Techn. Report 94-07, Univ. of Ulm, Fac. of Computer Science, July 1994. A short version appeared in:
Proc. 3rd Int'l Conf. on Parallel and Distributed Information Systems (PDIS), Austin, Texas, Sept. 1994.

A Classification of Multi-Database Languages

Markus Tresch [*]

IBM Almaden Research Center
650 Harry Road (K55/801)
San Jose, CA 95118, USA
tresch@almaden.ibm.com

Marc H. Scholl

University of Ulm
Faculty of Computer Science
D – 89069 Ulm, Germany
scholl@informatik.uni-ulm.de

Abstract

This paper defines a formal classification of multi-database languages into five levels of database integration with increasing degree of global control and decreasing degree of local autonomy. First, the fundamental interoperability concepts and mechanisms are identified for each of these levels. Their consequences on local autonomy as well as implementation draw-backs are discussed. Second, various multi-database languages are classified into these categories. In addition to our own language COOL*, other proposals are analyzed, including SQL*Net, Multibase, Superviews, VODAK, Pegasus, and O*SQL.

Keywords: federated database systems, multi-databases languages, object algebra, views, object unification, object identity, autonomy.

1 Introduction

Novel data-intensive information systems are characterized by cooperating (autonomous and heterogeneous) database systems and therefore increasingly require openness of database management systems (DBMSs) for a cooperation with other services, be they data managers or other service providers. Hence, the area of interoperable multi-database systems (MDBSs) has attracted a lot of recent attention, both in research and practice. Practical solutions of today typically consist of several DBMSs that are loosely integrated via data extraction – data conversion – data upload cycles. This requires extensive and error-prone application programming, yet guarantees only a minimum of data consistency. The challenge for future cooperative systems is to provide flexible and scalable mechanisms to support system-controlled interaction among different data management systems.

A wide variety of problems need to be solved in order to make MDBS work, including such diverse issues as MDBS transaction management, data model transformation, schema integration, MDBS query languages and optimization, and data and application migration. This paper concentrates on MDBS language aspects for integration of data (schema and

[*]Work done while at Faculty of Computer Science, University of Ulm, Germany.

instance level) from different component databases. We emphasize on homogeneous multi-databases, that is, we separate the issue of data model transformation from the rest, and assume that all schemas have already been transformed into a uniform data model.

Multi-databases systems are built up of several component database systems (CDBS) managing local component databases $DB_1, DB_2, \ldots$, and a federation dictionary FD managed by the MDBS itself (cf. Figure 1). The purpose of an MDBS is to support global operations (queries and updates) on objects stored in different CDBSs consistently. At the same time, CDBSs should continue autonomous processing of local operations.

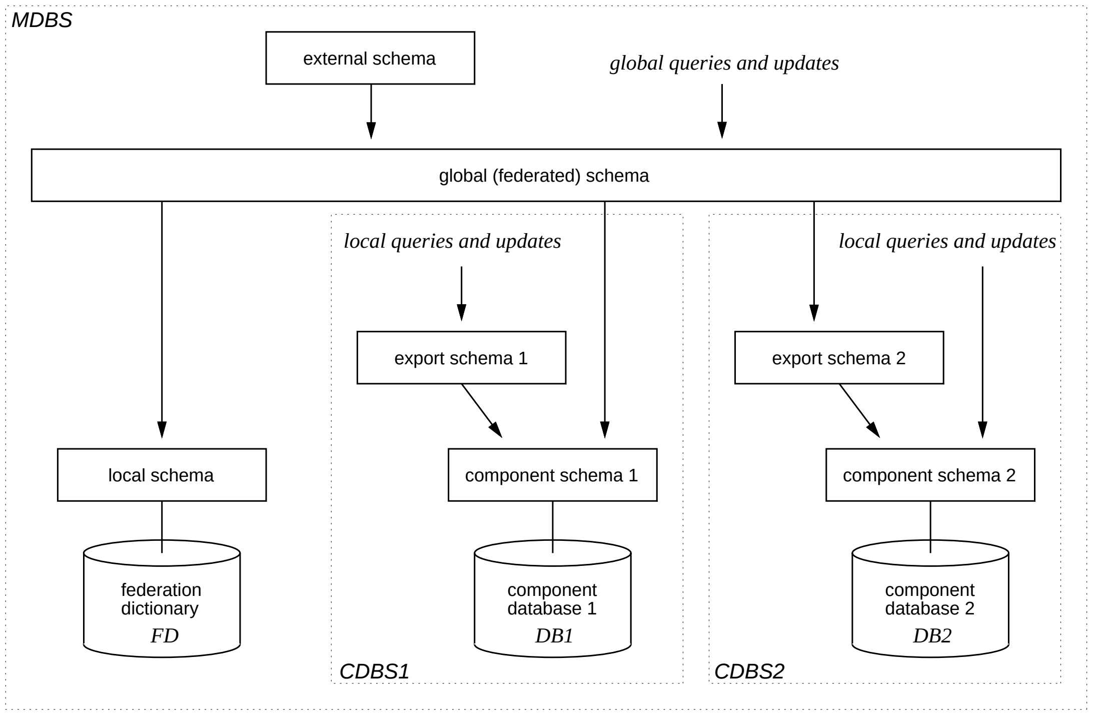

Figure 1: Schemas and Operations in Multi-Database Systems

Following the reference architecture of [SL90], the structure of each DB_i is given by a component schema and the structure of the multi-database is given by the global (federated) schema, which is an integration of (parts of) component schemas. The FD contains (meta) information about the distribution and integration of schemas.

The contribution of this paper is a classification of MDBS languages into five different integration levels ranging from very loosely coupled multi-database systems (Level 0), through three levels of federated DBMSs (Levels I–III), to fully integrated, distributed DBMSs (Level IV). We are most interested in the three intermediate levels I–III. These are separated by the way how local objects in CDBSs that represent "the same" real world entity can be identified and tied together in the MDBS. These levels of database integration are also a measure for the degree of autonomy that component systems have to give up as the price for closer cooperation. We clearly point out what the benefits and consequences of a particular MDBS integration level w.r.t. local autonomy and global control is.

The classification is presented using our own database language, COOL*, where all constructs are given sound and formal semantics. However, the classification is data model/language independent and we categorize other, related MDBS proposals accordingly by mapping some of the constructs proposed to their COOL* counterparts.

This paper is organized as follows: In the next section, we review the basic interoperability mechanisms. In Section 3 we define the classification into five levels of MDBS integration. Section 4 uses this as a platform to compare and classify various current MDBS language proposals. We conclude with a summary and outlook on future work in Section 5.

2 Basic Database Interoperability Mechanism

Stepwise database integration is the idea that previously isolated DBMSs (populated with local objects and running local applications) are starting to cooperate with other systems very loosely (e.g. by global transactions). Later, global schemas (e.g. views) are defined, such that systems are getting more tightly coupled, until they might eventually be completely integrated. In this section, we define "*same*-functions" [SST94], a basic abstraction mechanism, to which object and schema level integration can be reduced.

Object Identification. In an MDBS, *entity objects* (objects of the real world) are to be distinguished from *proxy objects* (their approximation in one database) [Ken91]. One particular entity object can be represented by multiple proxy objects in different component databases, and therefore, the fundamental assumption of object-oriented systems (every real world object corresponds to exactly one database object) is no longer true.

More formally: due to local design autonomy, the OID domains of different CDBSs are pairwise disjoint, such that no two proxy objects from different CDBSs can be the same (identical). This leads to the following first definition of global object identity:

Definition 1. (Global object identity – preliminary) The global object identity ($=_{gl}$) of multi-database objects o_1, o_2 is defined as

$$\begin{array}{l} =_{gl} : \mathbf{object} \times \mathbf{object} \rightarrow \mathbf{bool} \\ o_1 =_{gl} o_2 \iff \exists i : object_i(o_1) \wedge object_i(o_2) \wedge o_1 =_i o_2 \ . \end{array}$$

The notion $object_i(o)$ is to be read as a type predicate: it is true, iff o is an instance of type $\mathbf{object}_i$, i.e., is an object of component database DB_i.

One main task of database integration is to identify and unify proxy objects of different CDBSs, if they represent the same entity object.

Object Integration. Let o_i and o_j be two proxy objects from different CDBSs, representing the same (real world) entity object. Object integration requires a mechanism to logically unify o_i and o_j, such that the MDBS treats them as one single object in global queries and updates. As we already know, OIDs are not adequate to globally identify objects, since they are internal object representations within *one* CDBS. Entity objects can only be globally identified by characterizing values ("value identifiability" [Bee93], a generalization of identification keys from relational systems).

One approach would be to generate new, global proxies in an MDBS and somehow link them to the local proxies via translation tables maintained in the federation dictionary. We formalize this by special functions with special semantics ("the same"), defining a global MDBS integrity constraint, which is known to the global query and update operations. Such partial, injective, single-valued functions are called $same_{i,j}$:

$$\textbf{define function } same_{i,j} : \textbf{object}_i \rightarrow \textbf{object}_j$$

same-functions are inter-database functions with domain $\textbf{object}_i$ in database DB_i and range $\textbf{object}_j$ in database DB_j, and returning for a given DB_i-proxy object the "same" DB_j-proxy object (if any). Having *same*-functions, the definition of global object identity must be reconsidered:

Definition 2. (Global object identity – revised) The global object identity ($=_{gl}$) of multi-database objects o_1, o_2 is defined as

$$\begin{aligned} &=_{gl} : \textbf{object} \times \textbf{object} \rightarrow \textbf{bool} \\ &o_1 =_{gl} o_2 \iff (\exists i : object_i(o_1) \wedge object_i(o_2) \wedge o_1 =_i o_2) \\ &\qquad\qquad \vee\ (\exists same_{i,j} : \textbf{object}_i \rightarrow \textbf{object}_j : \\ &\qquad\qquad\quad object_i(o_1) \wedge object_j(o_2) \wedge o_2 =_j same_{i,j}(o_1)) \ . \end{aligned}$$

From now on: two objects are the same, if they stem from the same CDBS and are identical in it, *or* if they have been defined (by the user/DBA) to be the same using *same*-functions.

Schema Integration. The goal of schema integration is to find out what the common (structural) parts in the local schemas are and to define correspondences among them. In contrast to [BLN86, SPD92], our matter of concern is not to find another schema integration methodology for resolving structural and semantic conflicts. Rather we are interested in identifying (and later classifying) the necessary basic abstraction mechanisms for elementary database integration.

All CDBSs contain a meta database, that is, a database with objects representing the application schema. Every schema element is represented by a *schema object* in the CDBS meta database. Of course, the concrete structure of a CDBS meta database depends on the particular data model. In COOL* for example, objects of the meta database represent persistent variables, types, classes, and functions. However, the conceptual idea of representing the schema itself by objects remains unchanged for any data model.

Once CDBS meta databases are available, the same technology as discussed in the previous paragraph can be used to define correspondences between schemas of different CDBSs. As for "ordinary" object integration, we use *same*-functions for schema integration, but now applied on schema objects of the meta database, representing variables, types, classes or functions.

Integrating for example attributes (functions) *birthdate* from DB_i and *dateofbirth* from DB_j, we define a *same*-function between those objects of the CDBS meta database, representing these two functions. After that, the multi-database language treats these two integrated attributes as if they where one single global attribute.[1]

same-functions are to be understood as the basic, data model independent abstraction mechanism for object *and* schema integration, used within this paper. Schema integration methodologies/strategies, as proposed e.g. in [BLN86, SPD92], can be implemented, using *same*-functions as base technology (cf. Section 4). Instead of *same*-functions, one may alternatively think of global query expressions or relations (tables) mapping between objects from different CDBSs. Concrete implementation alternatives for such *same*-functions for different data models are discussed in the next section.

3 Five Levels of Multi-Database Integration

We now formalize the idea of stepwise database integration. We define a classification of MDBS languages into five levels of database integration with increasing degrees of global control and decreasing degrees of local autonomy (cf. Figure 2). This classification refines [SL90] that distinguishs between losely and tightly coupled database systems only.

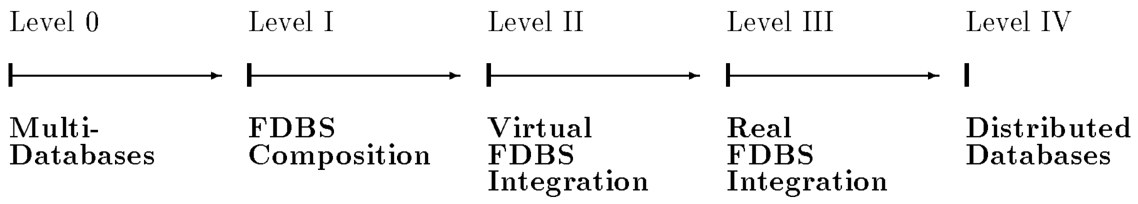

Figure 2: Stepwise Integration of Multi-Database Systems

At the leftmost end, level 0 integration represents non-integrated MDBSs.[2] This is the weakest form of database coupling, where component systems are completely independent of each other (fully autonomous). Neither objects, nor schemas are integrated. Level 0 is a kind of ad hoc data "integration". Global transaction management allows to process objects from different CDBSs within one global transaction, however, each individual query or update statement works on only one CDBS.

At the rightmost end, level IV represents fully integrated (maybe physically distributed) databases. Either, there exists only one single global DBMS, or participating component systems completely lost their local autonomy. Though objects might be physically distributed, these systems have one single logical database schema. Distribution is logically transparent, the system is fully integrated.

[1]Notice that, 1. the signatures of schema elements to be unified must be compatible, that is, they must have same names and structures; 2. unifying schema elements my cause value conflicts, that is, two attributes e.g. may be unified though they have different local values. The discussion of these issues is out of the scope of this paper; we refer to [SST94, ST94].

[2]The term "multi-database system" is overloaded: first, it is the general notion for multiple cooperative database systems, and second, it means in our classification non-integrated databases.

In between these two extremes, levels I, II, and III describe federated database systems (FDBS). They are the most interesting architectures, because on the one hand, their objects and schemas are subject to some global control, and on the other hand, participating CDBSs have retained some local autonomy. In the sequel, we therefore focus on these levels, that is, on federated object database systems. Besides a general, informal discussion of each level, we use the COOL* multi-database language [SLT91, SLR$^+$92] to illustrate the possibilities of a particular level, giving concrete examples. However, we emphasize that the conceptual idea is not bound to that data model, but can be transferred to other approaches, as we will see later in Section 4.

3.1 Level I: Composition

Intgration level I is called *schema composition*. It is the elementary process to combine multiple CDBSs DB_i into one composite schema GDB. It is therefore the foundation for establishing a federated database system. Schema composition places only minimal requirements on the degree of integration between participating systems. It basically just imports the names of all schema elements from CDBSs and makes them globally available, without establishing any connection between composite systems.

Furthermore, composition combines the type and class systems of the local databases. As an anchor, basic data types of component systems are assumed to be identical.[3] This ensures that at least values of elementary data types can be compared between component systems. Based on that, local object type and class hierarchies of the CDBSs are put together – in a so far trivial way – by defining a new global top type, being the common supertype of all local root types, and a new global top class, being the common superclass of all local root classes.[4]

In COOL* for example, names of persistent variables, functions, types, classes, and views are made globally available. A global hierarchy of object types is created with a new top type **object**@GDB, of wich all top types of the CDBSs (**object**@DB_i) are made direct subtypes. COOL* has a type lattice, therefore, a new bottom type **bottom**@GDB is made common subtype of all local bottom types. Similar, a global class hierarchy is established, with the top element **Objects**@GDB as common superclass to all local top classes **Objects**@DB_i [SST94]. For all other data models, the effect will be similar.

Example 1: Consider a university environment, where data about students are stored in multiple databases, a library database $LibDB$, a student database $StudDB$, and an employee database $EmplDB$. The following COOL* statements compose these three CDBSs into one global schema $UnivDB$:

```
define database UnivDB
    import LibDB, StudDB, EmplDB
end.
```

A graphical representation of the composition is given in Figure 3. ◇

[3]That is, the internal representations of integer, string, boolean are identical, or alternatively, an equivalence preserving transformation exists.

[4]In the sequel, we use the naming convention that schema components are suffixed by "@" and the name of the local schema. For example, class *Books* in *LibDB* has as globally unique name "*Books@LibDB*".

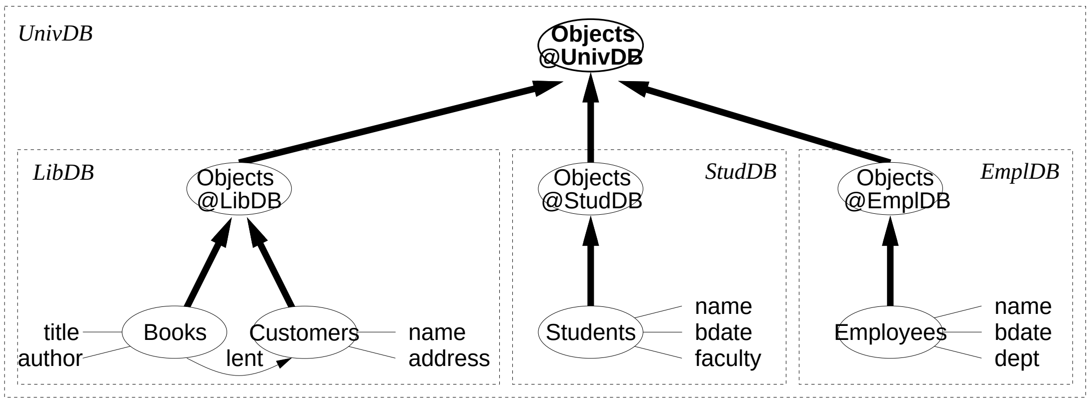

Figure 3: Composition of *LibDB*, *StudDB*, and *EmplDB*

Schema composition creates a global meta schema as well. This is the meta schema of GDB and has exactly the same structure as the meta schema of each DB_i. It contains global meta types and meta classes, being direct supertypes/superclasses of their local counterparts.

For the COOL* system used here, the meta schema of each DB_i contains meta types and meta classes representing persistent variables, functions, types, classes, and views. Though the concrete meta schema depends on the used data model, the idea of a composite meta schema remains unchanged for any other approach.

Once two (or more) schemas are composite, queries can be formulated that involve multiple CDBSs. Recall composition *UnivDB* from Example 1. Since composition made basic data types and name spaces globally available, comparing names of customers (from *LibDB*) with names of students (from *StudDB*) is legal. Hence, the following valid **select** query selects those customers being students as well:

$$\textbf{select}[\emptyset \neq \textbf{select}[name(c) = name(s)](s : Students)](c : Customers)$$

Unfortunately, the possibilities of inter-database queries are very limited up to now. E.g., the following more elegant solution of the same query is not allowed:

$$\textbf{select}[c \in Students](c : Customers)$$

Since objects of class *Students* are of type "student" and the type of c is "customer" and the two types "student" and "customer" are not (yet) related, the selection predicate $c \in Students$ would be rejected by the MDBS type checker.

The **extend** query operator defines new functions, derived by a query expression. This possibility can be used to establish connections between CDBSs. Suppose, we want to store, together with each employee (of *EmplDB*) the books (of *LibDB*), that she/he lent. The following query defines the desired new function, called *lbooks*:

$$\textbf{extend}[lbooks := \textbf{select}[name(e) = name(lent(b))](b : Books)](e : Employees)$$

The new function *lbooks* is an inter-objectbase function, linking employees from *EmplDB* with books at *LibDB*.

Schema composition (integration level I) is not yet "real database integration", in particular, no *same*-functions exist. As a consequence, no two objects can be the same (identical), unless they originate from the same DB_i and are identical in DB_i. Furthermore, type and class systems are integrated only at the very top level.

3.2 Level II: Virtual Integration

Integration level II is called *virtual integration* and forms the next encreased degree of database cooperation. It is based on the idea that *views* (derived/computed classes, external schemas [SLT91, TS93]) can be used to build a uniform, virtual interface over multiple databases. This step follows naturally from the previous discussion on querying composite schemas. At level II, we are able to define views, spanning multiple CDBSs and therefore defining persistent links between component systems and/or combining classes from different systems.

In contrast to schema composition (Level I), a federation dictionary (FD) is required at level II to store global information. However, since cooperation is restricted to virtual integration, the federation dictionary contains meta data, that is, *instance*-**in***dependent* information only, e.g., definitions of multi-database views (i.e. queries). Instance-dependent information, like e.g. object identifiers (OIDs) or object values, must not yet be stored in the federation dictionary at this level. This forms the main restriction of integration level II and prevents from more tight cooperation of CDBSs.

In COOL* for example, the above **extend** query can be used to define a view:

$$\begin{aligned}&\textbf{define view } \mathit{Employees}' \\ &\textbf{as extend}[\mathit{lbooks} := \textbf{select}[\mathit{name}(e) = \mathit{name}(\mathit{lent}(b))](b : \mathit{Books})](e : \mathit{Employees})\end{aligned}$$

Inter-database link *lbooks* from *EmplDB* to *LibDB* is now made persistent, and the definition of the link (the query) is stored in the global FD.

At integration level II, proxy objects from different CDBSs that represent the same real world entity can be unified. As a prerequisite, for any two component databases DB_i and DB_j, it is required that instances of which shall be unified, we are given a query expression that determines, for a given DB_i-object, what the corresponding DB_j-object is (if any). This resembles the requirement for value identifiability [Bee93]: the proxies in the other CDBS can be characterized in an instance independent way by a query.

In COOL* for example, *same*-functions (cf. Definition 2) can be used directly in order to unify proxy objects. At level II, derived *same*-functions from DB_i to DB_j are possible by using **extend** views, similar to the above *lbooks* example. The query expression defining a derived *same*-function is obviously application dependent and can, in general, not be derived automatically.

EXAMPLE 2: To state that objects of class *Students@StudDB* are identical with objects of class *Employees@EmplDB*, if they have identical names, for example, a *same*-function is defined by the following view:

$$\begin{aligned}&\textbf{define view } \mathit{Students}' \\ &\textbf{as extend}[\mathit{same}_{\mathrm{StudDB,EmplDB}} := \textbf{pick}(\textbf{select}[\mathit{name}(e) = \mathit{name}(s)](e : \mathit{Employees}))] \\ &\qquad (s : \mathit{Students})\end{aligned}$$

Notice that the **pick** operator does a set collapse, returning the object from a singleton. It returns undefined if the set if empty, and raises a run-time exception if the set contains more than one object. ◇

After unifying proxy objects in different CDBSs, we now focus on schema integration, that is, to record correspondances among common parts in the local schemas. Recall that schema composition (Level I) constructs a global meta schema as well. Defining correspondences between schemas of different CDBSs, uses the fact that every schema element is represented by a schema object in the meta database (cf. Section 2).

In COOL*, functions are for example unified by defining a *same*-function from meta type $function@DB_i$ to meta type $function@DB_j$.[5]

EXAMPLE 3: To unify functions *name@StudDB* and *name@EmplDB*, the following *same*-function is defined on the composite meta schema of *UnivDB*:

```
define view Functions′@StudDB
as extend[same_StudDB,EmplDB :=
                  pick(select[fname(f) = "name" ∧ fname(g) = "name"]
                              (g : Functions@EmplDB))]
      (f : Functions@StudDB)
```

Notice, that *fname(f)* is a meta function, returning the name of a function, represented by meta object f. ◇

Now, all prerequisites for virtual CDBS integration are defined: we showed (i) how to unify "same" objects over multiple systems, (ii) how to integrate schemas by unification of meta objects, and (iii) how to create multi-database views,

EXAMPLE 4: Local schemas are composite by importing *LibDB*, *StudDB*, and *EmplDB*. Then, class *Students* is extended with a *same*-function, and meta class *Functions@StudDB* is extended to integrate *name@StudDB* and *name@EmplDB* properties. Finally, view *Persons* defines a union over the extended classes *Students′@StudDB* and *Employees@EmplDB*, spanning multiple CDBSs (see Figure 4).

```
define schema UnivDB as
   import LibDB, StudDB, EmplDB;
   define view Students′@StudDB as extend ...;             // see Example 2
   define view Functions′@StudDB as extend ...;            // see Example 3
   define view Persons as Students′@StudDB union Employees@EmplDB;
end;
```

The extent of view *Persons* is the union of the base class objects. However, if there would be a customer object and a student object, having equal names, they are defined through the *same*-function to represent the same real world object, and will therefore appear only once in the union view. The type of a union view is the intersection of the base class functions. Since types of *Students′* and *Employees* are disjoint, except for unified functions

[5]Remember, not only the unification of functions, but of any meta object, representing variables, types, classes, or views, is possible.

name@StudDB and *name@EmplDB*, there is one single function, *name*, applicable to these objects. ◇

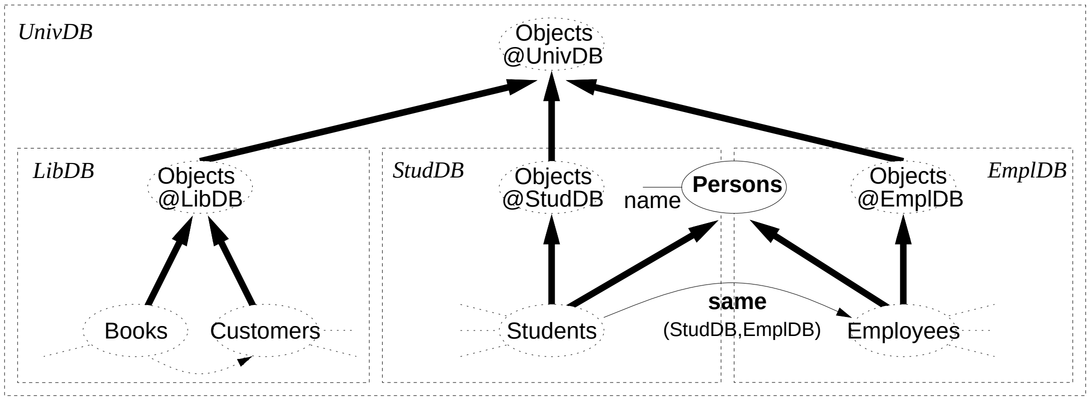

Figure 4: Virtual Integration of *LibDB*, *StudDB*, and *EmplDB*

3.3 Level III: Real Integration

Level III is called *real integration* and forms the next encreased degree of database co-operation without the need of completely giving up local CDBS autonomy. At level II, CDBSs are highly autonomous, since integration is restricted to views only, and therefore, the global FD stores meta information (instance-independent data, e.g. view definitions) only. As a consequence, there are several disadvantages of level II integration. For example, functions with domain and range type at different CDBSs (cf. *same*-functions), have only been allowed, if they are **derived** from a query expression. **Stored** inter-database functions have not been possible, because they would require to store instance-dependent information in the global FD. Furthermore, e.g. variables of a supertype of types of multiple CDBSs are not allowed at level II, since they can store objects of multiple databases.

Level III removes this limitation. In contrast to level II integration, the use of the FD is enhanced to *store instance-dependent* information (e.g. object values, OIDs) as well. This does not say that all objects from CDBSs are copied into the FD. However, if needed, values/OIDs of local objects are stored in the FD. As a consequence, CDBSs are loosing further autonomy, since they must inform the multi-DBMS upon local database updates (e.g. object deletion), in order to insure that copies of values/OIDs of local objects are changed/deleted in the FD as well (cf. keeping multiple representations consistent).

In general, schema augmentation at integration level III is not any more limited to views. In COOL* for example, stored inter-database functions are now allowed.

EXAMPLE 5: Consider again multi-database *UnivDB*. An inter-database function *favourite_book* from *StudDB* to *LibDB* can be defined, which is not derived by a query, but stored explicitely and needs therefore the enhanced FD to store its values:

define function $favourite\_book : student@StudDB \rightarrow book@LibDB$

A special case of that are stored *same*-functions, like e.g.:

> **define function** $same_{LibDB,StudDB} : customer@LibDB \rightarrow student@StudDB$

Assume variable c, holding a customer object from *LibDB* and s, holding a student object from *StudDB*. We can now directly assign (**set**) these objects as being the same:

```
define var $c$ : $customer@LibDB;$
define var $s$ : $student@StudDB;$
set[$same_{LibDB,StudDB} := s$]($c$)
```

Notice, that we really get advanced possibilities, since we do not need to know a query to retrieve *same* objects from other CDBSs. This was not possible at level II. ◇

Additional global schema augmentation possibilities of level III are: (i) to define global object types, that are subtypes of different CDBSs and therefore contain functions from multiple CDBSs, (ii) classes that are subclasses from different CDBSs, and (iii) variables that can hold objects from multiple CDBSs as values. Notice that these global schema elements are only visible to the FDBS and are not known to a local CDBS.

Once CDBSs are really integrated, not only multi-database queries respecting the global object identity are available, but general updates, spanning multiple CDBSs are possible as well. Such updates are again data model/language dependent.

In COOL* for example, there is a generic update operation **gain**[t](o), adding object type t to object o.[6] As long as type t and object o stem from the same database, the **gain** operation works as in one centralized database. However, if o and t are from different databases, the semantics becomes unclear, since an object can usually not get a type from an other database. Let DB_i be the database, where object o is stored and DB_j be the database, where type t is defined. One possible realization of the **gain** operation for multi-databases might work as follows:

```
IF $i = j$ THEN
    gain[$t$]($o$)                    // perform update locally within $DB_i$ (= $DB_j$)
ELSE
    $o' = same_{i,j}(o)$              // find the same-object of $o$ in $DB_j$
    IF $o'$ = undefined THEN          // if there is no same object $o'$ of $o$ in $DB_j$
        $o'$ := new[object$_j$]();    // create a new object $o'$ in $DB_j$
        set[$same_{i,j} := o'$]($o$); // assign $o'$ to be the same object of $o$ in $DB_j$
    END
    gain[$t$]($o'$)                   // perform update on object $o'$ locally within $DB_j$
END
```

This realization maps the multi-database **gain** operation to a sequence of operations, each of which can be executed within one single CDBS. This would not have been possible at integration level II, since an object o' of DB_j is assigned (**set**) to be the *same* object as

[6]Other generic updates are removing a type of DB_j from an object of DB_i, adding/removing an object of DB_i to/from a class of DB_j, setting an object of DB_i as a value of a function or variable of DB_j, creating/deleting objects stored in multiple databases, and migrating objects from DB_i into DB_j.

o of DB_i, and needs therefore the facility of stored *same*-functions, that are only possible at level III or higher.

It is important to understand, that the above global **gain** operation cannot be implemented, using derived (Level II) *same*-functions. To be even more general, although the above realization of **gain** is just one possible way of how to do it, we argue, that there is no other realization of such an operation in any other language, that can be done, using virtual, level II concepts exclusively.

3.4 Summary

Table 1 gives a comparison of the main characteristics of integration levels 0 to IV. At level 0, CDBSs are not integrated at all and therefore, full local autonomy is provided. The only possibility for operations (queries and updates), involving multiple CDBSs, are global (multi-database) transactions. At level I, CDBS schemas are composite and schema names are known globally. Local autonomy is now restricted, since local schema changes must be acknowledged by the multi-database management system. Global (however very limited) generic query and update operations are allowed, spanning multiple CDBSs. At level II, CDBSs are virtually integrated, using e.g. multi-database views. Unification of component objects and schemas is possible. Hence, global query and update operations are much more meaningful. A federation dictionary is available, storing however instance-independent information only. At level III, advanced real integration of CDBSs is available. Augmentation of the global multi-database schema is not limited to derived views. CDBSs are loosing further local autonomy, since copies of local CDBS object data/OIDs are stored in the FD. The design of a language with global query and update operations without any limitations is now possible. At level IV, CDBSs are fully integrated, such that distribution is logically not visible at all.

Table 1: Five Levels of Multi-Database Integration

	Multi-DBS	**Federated DBS**			**Distr. DBS**
	Level 0	*Level I*	*Level II*	*Level III*	*Level IV*
logical schema integration	schemas not integrated	schemas composite	schemas virtually integrated	schemas really integrated	schemas completely integrated
proxy-objekt unification	fully disjoint sets of objects		derived *same*-functions	stored *same*-functions	one set of objects only
global query and update operations	global transactions	restricted global operations	queries using global object identity	updates using global object identity	as in central DBS
federation dictionary (FD)	not necessary	used for instance-independent information only		used for instance-dependent information too	not available

4 Classification of Interoperability Mechanisms

The above interoperability platform might serve as a guideline while designing new multi-database languages or systems. In this section, we concentrate on a second utility, the classification and comparison of related multi-database approaches.

For this purpose, we selected a couple of (well known) multi-database languages (SQL*Net, Multibase, Superviews, VODAK, Pegasus, and O*SQL) and identified their main interoperability mechanisms, i.e. static (schema) and dynamic (operational/language) ones. According to that, these languages are classified into level I, II, or III, in order to show that our classification is helpful understanding and comparing related systems.

4.1 *connect-to*-Statement of Oracle SQL*Net and INGRES/Star

Many relational database system products do offer the possibility to manage a distributed database. Using special software packages, like e.g. Oracle SQL*Net [SQL89] or INGRES/Star [Ing91], they allow for the definition of connections between multiple database systems, making distribution of data more transparent.

After establishing connections to multiple databases, for example by a `CONNECT TO <database>` statement, queries like the following can be written, joining tables from different component databases (`<table> AT <database>`):

```
CONNECT TO BibDB, AngDB;
SELECT Books.title, Employees.name
FROM   Book AT BibDB, Employees AT AngDB
WHERE  Employees.name = Books.lent;
```

The above join predicate (`Employees.name = Books.lent`) is only allowed to compare between basic data types (here: character string). As we stated in Section 3.1, this follows directly from that only basic data types of different CDBSs are unified. Therefore, the above *connect-to*-statement is equivalent to schema composition and hence to integration level I.

4.2 Multi-Database Views in Multibase and Superviews

Multibase [LR82] and Superviews [Mot87] are two MDBSs, providing a uniform retrieval interfaces (no updates) on top of multiple database systems, using global views. Thus, both approaches correspond to integration level II.

Multibase integrates pre-existing databases via view mappings, building global entity types out of local attributes. Queries must be given, describing how global entities and their values are derived from local entities. One may, for example, define that two entities with equal key value globally appear only once (cf. proxy object integration).

Superviews describes virtual integration using a set of integration operations (`meet`, `join`, `fold`, `rename`, `combine`, `connect`, `aggregate`, `telescope`, `add`, `delete`). It does not provide a general view mechanism based on a query language. Thus, together with each integration operation, a transformation of global queries into queries of local classes is defined.

Since Superviews is a level II system, some integration operations are restricted in use. Consider e.g. the operation `add`, augmenting the global schema with a new attribute.

While this is a level III mechanism in general (cf. Section 3.3), Superviews allows only for adding attributes with constant values, which is, in contrast, possible at integration level II, because it compares to **extent** views, defining a new function with constant value.

4.3 Generalizations of VODAK

VODAK [Sch88, NS88] integrates databases via generalizations over classes of multiple CDBSs. To support different semantic relationships between CDBS objects and attributes, multiple kinds of generalizations are identified and enumerated (`data-type-generalization`, `identical-generalization`, `role-generalization`, `history-generalization`, `category-generalization`). All of these special purpose generalizations are equivalent to virtual integration and therefore to cooperaion level II. Consider for example the following VODAK role-generalization:

```
class TAXPAYING-EMPL
   role-generalization-of:
      UNIV-EMPL, COMP-EMPL
   object correspondence rules:
      UNIV-EMPL.SS# = COMP.EMPL.ID#
   attributes:
      BORNON
      identical:
         UNIV-EMPL->BIRTHDATE
         COMP-EMPL->BIRTHDATE
end TAXPAYING-EMPL
```

To show, that this generalization is a level II mechanism, we sketch its reduction to (derived) *same*-functions and a **union** view: First, a derived *same*-function from $CompEmpl\ c$ to $UnivEmpl\ u$ is defined, unifying objects with $ss\#(u) = id\#(c)$:

define view $UnivEmpl'$
as extend[$same$:= **pick**(**select**[$ss\#(u) = id\#(c)$]($c : ComEmpl@DB2$))]
($u : UnivEmpl@DB1$);

Second, functions $birthdate@DB1$ and $birthdate@DB2$ are unified using a *same*-function on the meta database:

define view $Functions'$
as extend[$same$:= **pick**(**select**[$fname(f) =$ "$birthdate$" $\wedge\ fname(g) =$ "$birthdate$"]
($g : Functions@DB2$))]
($f : Functions@DB1$)

Finally, classes are integrated by a **union** view $TaxpayingEmpl$, which is now equivalent to the above VODAK generalization:

define view $TaxpayingEmpl$ **as** $UnivEmpl'$ **union** $CompEmpl$

In COOL*, we require that functions to be unified have identical names, wich is not necessary in VODAK. However, renaming parts of a schema (e.g. functions) can be done at level II (see Section 4.6 below).

4.4 *unifier*- and *image*-Functions in Pegasus

Pegasus [ASD$^+$91, AAD$^+$93] internally describes type and object integration using two system functions: *unifier*(t) defines for each CDBS type t exactly one unified type of the global (federated) schema. *image*(o) returns for each local object o at most one unified global object. The constraint o *instance_of* $t \Rightarrow$ *image*(o) *instance_of* *unifier*(t) must always hold. The default assumption is *unifier*(t) $= t$ and *image*(o) $= o$ and can be overriden by the DBA/user by defining global inter-database types. The following statement, for example, integrates three local types $NStud, EStud, WStud$ from different CDBSs into one global type *Student* (HOSQL syntax [AAD$^+$93]):

```
CREATE TYPE Student
ADD UNDERLYING TYPES NStud, WStud, EStud
UNDER Student
(WStud.Image(x) AS SELECT s FOREACH Student s WHERE ssn(s) = ssn(x))
(EStud.Image AS STORED)
```

Corresponding *unifier* and *image* functions are created automatically by the system. For each underlying type, *unifier* is set to *Student*, e.g. *unifier*($NStud$) $=$ *Student*. For $NStud$ objects, *image* is the default mapping *image*(o) $= o$. For $WStud$, it is a derived mapping, given by a HOSQL SELECT expression.

So far, these are level II mechanisms. However, for $EStud$ the *image* function is a stored function, that is, *image*(o) is undefined until an instance of *Student* is assigned explicitely. As we know, this is a real extension, since it needs for an advanced federation dictionary storing instance-dependent information and requires therefore integration level III. The Pegasus federation dictionary must store e.g. tables, containing mappings between local and global OIDs and the local CDBS must allow to store their objects in foreign systems.

It is interesting to notify, that Pegasus is mainly a level II (virtual) system (derived *unifier* and *image* functions), except of some very few mechanisms, like e.g. stored *image* functions, that are of level III.

4.5 *merge*-Operation in O*SQL

O*SQL [Lit92] is a comprehensive multi-database language, providing e.g. types and functions spanning multiple databases. Such MDBS types and functions can be derived from an O*SQL query expression, resulting therefore in a level II intergration. Whether stored inter-database functions and types augmenting the global schema are allowed as well, is unclear from the available paper. However, such possibilities are serious language extensions, resulting in interation level III and further loss of local CDBS autonomy.

In O*SQL, proxy objects can be unified by a *merge* operation. The following first expression, unifies objects $o1$ and $o2$. The second expression describes a kind of object-unifying join, unifying employees and students with equal $ss\#$:

```
merge :o1, :o2;
select merge(ss#(e) e s) for each Empl e Stud s where ss#(e) = ss#(s);
```

In both cases, a global table of "same" objects must be allocated in the FD. For the first expression, this table holds OIDs $o1$ and $o2$. For the second select-expression, this table stores the result of the query, i.e. the corresponding OIDs. Notice, that the semantics of the select operation is not that of a derived *same*-function, but the result is stored (materialized).

The *merge* operation is a level III mechanism. Whether O*SQL is mainly a level II or III system is unclear, due to the lack of precise, formal definition of the language is not avaliable.

4.6 Discussion – Information Capacity

We presented interoperability mechanisms of some selected multi-database languages, as summarized in Table 2. We say, that a language is called "of level n", if it contains at least one mechanism of level n and none of level n+1. Of course, the enumeration of languages was not complete. We considered those systems, focusing in object and schema issues. Other approaches, discussing for example mainly MDBS transactions, architectures, or data model heterogeneity are not taken into account yet.

Table 2: Selected Interoperability Mechanisms of Integration Levels I – III

Level	*Concepts and Mechanisms*
I	schema composition in COOL* (Sect. 3.1)
	connect to-statement of Oracle SQL*Net [SQL89], INGRES/Star [Ing91]
II	MDBS-views in COOL* (Sect. 3.2), Superviews [Mot87], Multibase [LR82]
	derived *same*-functions in COOL* (Sect. 3.2)
	generalizations in VODAK [Sch88, NS88]
	unifier-functions and derived *image*-functions in Pegasus [ASD$^+$91, AAD$^+$93]
III	stored *same*-functions in COOL* (Sect. 3.3)
	update operations of COOL* [SLR$^+$92]
	stored *image*-functions in Pegasus [ASD$^+$91, AAD$^+$93]
	merge-operation in O*SQL [Lit92]

One may ask, whether there isn't a general notion on how to find out, what kind of mechanism is of what particular integration level. In other words: What do all languages of one level have in common? It shows, that the key to answer this question is *change of information capacity* [Hul86, MIR93].

Definition 3. (Information Capacity) Let S be a given database schema. The information capacity $\mathcal{DB}_S$ is the set of all potential states, a database can take with schema S.

The capacity of a database is therefore given by its schema. Hence, changing the schema of a database may directly have an impact on its capacity. We say, a schema change is capacity *preserving* (CP) / *augmenting* (CA), if it does preserve / augment the information capacity of the database [ST94].

For multi-databases, the *global information capacity* is given by the composite (global, federated) schema, reached by schema composition at integration level I. Any further database (schema or object) cooperation mechanism may now change this global information capacity.

Proposition. *An interoperability mechanism is of level II, iff it preserves (CP) the information capacity of the global (composite) database.*

Any kind of adding derived (virtual) information, like MDBS views e.g. in COOL*, Superviews, and Multibase, generalization of VODAK, derived *same*-functions of COOL*, and derived *unifier*- and *image*-functions of Pegasus, are CP mechanisms and therefore of level II. Furthermore, adding attributes with constant values (cf. Section 4.2), as well as renaming schema elements (cf. Section 4.3) is CP.

Proposition. *An interoperability mechanism is of level III, iff it augments (CA) the information capacity of the global (composite) database.*

Any kind of adding stored and not any more derived information is CA and therefore of level III. Adding stored *same*-functions of COOL*, stored *image*-functions of Pegasus, and the *merge*-operation of O*SQL are examples of CA schema changes. Finally, most of the generic update operations of COOL* (e.g. **gain**) are level III operations as well, because they define implicitely new functions, and therefore augment the global information capacity as well.

5 Conclusion and Outlook

The contribution of this paper is a formal classification of multi-database languages into five levels with increasing strength of database integration. Level 0 represents non-integrated, fully autonomous CDBSs. Level I allows for schema composition. Level II is called virtual CDBS integration. Level III is characterized by real CDBS integration, and at level IV, CDBSs are finally completely integrated. The utility of this classification is twofold:

1. A designer of a new multi-database language is able to understand, what kind of concepts and mechanisms he is allowed to include into his language, in order to build a multi-database system of a particular, desired integration level. As a consequence, local CDBS autonomy and the possibilities for designing global query and update operations are well known.

2. Any multi-database language (e.g. SQL*Net, Multibase, Superviews, VODAK, Pegasus, O*SQL, COOL*, ...) may be classified into level I to IV according to the implemented concepts and mechanisms. This is very helpful to understand related work and to compare systems among each other. We argued for example, that Pegasus and O*SQL are mainly systems of integration level II (virtual integration), however, they include some very few concepts, making them finally level III systems (real integration).

Future work will include more MDBS languages, as well as the consideration of data model heterogeneity and transaction mechanisms. Whereas we think, that transaction mechanisms are orthogonal to the presented classification, it might be interesting to investigate, what kind of data model transformation mechanisms are possible at a particular integration level.

References

[AAD$^+$93] R. Ahmed, J. Albert, W. Du, W. Kent, W.A. Litwin, and M.-C. Shan. An overview of Pegasus. In *Proc. 3rd Int'l Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS)*, Vienna, Austria, April 1993. IEEE Computer Society Press.

[ASD$^+$91] R. Ahmed, P. De Smedt, W. Du, W. Kent, M.A. Ketabchi, W.A. Litwin, A. Rafii, and M.-C. Shan. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12), December 1991.

[Bee93] C. Beeri. Some thoughts on the future evolution of object-oriented database concepts. In *Proc. GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, Braunschweig, Germany, March 1993. Springer, Informatik aktuell.

[BLN86] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4), December 1986.

[Hul86] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3), 1986.

[Ing91] Ingres Corp. *INGRES/Star User's Guide, Release 6.4*, December 1991.

[Ken91] W. Kent. The breakdown of the information model in multi-database systems. *ACM SIGMOD Record*, 20(4), 1991.

[Lit92] W. Litwin. O*SQL: a language for multidatabase interoperability. In *Proc. IFIP DS–5 Semantics of Interoperable Database Systems*, Lorne, Australia, November 1992.

[LR82] T. Landers and R.L. Rosenberg. An overview of multibase. In *Proc. 2nd Int'l Symp. on Distributed Data Bases*, Berlin, Germany, September 1982. North-Holland.

[MIR93] R.J. Miller, Y.E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th Int'l Conf. on Very Large Data Bases (VLDB)*, Dublin, Irland, August 1993.

[Mot87] A. Motro. Superviews: virtual integration of multiple databases. *IEEE Trans. on Software Engineering*, 13(7), July 1987.

[NS88] E.J. Neuhold and M. Schrefl. Dynamic derivation of personalized views. In *Proc. 14th Int'l Conf. on Very Large Data Bases (VLDB)*, Los Angeles, California, September 1988. Morgan Kaufmann.

[Sch88] M. Schrefl. *Object-oriented database integration.* PhD thesis, Technical University of Vienna, June 1988.

[SL90] A.P. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneuos, and autonomous databases. *ACM Computing Surveys*, 22(3), September 1990.

[SLR$^+$92] M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, and M. Tresch. The COCOON object model. Technical Report 193, ETH Zürich, Dept. of Computer Science, December 1992.

[SLT91] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991. Springer, LNCS 566.

[SPD92] S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *The VLDB Journal*, 1(1), July 1992.

[SQL89] Oracle Corp. *SQL*Net TCP/IP User's Guide, Version 1.2*, November 1989.

[SST94] M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[ST94] M. H. Scholl and M. Tresch. Evolution towards, in, and beyond object databases. In K. von Luck and H. Marburger, editors, *Management and Processing of Complex Data Structures, Proc. 3rd Workshop on Information Systems and Artificial Intelligence*, Hamburg, Germany, February 1994. Springer, LNCS 777.

[TS93] M. Tresch and M.H. Scholl. Schema transformation processors for federated objectbases. In *Proc. 3rd Int'l Symp. on Database Systems for Advanced Applications (DASFAA)*, Taejon, Korea, April 1993.

Contents