Managing Object Identity in Federated Database Systems

Ingo Schmitt and Gunter Saake

Otto-von-Guericke-Universität Magdeburg
Institut für Technische Informationssysteme
Universitätsplatz 2, D-39106 Magdeburg, Germany
E-mail: {schmitt|saake}@iti.cs.uni-magdeburg.de
Phone: ++49-391-67-12994 Fax: ++49-391-67-12020

Abstract. The federation of heterogeneous data management systems is still a challenge for the database system research. A main topic in federated system environments is the global identification of objects. On the federation level, a framework for object identification is needed that meets the demand for global uniqueness and allows to access the objects in the local systems. Furthermore, we suggest an identification module in this paper which satisfies these requirements and supports the handling of proxy-objects representing the same real-world entity as well as restructuring objects during the federation process. After describing this module we shall examine some aspects of implementation.

1 Introduction

In many large organizations, different legacy data management systems are maintained and operated. These data management systems and the data managed by them have developed independently from each other. The data management systems are often database management systems or merely file-based systems differing in several aspects such as data model, query language, system architecture, etc. as well as the structure and semantics of the data managed. In this context the term 'heterogeneity of the data management systems' is commonly used. As a rule, applications for specific needs continue to be based on such systems. More recent applications often require access to the distributed databases but their implementation fails due to heterogeneity. *Federated database management systems* (FDBMS) are designed to overcome this problem. The need for FDBMS is explained in greater detail in [McL93]. In this paper we concentrate on the requirements to be attributed to FDBMS:

1. A virtual, homogeneous interface (federated schema) has to be provided that supersedes this heterogeneity;
2. The autonomy of the local data management systems (*component database system* CDBS) has to be maintained in order to guarantee that the local legacy applications can continue to be used without any changes;
3. The federated schema is to facilitate the definition of an external schema, which can either be a view or the same schema with some other underlying data model;

4. The support of mechanisms to guarantee consistency (e.g. the transaction mechanism).

In many papers such as in [SL90], architectures of FDBS's have been suggested. But the requirements mentioned above have not yet been met. Especially the process of reconciling the heterogeneous component schemata with a federated schema raises various questions, among which is the problem of how to handle properly the identity of the objects provided in a federated schema. Schema mappings, transactions and other mechanisms are only possible if intelligent techniques are used to identify the objects in heterogeneous databases [Bee93]. The issue of *identifying objects* in a federated environment was discussed in [EK91, NKS91, Mar91, HD93]. [BEK+94, Kac94] and [Rad94] describe a prototype architecture including a data identification module that serves to manage identity.

The papers mentioned describe ways to globalize local object identifiers for federation purposes. Apart from generating global identifiers this paper also analyzes some additional aspects of the object identification in a federated environment, formulates requirements and suitable operations of an identification module. Only then implementation aspects will be considered.

2 Managing Global Object Identity in a Federated Environment

The *object identity* (OID) is a property of an object that provides a means to uniquely denote or refer to the object irrespective of its state or behavior [KC86]. The object identity has two dimensions which influence the degree of support of identity. The scope dimension refers to the scope within which uniqueness can be guaranteed. The temporal dimension expresses the duration, in which the OID of an object remains unchanged. In an FDBMS the scope of the OID is represented by the federated database. The duration, however, is represented by the availability of the federated database.

The data objects available within the federation schema are stored in the local data management systems and can be accessed by means of local identifiers. [EK91] distinguishes three kinds of local support levels of object identities: *value based identity*, *session* and *immutable object identifiers*. Another classification was made by [HD93].

Objects (tuples) in relational database management systems (RDBMS) are unique due to the user-defined primary key within the scope of their relation. Any local application can change the key value of a tuple if required. Its visibility is necessary as it serves to carry additional user-defined information.

Each object in object-oriented database systems (OODBMS) is identified by an invisible and unchangeable object identifier generated by the system itself and is unique within the scope of the database. The invisible object identifiers are handled by techniques of storing object identifiers as references and comparing them with each other.

In file-based data management systems, statements about object identifiers are not predictable.

Thus an FDBMS requires a module to handle OIDs, so that it

1. possesses the property of uniqueness as mentioned above;
2. maintains the autonomy of the CDBS;
3. enables access to the objects in the CDBS via local object identifiers;
4. allows for an efficient navigation;
5. supports object migration (see also [Kac94, RS94]).

Consequently, an object identification module has to deal with system-generated *global object identifiers* (GOID), with *local object identifiers* (LOID) and with a dynamic assignment between them (see also [Mar91, NKS91, HD93]). In addition to LOID and GOID mapping, a management of information about the location of the object is needed in order to support object migration and to bridge the different scope sizes of the object identifiers.

The uniqueness scopes [Ken91] of the different object identifiers can be visualized in a scope model. In this model, tuples, objects, relations, database systems, and also the FDBS are considered as objects with a specific ID. [Bee93] strongly recommends to regard meta-constructs as objects. The identifier of each object is only unique within the scope defined by the meta-object enclosing it. From now on we shall refer to an enclosing meta-object as a *container object* and to an enclosed object as an *element object*. Each object can only be a direct element object of no more than one container object.

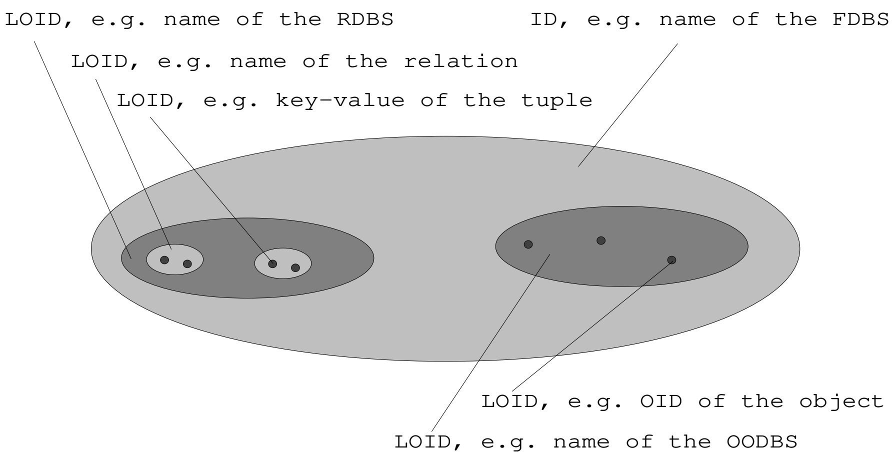


Fig. 1. Scope model

According to the model depicted in figure 1, the following identification module satisfies the requirements described above.

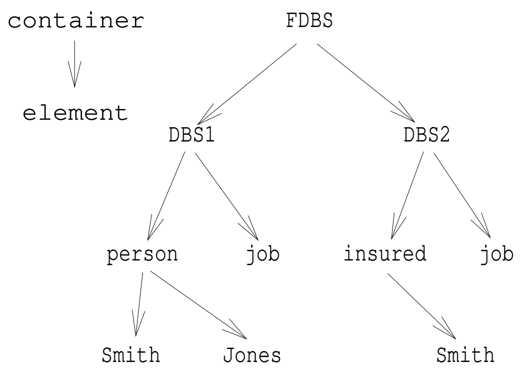
The OID module encompasses the operations *null*, *next*, *get*, *combine*, *rename*, *migrate* and *delete* and a data structure that stores the current assignments. Before a GOID can be assigned to an object which does not yet have a GOID of its own, a new GOID has to be generated. The operation *get(next)* produces unique OIDs in succession. The *combine* operation relates the LOID of a local object to a generated GOID and to the GOID of its container object. It is obvious that the GOID of the container object has to exist before the *combine* operation can be carried out. Thus we assign the default GOID *get(null)* to the FDBMS. *Get(null)* is the first possible GOID of the GOID generator. To ensure the uniqueness of the identification, each GOID has to be assigned to only one federated object at most. Furthermore, the GOID of any element object has to be different from the GOID of its container object.

As local applications are capable of changing the LOIDs of objects, any change of this kind must be announced to the OID module by means of the *rename* operation. The *rename* operation changes the LOID of an object. Thus the OID module satisfies the requirements for stable GOIDs.

Any object can migrate to another container object without changing its GOID. This procedure can be performed with the help of the *migrate* operation which only changes the assignment of the GOID of the migrated object to the GOID of its changed container object. If the migrated object is a container object then all its element objects assigned have virtually migrated accordingly.

The deletion of an object in a local application must be propagated to the OID module by means of the *delete* operation.

The effect of each operation has to be stored in a suitable data structure. The implementation of this data structure will be discussed in section 5. Operations changing the assignments have to be accompanied by query operations. The latter are not considered here. The following figure exemplifies the assignments related to the OID structure.



LOID	GOID	GOID container object
DBS1	#1	#0
DBS2	#2	#0
person	#3	#1
job	#4	#1
insured	#5	#2
job	#6	#2
Smith	#7	#3
Jones	#8	#3
Smith	#9	#5

Fig. 2. Sample scope and corresponding *is-combined* relation

3 Managing Proxy-Objects of same Real-World Entities

As the component databases are developed independently from each other, objects representing one real-world entity can be stored as local objects in more than one CDBS. These local objects have varying LOIDs and are often structured differently [RS94]. Such objects are called *proxy-objects of same real-world entities* [KAA+93, TS93, Tre95]. For example, data about the same person can be stored both in the database of a university and that of an insurance company.

The management of such proxy-objects is inevitable for homogenizing heterogeneous databases properly. The GOIDs of objects that refer to the same real entity must be related to each other. This relation can be the basis of a controlled redundancy. The problem of how to guarantee value equality between such redundancy objects might be an interesting subject, which, however, shall not be discussed here. In [TS93, Tre95] the relation between the GOIDs of such proxy-objects is called *same-relation*. The same-relation is an equivalence relation in a mathematical sense. Consequently, it has the properties of reflexivity, symmetry and transitivity. If such an equivalence relation were simply stored it would cause redundancy that may imply incorrect entries and inefficient handling. For instance, the property of symmetry would necessitate to store the tuples (GOID_1,GOID_2) and (GOID_2,GOID_1) explicitly.

The OID module proposed here contains the *is-same* relation and a derived *is-equivalent* relation that corresponds to the properties of an equivalence relation. The *same* operation establishes the relation of the GOIDs of two proxy-objects representing one same real-world entity provided that the GOIDs of these objects are not in an *is-combined* relation to each other. This handling of such proxy-object is illustrated in the next section.

4 Restructuring Objects

According to the requirements mentioned in the first section, objects have to be restructured in FDBSs. Restructuring objects means splitting objects of one schema into smaller ones or assemble a few of them to a new object of a different schema. Various aspects of restructuring objects in a federated environment are described in [RS94].

The homogenization process is facilitated by splitting objects of heterogeneous CDBS. For example, before two semantically related objects with overlapping type structures can be integrated into one object, their structure has to be adjusted by a splitting operation. Splitting operations produce new (virtual) objects with GOIDs of their own which contain different information about one real-world object. In this paper we consider neither the conversion of attributes into separate objects nor vice versa [KAA+93]. Mapping between objects in a federated schema and objects in a component schema requires the management of *is-composed* relations between the objects involved in the splitting operation.

An FDBS has to support the generation of views and schemata with other underlying data models form an external schema [Bee93, SCG91]. This can be achieved by assembling new (virtual) objects with their own GOIDs from other existing objects which may have been split up before. Managing the *is-composed* relation is also essential in this context in order to map external and federated schemata.

The *is-composed* relation always relates objects from different schemata. As regards the *is-composed* relation, the terms *aggregate object* and *component object* shall be used here henceforth. The relation between aggregate objects and their component objects can only be established properly by using their GOIDs. Managing the *is-composed* relation within the OID module proposed makes sense as it is dependent on the *is-equivalent* relation.

- During the homogenizing process each aggregate object of the component schema level can be split into component objects of the federated schema level. These component objects represent proxy-objects of the same real-world entity. Therefore, the derivation of the *is-equivalent* relation from the *is-same* relation includes such proxy-objects.
- *Is-same* relations between objects of the component schema level survive any splitting operations. So if two objects of the component schema level related by the *is-same* relation are decomposed into component objects of the federated schema level then these component objects are related by the *is-equivalent* relation too.
- Only component objects of the federated schema level related by the *is-equivalent* relation are allowed to be component objects of an aggregate object of the external schema level.

The OID mechanism provides the operation *decompose(component,aggregate)* to relate a new component object to its aggregate object, the operation *compose(component,aggregate)* to relate a new aggregate object to a component object and the operation *revoke-compose-decompose(component,aggregate)* to undo the *compose* and the *decompose* operations. As the transitive *is-composed* relation is hierarchic, cycles are not permitted.

Note: The *is-composed* relation must not be confused with the customary part-of relation between objects within the same schema.

The figure 3 and the table 1 below show an example of interrelations between the *is-same*, *is-combined*, *is-composed* and *is-equivalent* relations.

One object from a relational database and one from an object-oriented database are exported for federation purposes. The LOIDs of these objects ("Smith" and #5) are globalized by the *is-combined* relation to the GOIDs #8 and #9. Since the object #8 and #9 represent the same real-world entity, they are related by the *is-same* relation. The following splitting process of both objects generates the component objects #10, #11, #12 and #13 of the federated schema level which are mutually related by the *is-equivalent* relation. Therefore, the object #14 of the external schema level can be composed of the component objects #10, #11 and #13.

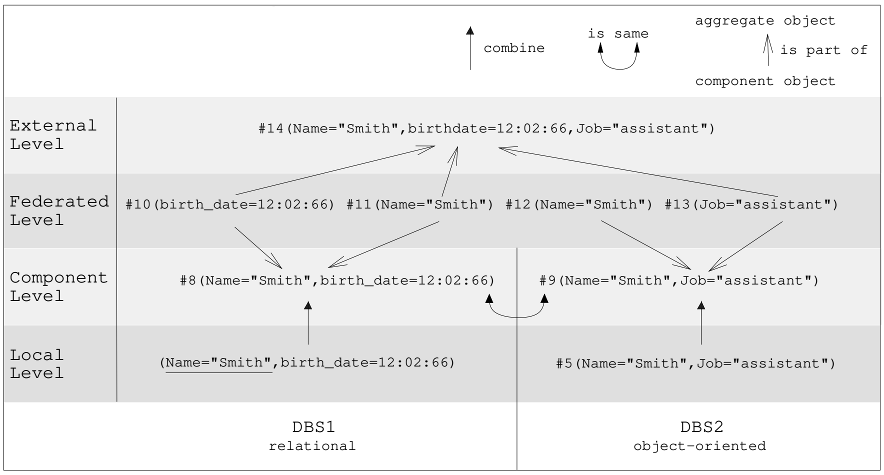


Fig. 3. Example of restructuring objects

is-same	
GOID	GOID
#8	#9

is-combined		
LOID	GOID	GOID container object
DBS1	#6	#0
DBS2	#7	#0
Smith	#8	#6
#5	#9	#7

is-composed	
component	aggregate
#10	#8
#11	#8
#12	#9
#13	#9
#10	#14
#11	#14
#13	#14

is-equivalent	
GOID	GOID
#8	#9
#10	#11
#12	#13
#11	#12
#9	#8
#8	#8
#10	#13
#13	#10
...	...

Table 1. OID relations

5 Aspects of Implementation

The most obvious implementation of the OID module presented here utilizes an RDBMS. The OID module is characterized by a state-dependent semantics[1]. The state has to be structured and maintained by the DBMS. It can be expressed by the *is-combined*, *is-same*, *is-composed* relations and the single column and single row relation termed *counter* relation that represents the current counter value of the GOID generator. The data type of a GOID can be the SQL data type *integer*.

[1] A formal algebraic specification of the OID module can be obtained from ftp anonymous on otto.cs.uni-magdeburg.de in /pub/identity/Spec.ps or from www on http://max.cs.uni-magdeburg.de/institut/staff/schmitt.html.

Due to the different data types of the given LOIDs, the data type *char(n)* as a sequence of bytes and suitable (bijective) converters can be employed to represent the LOID's [Mar91].

The operation

- *null* initializes the state. The relations have to be empty but the *counter* value has to be set to 0;
- *next* increases the *counter* value;
- *get* only reads the current *counter* value;
- *combine*, *same*, *compose* and *decompose* are expressed by the SQL *Insert* commands which insert new tuples into the *is-combined*, *is-same* and *is-composed* relations;
- *rename* and *migrate* update a tuple of the *is-combined* relation;
- *delete* removes tuples from the *is-combined*, *is-same* and *is-composed* relations;
- *revoke-compose-decompose* merely removes tuples from the *is-composed* relation.

Most operations have to check some integrity conditions before they can be carried out. During this check the current state has to be read. Queries about the state of the OID module can be performed by the *Select* command of SQL.

The derivation of the *is-combined-trans* (transitivity), *is-equivalent* (reflexivity, symmetry, transitivity) and *is-composed-trans* relations from the *is-combined*, *is-same* and *is-composed* relations requires the transitive closure to be computed. This computation cannot be achieved by standard SQL commands. But some RDBMSs offer additional functions to solve this problem. In this case the relations derived can be handled as views. Otherwise the computation has to be done operationally.

In the following we shall discuss some problems with respect to the implementation of our OID module. We shall try to outline some improvements.

1. *Problem*: The *counter* relation and some update (*next*) and select (*get*) commands are too awkward for generating GOIDs.
   Solution: Most commercial RDBMS offer sequence generators which can be used instead.
2. *Problem*: Applications operating on the federated level require efficient techniques for handling objects. This handling is influenced by the OID module. The efficiency (access time) of the OID module is to be improved.
   Solution: Suitable cache algorithms are able to decrease the access time to complex objects using GOIDs [KM92].
3. *Problem*: LOIDs can be changed by local applications. These local changes have to be propagated to the global GOID module by the *rename* operation. Since the GOID module is located on the global level, the above mentioned propagation goes beyond the boundaries of the local data management systems. Therefore, it is often impossible to use system based propagation techniques. Other techniques can be inefficient.

Solution: The *is-combined*(LOID,GOID_1,container object GOID_2) relation is split up vertically into the r1(LOID,GOID_1) relation and the r2(GOID_1, container object GOID_2) relation. r1 can then be subdivided horizontally into further relations. Each of them only contains GOID's of objects of the same data management system and can therefore reside in the local data management systems. Now, local changes of OIDs will influence the local mapping table only. The splitting process can be continued into smaller relations that contain only element objects of the same container object, e.g. relation or class.

A *migration* operation would then require to move both the objects and their respective r1(LOID, GOID_1) tuples to another container object. The r2 relation cannot be moved from the global level to the local level, because any access to an object via GOID from global level account is not feasible without any navigation information, i.e. the information, in which CDBS the required object is stored.

4. *Problem*: Local applications have to announce the creation and deletion of any object as well as changes of LOIDs to our GOID module. But local applications have to be considered as autonomous systems. The problem is here, how to detect an occurrence of such an operation without violating the autonomy of the local applications [EK91].

   Possible solutions: Up to now no general solution has been known which enables stable OID's and likewise guarantees the autonomy of local applications. But considering the given local situations some of the following methods can be applied.

   - A number of data management systems offer methods to detect these critical operations. Such methods can be trigger techniques or ways of generic operations with cascading effects [Mar91]. Using these methods local applications do not have to be changed.
   - If a data management system does not support such detecting techniques, the evaluation of system data can be used to detect the specific operations. For instance, the transactions log file can be monitored [EK91].
   - If updates on LOIDs are not allowed, the mapping table remains correct. Some applications satisfy this requirement. Modifying other local applications is acceptable. But there may be local applications for which such modifications are unacceptable because their autonomy should not be violated.
   - If the creation and deletion of objects and the change of LOIDs are not detectable, a polling technique, that periodically scans the data managed can detect the effects of critical operations. But it is often impossible to conclude the operation that has caused that effect. This method is only useful to detect problems and to announce them to the administrator.
   - If the data management system is an OODBMS, the stored objects can be modified by an overriding technique. Any critical method will have to be refined adding operations that announce changes to the GOID

module. Using this method will violate the autonomy of the local applications. The refined classes will have to be relinked to the applications.

Note: The demand for autonomy of local applications is contradictory to the requirement to stable GOIDs. Therefore, [HD93] advises to refrain from the requirements to stable GOIDs. In our opinion, reliable GOIDs are more important than maintaining strict autonomy.

5. *Problem*: The LOID-GOID_1 mapping table is an additionally managed data object. Managing additional data objects can be inefficient.
   Solution: The GOID of an object is appended to the local object data structures. Relations are extended by a new column using the *Alter Table ... Add Column ...* commands and the type of classes are extended by using the inheritance technique. This method also violates the autonomy of local applications.
6. *Problem*: How can the *is-combined*, *is-same* and *is-composed* relations be built?
   The process of building these relations can be subdivided into two phases, the preparation and the online phase. During the preparation phase both the existing exported schemata and the local legacy data have to be integrated with no user application running. But user applications can create, update and delete local objects that have to be exported during the online phase. Such operations have to be propagated to the federated database system as soon as possible by one of the technique described above.
   Building the
   - *is-combined* relation during the preparation phase simply means scanning the extensions of the exported meta-objects, generating new GOID's and storing them as *is-combined* tuples.
   - *is-same* relation appears more difficult than building the *is-combined* relation. The problem is how to recognize proxy-objects related to same entities which are based on local data. [LSPR93] describe various aspects of this process.
   - *is-composed* relation depends on the homogenizing process. Due to the limitation of this paper, this aspect is not detailed here. Future work on homogenizing heterogeneous schemata by means of the OID module are to follow soon.

We believe that these suggestions can contribute to improving real OID techniques in specific situations. Some methods, however, will violate the autonomy of local applications. Other methods are inefficient, but preserve this autonomy. Therefore, both the specific situation of the federation and the applications have to be taken into account when using any of the methods described above.

6 Conclusions

The first part of this paper described requirements attributed to the OID module in a federated environment. Then the OID module proposed was developed in order to obtain unique global object identifiers on the federation level. Due to the dynamic assignment of LOID's to GOID's in federated objects, local changes of the LOID's of objects will not affect their GOID's. The OID module suggested also supports the management of location information about the objects referred to by the GOID's. A migration operation on objects will change their locations, that is why it has to be announced to the module. This module is able to guarantee stable GOID's after such migration operations have taken place.

During the homogenization process proxy-objects of same real-world entities are likely to appear. There are specific integrity conditions, e.g. the semantic equivalence of the values of the same attributes that have to be considered. In order to monitor such conditions, the relations between the GOID's of such proxy-objects managed by the OID module have to be known.

In order to homogenize CDBSs and to establish external levels, objects will have to be restructured. New, smaller objects arise from splitting operations and bigger objects from merging operations. The new (virtual) objects are existentially dependent on the old (real or virtual) objects. In order to enable consistency techniques their connections have to be known. Our OID module allows for the handling of object relations involved in a restructuring process.

The previous section demonstrated that the demand of autonomy for local applications is contradictory to the requirement to stable GOIDs. Compromises between them are listed. The choice of the proper compromise is dependent on local and current situations.

Improvements in terms of aspects of implementation are open for further investigation. Moreover, the feasibility of the OID module suggested in the present paper is still to be explored in the practical field.

Acknowledgements

Thanks to Can Türker for many creative discussions. We also have benefited from linguistic advice provided by Mark Krogel and Dietmar Schmitt.

References

[Bee93] C. Beeri. Some Thoughts on the Future Evolution for Object-Oriented Databases Concepts. In W. Stucky and A. Oberweis, editors, *Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft" (BTW'93), Braunschweig*, pages 18–32. Springer-Verlag, Informatik aktuell, May 1993.

[BEK+94] R. Böttger, Y. Engel, G. Kachel, S. Kolmschlag, D. Nolte, and E. Radeke. Enhancing the Data Openness of Framworks by Database Federation Services. Technical Report, Cadlab Paderborn, 1994.

[EK91] F. Eliassen and R. Karlsen. Interoperability and Object Identity. *ACM SIGMOD RECORD*, 20(4):25–29, December 1991.

[HD93] M. Härtig and K. Dittrich. Objektidentifikation in Heterogenen Datenbanksystemen. In W. Stucky and A Oberweis, editors, *Proc. of GI-Conf.*

Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'93), pages 296–305. Springer-Verlag, Informatik aktuell, May 1993 (in german).

[KAA+93] W. Kent, R. Ahmed, J. Albert, M. Ketabchi, and M. Shan. Object identification in multidatabase systems. In *Proc. of IFID DS-5 Working Conference on the Semantics on Interoperable Systems*, Elsevier, 1993.

[Kac94] Gerd Kachel. Coupling of Heterogeneous Databases by Federation Services. Technical Report, Cadlab Paderborn, September 1994.

[KC86] S. N. Khoshafian and G. P. Copeland. Object Identity. In N. Meyrowitz, editor, *Proc. of the 1st Int. Conf. on Object Oriented Programming Systems, Languages and Applications, Portland, Oregon*, pages 406–416. ACM Press, SIGPLAN Notices 21(11), November 1986.

[Ken91] W. Kent. A Rigorous Model of Object Reference, Identity, and Existence. *Journal of Object-Oriented Programming*, 1991.

[KM92] K. Kato and T. Masuda. Persistent Caching: An Implementation Technique for Complex Objects with Object Identity. *IEEE Transaction on Software Engineering*, July 1992.

[LSPR93] E.-P. Lim, J. Srivastava, S. Prabhakar, and J. Richardson. Entity identification in database integration. In *Proc. 9th Intern. Conf. on Data Engineering Conference*, pages 294–301, Vienna, Austria, April 1993.

[Mar91] V. Markowitz. An Architecture for Identifying Objects in Multidatabases. In *Proc. of the First Int. Workshop on Interoperability in Multidatabase Systems*, pages 294–301. IEEE Computer Society and the Information Processing Society of Japan, April 1991.

[McL93] D. McLeod. Beyond Object Databases. In W. Stucky and A. Oberweis, editors, *Proc. of GI-Conf. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'93)*, pages 1–17. Springer-Verlag, Informatik aktuell, May 1993.

[NKS91] E. Neuhold, W. Kent, and M. Shan. Object Identification in Interoperable Database Systems. In *Proc. of the First Int. Workshop on Interoperability in Multidatabase Systems, Kyoto, Japan*, pages 302–305. IEEE Computer Society, April 1991.

[Rad94] E. Radeke. Efendi: Federated Database System of Cadlab. Technical Report, University of Paderborn & Siemens Nixdorf, 1994.

[RS94] E. Radeke and M. H. Scholl. Federation and Stepwise Reduction of Database Systems. In W. Litwin and T. Risch, editors, *Proc. of the 1st Int. Conf. on Applications of Databases, Sweden*, pages 381–399. Springer-Verlag, LNCS 819, June 1994.

[SCG91] F. Saltor, Castellanos, and M. Garcia-Solaco. Suitability of Data Models as Canonical Models for Federated Databases. In *Proc. of the ACM Conf. on Management of Data (SIGMOD'91)*, pages 44–48. ACM Press, 1991.

[SL90] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogenous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[Tre95] M. Tresch. *Evolution in Objekt-Datenbanken*. Teubner Verlag Stuttgart, Leipzig, 1995 (in german).

[TS93] M. Tresch and M. Scholl. Schema Transformation Processors for Federated Objectbases. In *Proc. of the 3rd Int. Symposium on Database Systems for Advanced Applications (DADFAA'93), Daejon, Korea*, April 1993.