duplicate Databases. http://www.rfc-archive.org, 1975.

[63] JXTA Project Homepage. http://www.jxta.org, 2006.

[64] Vasiliki Kantere, John Mylopoulos, and Iluju Kiringa. A Distributed Rule Mechanism for Multidatabase Systems. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 56–73. Springer, 2003.

[65] Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 325–336. ACM Press, 2003.

[66] Arne Koschel and Ralf Kramer. Configurable Event Triggered Services for CORBA-based Systems. In *Proceedings of the 2nd International Enterprise Distributed Object Computing Workshop*, pages 306–318, 1998.

[67] Thomas Kudrass, Andreas Loew, and Alejandro P. Buchmann. Active Object-Relational Mediators. In *Proceedings of the International Conference of Cooperative Information Systems*, pages 228–239, 1996.

[68] Krasimir Kutsarov. Entwicklung eines Wrapper-Teilsystems zur Erkennung von Ereignissen in passiven Datenquellen. Bachelor's Thesis. Heinrich-Heine-Universität Düsseldorf, 2005.

[69] Wilburt Labio and Hector Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 63–74. Morgan Kaufmann, 1996.

[70] Justin Langseth. Real-time Data Warehousing: Challenges and Solutions. http://dssresources.com/papers/features/langseth/langseth02082004.html, 2004.

[71] Ki Yong Lee and Myoung Ho Kim. Optimizing the Incremental Maintenance of Multiple Join Views. In *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP*, pages 107–113. ACM Press, 2005.

[72] Christoph Liebig, Marco Malva, and Alejandro P. Buchmann. Integrating Notifications and Transactions: Concepts and X^2TS Prototype. In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects*, pages 194–214. Springer, 2000.

[73] Kevin Loney and Bob Bryla. *Oracle Database 10g, DBA Handbook*. McGraw-Hill/Osborne, 2005.

[74] Kevin Loney and George Koch. *Oracle8i: The Complete Reference*. Osborne/McGraw-Hill, 2000.

[75] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic Schema Matching with Cupid. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 49–58, 2001.

[76] MSDN Library. www.microsoft.com/germany/msdn, 2006.

[77] James G. Mullen, Ahmed K. Elmagarmid, Won Kim, and Jamshid Sharif-Askary. On the Impossibility of Atomic Commitment in Multidatabase Systems. In *Proceedings of the 2nd International Conference on System Integration*, pages 625–634. IEEE Computer Society, 1992.

[78] Craig S. Mullins. *DB2 Developer's Guide*. Sams, 2004.

[79] Peter Muth and Thomas C. Rakow. Atomic Commitment for Integrated Database Systems. In *Proceedings of the 7th International Conference on Data Engineering*, pages 296–304. IEEE Computer Society, 1991.

[80] Wolfgang Nejdl, Wolf Siberski, and Michael Sintek. Design Issues and Challenges for RDF- and Schema-based Peer-to-peer Systems. *SIGMOD Record*, 32(3):41–46, 2003.

[81] Tho Manh Nguyen and A. Min Tjoa. Zero-Latency Data Warehousing for Heterogeneous Data Sources and Continuous Data Streams. In *Proceedings of the 5th International Conference on Information Integration and Web-based Applications Services*, volume 170 of *books@ocg.at*. Austrian Computer Society, 2003.

[82] Oracle Homepage. http://www.oracle.com, 2006.

[83] Oracle Technology Network. http://otn.oracle.com, 2006.

[84] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[85] Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, and Jeffrey D. Ullman. A Query Translation Scheme for Rapid Implementation of Wrappers. In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*, pages 319–344. Springer, 1995.

[86] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999.

[87] Cristian Pérez de Laborda and Stefan Conrad. A Semantic Web based Identification Mechanism for Databases. In *Proceedings of the 10th International Workshop on Knowledge Representation meets Databases (KRDB 2003), Hamburg, Germany, September 15-16, 2003*, volume 79 of *CEUR Workshop Proceedings*, pages 123–130. Technical University of Aachen (RWTH), 2003.

[88] Cristian Pérez de Laborda and Stefan Conrad. Relational.OWL - A Data and Schema Representation Format Based on OWL. In Sven Hartmann and Markus Stumptner, editors, *Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005), Newcastle, Australia*, volume 43 of *CRPIT*, pages 89–96. ACS, 2005.

[89] Cristian Pérez de Laborda, Christopher Popfinger, and Stefan Conrad. DÍGAME: A Vision of an Active Multidatabase with Push-based Schema and Data Propagation. In *Proceedings of the GI-/GMDS-Workshop on Enterprise Application Integration (EAI'04)*, volume 93 of *CEUR Workshop Proceedings*, pages 49–56, 2004.

[90] Cristian Pérez de Laborda, Christopher Popfinger, and Stefan Conrad. Dynamic Intra- and Inter-Enterprise Collaboration Using an Enhanced Multidatabase Architecture. In *DEXA Workshop Proceedings of the 16th Intl. Workshop on Web Based Collaboration (WBC 2005), August 22 - 26, Copenhagen, Denmark*, pages 626–631. IEEE Computer Society Press, 2005.

[91] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[92] Christopher Popfinger and Stefan Conrad. Maintaining Global Integrity in Federated Relational Databases using Interactive Component Systems. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, volume 3760 of *Lecture Notes in Computer Science*, pages 539–556. Springer, 2005.

[93] Christopher Popfinger and Stefan Conrad. Tightly-coupled Wrappers with Event Detection Subsystem for Heterogeneous Information Systems. In *DEXA Workshop Proceedings of the 8th Intl. Workshop on Network-Based Information Systems (NBiS 2005), August 22 - 26, Copenhagen, Denmark*, pages 62–66. IEEE Computer Society Press, 2005.

[94] Christopher Popfinger, Cristian Pérez de Laborda, and Stefan Conrad. DÍGAME: A Push-based P2P Database. In L.M. MacKinnon, A.G. Burger, and P.W. Trinder, editors, *BNCOD21 (21st Annual British National Conference on Databases), Proceedings Volume 2*, pages 45–46. Heriot Watt University, 2004.

[95] Christopher Popfinger, Cristian Pérez de Laborda, and Stefan Conrad. Link Patterns for Modeling Information Grids and P2P Networks. In Il-Yeol Song and Stephen W. Liddle and Tok Wang Ling and Peter Scheuermann, editor, *Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling, Shanghai, China, November 8-12, 2004*, volume 3288 of *Lecture Notes in Computer Science*, pages 388–401. Springer, 2004.

[96] PostgreSQL Homepage. http://www.postgresql.org, 2006.

[97] Ramana Rao, Stuart K. Card, Herbert D. Jellinek, Jock D. Mackinlay, and George G. Robertson. The Information Grid: A Framework for Information Retrieval and Retrieval-Centered Applications. In *Proceedings of the 5th Annual Symposium on User Interface Software and Technology (UIST'92)*, pages 23–32, 1992.

[98] Juan Raposo, Alberto Pan, Manuel Álvarez, and Ángel Viña. Automatic Wrapper Maintenance for Semi-structured Web Sources using Results from Previous Queries. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 654–659. ACM Press, 2005.

[99] Mary Tork Roth and Peter M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 266–275. Morgan Kaufmann, 1997.

[100] David De Roure, Mark A. Baker, Nicholas R. Jennings, and Nigel R. Shadbolt. The Evolution of the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 65–100. John Wiley & Sons Inc., New York, April 2003.

[101] SD Times Survey: Relational Databases Rule the Roost. http://www.sdtimes.com, 2004.

[102] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[103] Jacob Slonim, Fred J. Maryanski, and Paul S. Fisher. Mediator: An Integrated Approach to Information Retrieval. In *Proceedings of the First International ACM SIGIR Conference on Information Storage and Retrieval*, pages 14–36. ACM Press, 1978.

[104] Stefano Spaccapietra and Christine Parent. View Integration: A Step Forward in Solving Structural Conflicts. *Transactions on Knowledge and Data Engineering*, 6(2):258–274, 1994.

[105] Stefano Spaccapietra, Christine Parent, and Yann Dupont. Model Independent Assertions for Integration of Heterogeneous Schemas. *The VLDB Journal*, 1(1):81–126, 1992.

[106] Mark Spenik and Orryn Sledge. *Microsoft SQL Server 2000 DBA Survival Guide*. Sams, 2003.

[107] Heinz Stockinger. Distributed Database Management Systems and the Data Grid. In *Proceedings of the 18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, San Diego, CA, 2001.

[108] Sandra Suljic. Fehleranalyse bei 'Injected Transactions'. Bachelor's Thesis. Heinrich-Heine-Universität Düsseldorf, 2005.

[109] Radhakrishnan Sundaresan, Tahsin M. Kurç, Mario Lauria, Srinivasan Parthasarathy, and Joel H. Saltz. A Slacker Coherence Protocol for Pull-based Monitoring of On-line Data Source. In *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 250–257. IEEE Computer Society, 2003.

[110] Sybase Homepage. http://www.sybase.com, 2006.

[111] Sybase Transact-SQL User's Guide - Application Server Enterprise 15.0. http://infocenter.sybase.com, 2005.

[112] Igor Tatarinov, Zachary Ives, Madhavan Jayant, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The Piazza Peer Data Management Project. *SIGMOD Record*, 32(3), 2003.

[113] Alexander Tchernin. Aktive Ereignisübermittlung in einer eng gekoppelten Wrapper-Architektur. Bachelor's Thesis. Heinrich-Heine-Universität Düsseldorf, 2005.

[114] W. B. Teeuw and H. M. Blanken. Control Versus Data Flow in Parallel Database Machines. *IEEE Transactions on Parallel Distributed Systems*, 4(11):1265–1279, 1993.

[115] Can Türker and Stefan Conrad. Towards Maintaining Integrity of Federated Databases. In *Data Management Systems, Proceedings of the 3rd Int. Workshop on Information Technology, BIWIT'97, July 2–4, 1997, Biarritz, France*, pages 93–100, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[116] Marshall van Alstyne, Erik Brynjolfsson, and Stuart Madnick. Why not one big database?: Principles for data ownership. *Decision Support Systems*, 15(4):267–284, 1995.

[117] Günter von Bültzingsloewen, Arne Koschel, and Ralf Kramer. Active Information Delivery in a CORBA-based Distributed Information System. In *On The Move to Meaningful Internet Systems 1996: CoopIS, DOA, and ODBASE*, pages 218–227. Springer, 1996.

[118] Yuan Wang, David J. DeWitt, and Jin-yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. In *Proceedings of the International Conference of Data Engineering*, pages 519–530, 2003.

[119] Seth White, Maydene Fisher, and Rick Cattell. *JDBC API Tutorial and Reference*. Addison Wesley, 2001.

[120] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, 1992.

[121] Janet L. Wiener, Himanshu Gupta, Wilburt Labio, Yue Zhuge, Hector Garcia-Molina, and Jennifer Widom. A System Prototype for Warehouse View Maintenance. In *Proceedings of the Workshop on Materialized Views: Techniques and Applications*, pages 26–33, 1996.

[122] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Database Replication Techniques: a Three Parameter Classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, pages 206–215. IEEE Computer Society, 2000.

[123] Elizabeth Winey. Data Flow Architecture. In *Proceedings of the 16th Annual Southeast Regional Conference*, pages 103–108. ACM Press, 1978.

[124] William Winkler. The State of Record Linkage and Current Research Problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Wachington, DC, 1999.

[125] Jian Yang, Mike P. Papazoglou, and Bernd J. Krämer. A Publish/Subscribe Scheme for Peer-to-Peer Database Networks. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 244–262. Springer, 2003.

[126] Xin Zhang, Lingli Ding, and Elke A. Rundensteiner. Parallel Multisource View Maintenance. *The VLDB Journal*, 13(1):22–48, 2004.

[127] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 316–327. ACM Press, 1995.

List of Figures