

# Processing Semi-Structured Data in Object Bases

**Kazimierz Subieta, Jacek Leszczyłowski,**

Institute of Computer Science  
Polish Academy of Sciences  
Ordona 21, 01-237 Warszawa, Poland  
subieta@ipipan.waw.pl  
jacek@ipipan.waw.pl

**Irek Ulidowski**

Research Institute  
for Mathematical Sciences  
Kyoto University  
Sakyo, Kyoto 606-01, Japan  
irek@kurims.kyoto-u.ac.jp

## Abstract

We address the problem of null values and other forms of semi-structured data in object-oriented databases. Various aspects and issues concerning semi-structured data that are currently presented in the literature are discussed in the paper. We propose a new universal approach to semi-structured data based on the idea of absent objects. The idea covers null values and union types and can be smoothly combined with the idea of default values. We introduce a simple model of object store that is similar to the Tsimmis model. In contrast to the main stream of the research, our basic assumption is that semi-structured data are not only to be queried but also processed by an integrated query/programming language. To this end we discuss query language constructs that are relevant to query semi-structured data and corresponding issues in programming languages. The idea follows the stack-based approach to integrated query/programming languages that we have implemented in the LOQIS system. Finally we briefly discuss a new approach to polymorphic typing of semi-structured data that is implemented in LOQIS.

## 1. Introduction

The problem of semi-structured data is closely related to the classical problem of null values in databases and has about thirty years of history. Starting on early papers devoted to network (CODASYL) and relational databases many authors proposed extensions of capabilities of database systems for storing and querying irregular, uncertain or missing information. In particular, [LiSu90a, LiSu90b, Sadr91, VrLi91, VrLi93, WoLe90] propose various extensions of the relational algebra, [CaPi87, Fuhr90, BGP90, BGP92] propose specialized probabilistic DBMSs, [DrCh89, LeLo91, RKS89, RKS91, Yazi90] discuss nested relational algebras dealing with null values and [AKG91, INV91, Libk94] (and a lot of other papers) propose methods of querying disjunctive information or so-called “possible worlds”. In [Codd79, Zani84, Gess90] and other papers three/four/multi-valued logics are proposed. There are other concepts; a survey can be found in [Libk94]. A bibliography on uncertainty management by Dyreson [Dyre95] presents about 400 papers devoted to this topic. Null values received the first-class citizenship in the relational model and became a component of many important notions such as outer joins and universal relations.

Currently relational databases, SQL and their object-oriented counterparts introduce null values and some simple means to query/process them as a standard feature. However, it is rather a common opinion that the problem remains unsolved. Up to now the mentioned above theoretical proposals do not fit well with other features of practical database systems, thus they are not implemented, even in prototypes. Pirotte and Zimanyi [PiZi92] characterize them as “...a great variety of approaches that are not easily comparable, and (...) few practically usable results”. Dubois, Prade and Testemale [DPT88] observe that “...the problem is generally not well understood, and any attempt to incorporate support for null values into an implemented system should be considered premature at this time”. These doubts are actually supported by many professionals, including some of the former proponents of theoretical ideas related to null values (for instance, see [Date86c, DaDa92b, DaDa95]).

A new wave known under the term “semi-structured data” [AQMWW97, Abit97, AbVi97, BDS95, BDHS96, BDFS97, QRSUW95, Semi97, Tsim95] approaches the problem from a different angle (and actually does not refer to null values), but inherits a lot of these problems and doubts. Although one can imagine specialized database systems with capabilities limited to storing and querying semi-structured data (especially in the context of Web-based applications), in general we argue that any database management systems must eventually deal with complete

computational and pragmatic power of user/programmer interfaces. It is difficult to imagine that such systems could not be prepared to make applications on the top of their data structures and query languages. Hence, features related to semi-structured data should be combined with all aspects of database systems: database design, data description, query languages, updating and other imperative constructs, typing, object-orientedness, procedures, views, active rules, transactions, security, catalogs, etc. As a consequence, to avoid eclectic *ad hoc* solutions in further extensions of a system, the support for semi-structured data must be designed with an extreme care concerning conceptual simplicity, minimality, consistency and universality.

Taking into account the above cautions we address the problem of null values and other forms of semi-structured data in object bases, and consider how they are to be manipulated in the integrated query/programming languages. The difference between our approach to the problem and the approaches known in the literature (e.g. Lorel [AQMWW96], UnQL [BDHS96], Rufus, WebSQL, WebSemantics and others) is that we extend the scope of issues related to semi-structured data to the mentioned above aspects of storing, querying and processing semi-structured data.

Our experience concerning processing semi-structured data is rather long. In late 70-ties we implemented the system "The Great Emigration" devoted to historical research. The system recorded data on Polish emigrants (about 15000) after the unsuccessful November Revolution against Russian domination in 1830-31. (Many of them were famous names related to Polish and European politics and culture.) The data were collected from police archives all over the Europe, mostly from France. Each emigrant was described by about 50 attributes, such as *AttendedSchools*, *Opinions*, *StayingPlaces*, *Duels*, *Jobs*, etc. Each attribute might be optional (null valued), repeated and complex; each occurrence of an attribute might be (optionally) associated with dates. To the purpose of this system we implemented a powerful query/programming language, which allowed us to ask non-trivial queries; for example "Identify the groups of emigrants according to the same time of their staying in at least three places". The generic part of this system we upgraded to a universal DBMS called LINDA. This system had several other applications, e.g. for processing medical data. Some elements of it are described in [Subi83]. It may be interesting to note that the LINDA data model was almost identical with the Tsimmis data model (see [Subi83, Subi85]). The successor of this system is the LOQIS prototype [SMA90] having powerful capabilities to store, query and process semi-structured data (much more powerful than Lorel and UnQL); the system has been used to make several expert systems. The data model and a query language of this system are described in [Subi91, SBMS93, SBMS95, SKL95]. Currently our group is developing a next version of LOQIS, which will be integrated with the Web. Majority of the discussion of this paper is related to the solutions in LOQIS and in its successor.

## 2. From Null Values to Semi-structured Data

### 2.1. Null Values

From the practical point of view null values are only loosely related to the problem of uncertain or incomplete information. The association was made by hasty theoretical conclusions rather than by facts stemming from data processing reality. (See examples of sources of null values in practical systems, presented in Section 2.5.) Following the current trend, instead of "incomplete/missing/uncertain information" we will use the term "semi-structured information", which better expresses the attitude to the problem. It concerns situations when particular information does not fit exactly the predefined data format, or when some spots in the storage media (or in their formal or conceptual model) are not filled in by meaningful data. The interpretation of the fact that a particular data spot is unfilled is a data semantics issue outside of formal models.

Actually, SQL adopts such attitude to null values as described above. The meaning of SQL null values is not predefined - rather it is a technical trick which can be used for different purposes (concerning conceptual data semantics) and informally interpreted by designers, programmers, database administrators and other users. Because of many sources of null values and many reasons for which they are introduced it is difficult to imagine that another, more semantically specific approach to null values makes a sense. Also default values [Date86c, DaDa92c, DPT88, Gess91], proposed as an alternative to the SQL solution, follow the same assumption. We support Date's and Darwen's arguments (see the hot discussion in Database Programming & Desing; we skip citations) that approaches to null values based on three, four or multi-valued logics are misleading, especially if one considers not only a (usually over-simplified) query language, but a complete application programming interface.

## 2.2. Unions

In the domain of PLs there is another well-known feature dealing with semi-structured data. It is called “variants” in the Pascal family of languages, or “unions” in the C family. (In the following we will use the term “unions”.) Unions and null values are conceptually similar notions. If a null value we interpret as an absent attribute, than the assumption that an attribute A can be null-valued might be modeled as a union of record types  $R(A, B, C, \dots)$  and  $R(B, C, \dots)$ . The notions of null values and unions are, however, not equivalent. For example, if some record type involves  $n$  attributes that can be null valued, then the number of corresponding components of a union is  $2^n$ . On the other hand, the union of  $R(A, B, \dots)$  and  $R(A, C, \dots)$ , where A,B,C are distinct attributes, cannot be precisely modeled by null values. (If in  $R(A, B, C, \dots)$  we choose B and C to be null-valued for modeling the union, we obtain the union of  $R(A, \dots)$ ,  $R(A, B, \dots)$ ,  $R(A, C, \dots)$  and  $R(A, B, C, \dots)$ .) Unions cover also the situation when names of attributes are the same but types are different. In this context it is interesting to note that the PL community noticed unions and ignored null values, and the database community did *vice-versa*. The reasons seem to be obvious: unions do not fit well to relational data structures, and null values are a risky feature for strong typing. Another interesting observation is that in the PL field unions were not associated with incomplete information and caused different research, mainly concerning conceptual modeling, (polymorphic) typing and efficient implementation.

Unions are proposed in IDL of OMG [CORBA95], with an explicit discrimination attribute, and slipped from the OMG model to the ODMG model [ODMG97]. Unfortunately, the current version of ODMG (2.0) contains no construct to deal with unions in the query language OQL and contains no suggestion how unions will be treated by the assumed strong typing system. The issue is not trivial, especially concerning unions with an explicit discrimination attribute. As far as we know, actually there is no powerful query language consistently dealing with unions.

## 2.3. Semi-Structured Data in Object Models

The object-oriented literature tends to ignore semi-structured data. We checked (besides the mentioned Dyreson’s bibliography) major database journals and conference proceedings in the field covering last several years, and made an extensive search in the bibliographic WWW database (<http://www.informatik.uni-trier.de/ley/db/index.html>). In the result we have discovered items [BDW90, GYPB92, INV91, Lenz91, Motr90, TYI89, VrLi91, VrLi93, Zica90], presenting continuations or generalizations of the ideas known from the relational school. They make no comment concerning how they intend to avoid very low practical impact that plagued their relational predecessors. Null values and unions are not mentioned in the object-oriented manifesto [ABDD+89] (neither in the competitive Third Generation Manifesto [SRLG+90] although it considers unions as a requirement for typing systems). They are not present in classical textbooks on object-orientation, e.g., [Kim90, ZdMa90, BDK92, Loom95, Simo95], nor in books devoted to object-oriented analysis and design (e.g. [CoYo91, Mart93, RBPEL91]). A quite complete dictionary of object-oriented terminology [FiEy95] presents the term “null” (“=“nil”) as a value of non-initialized variable, and explains the term “union” as “a low-level construct for saving storage”; i.e., these irregularities are not considered conceptual. In the OMG object model [CORBA95] and the ODMG proposal [ODMG97] every attribute of an object can be assigned the distinguished value “nil” (no value); in relational systems such a freedom can be limited by a *not null* clause. (See also the comment above concerning unions.) There is very little information how these features have to be served by advanced query constructs. ODMG assumes a very restrictive approach in which any non-typical occurrence of a null value in a (sub-) query argument causes a run-time error. This can be considered a step back in comparison to SQL, but on the other hand ODMG radically cuts the semantic problems related to null values.

This small attention to the issue is probably caused by the fact that object-oriented database models assume programming in classical PLs, which do not provide capabilities for dealing with null values and where querying is (unjustly) considered less essential. Some reluctance can also be caused by the fact that object-oriented models assume strong typing, which becomes more difficult in the presence of null values and unions.

## 2.4. Semi-Structured Data in Theories

The fundamental defect of relational theories w.r.t. null values is the *scope mismatch*. The phenomenon is similar to the impedance mismatch and concerns the difference between the scope of theories and the scope in which null values need to be considered in practice. Theories usually address an idealized query language with very limited capabilities (e.g., SPJ queries in the relational algebra, or the language of some predicate logic). The scope for null values is much wider and should include:

- all advanced query constructs, such as grouping, aggregate functions, quantifiers, ordering;
- imperative constructs based on queries: creating, updating, inserting, deleting;
- the interface for embedding a query language into a programming language;
- database semantic enhancements: views, database procedures, integrity constraints, deductive rules, active rules;
- object-oriented extensions: ADT-s, types, classes, interfaces, encapsulation, inheritance;
- static typing systems;
- privacy and security;
- transaction mechanisms;
- interfaces for end users and programming interfaces: graphical query languages, 4GLs, and other.

The scope mismatch is the main reason for a very low applicability of the theories addressing null values. From the position of designers and vendors of DBMS it is very risky to introduce some particular solution without a clear view how it could be expanded to the whole conceivable database environment. The theoretician have proposed to Mount Everest climbers sophisticated equipment for the first 1000 meters of the approach; but DBMS designers well know that the mountain has 8848 meters and real problems appear above 5000 meters.

Unfortunately, the situation concerning the theories in the post-relational era is not better. Semi-structured data obscure and perhaps compromise current theoretical concepts, which are claimed to be relevant for object-oriented models and their QLs. Null values and other irregularities in data are not addressed from the start in these theories. Thus, when eventually the attempt is made to incorporate them into a given theory, it becomes apparent that it usually requires redeveloping the basics of the theory. This eventually could be possible; for example, there are attempts to introduce null values to comprehensions [BLSTW94]. But, in our opinion, a query language alone is insufficient for many kinds of databases users. A query language makes a sense only in combination with other database functionalities, in particular, with application programming interfaces. Hence, providing the theories will be implemented, semi-structured data will impose *ad hoc* solutions in all functionalities that are not covered by the theory. The practice concerning relational systems has shown that such solutions are hardly to be consistent (see later comments on SQL).

There are also special theories devoted to the problem, such as those based on querying possible worlds (e.g. [AKG91, INV91, Libk94]) or special nested relational algebras (e.g. [DrCh89, LeLo91, RKS89, RKS91, Yazi90]). Again, due to the scope mismatch the usefulness of these theories in practice is questionable. There are also doubts concerning utopian assumptions; this concerns, in particular, the theoretical concepts related to querying possible worlds (see further papers presented at such conferences as PODS, SIGMOD, ICDT, DOOD, and journals such as TODS and Fundamenta Informaticae; we skip citations). Besides the above flaws it is difficult to consider many theoretical proposals as practically relevant to object-oriented databases because they neglect such fundamental concepts as object identity, types, classes, methods and inheritance. In our opinion, there are doubts if any existing theoretical idea is able to support the designers of practical object DBMS systems w.r.t. semi-structured data.

## 2.5. Do We Need Data Irregularities in Databases?

At the hight of popularity of the relational model the above question might have exacerbated many researchers. It was a common belief that null values are an important and inevitable feature of relational data structures. Many papers devoted to the relational model contained a large number of examples showing the necessity of the concept, and many proposals for its thorough treatment.

The theoretical research has considered some particular sources of semi-structured information, but in practice there are many such sources. None of the cases mentioned below can be qualified as uncertain information; rather, they are related to specific aspects of data storing and processing.

- Information is irrelevant, for example, “nickname” for a person who has no nickname.
- Information is known, but not filled yet because data input lasts some time. For example, a new employee is hired, all his/her data are already introduced to the database, except “salary” which has to be input by another administration division.
- Information is valid only for fixed time; after it must be nullified (to prevent errors) and then input again or recalculated; for example, weather or stock market forecasts.

- Information is known and could be filled in, but it is considered inessential for the particular domain of applications. For example, in some airline ticket reservation system the nationality of passengers is not filled in for domestic flights.
- Information consists of “legacy” records, with limited amount of information, and actual records having an extended set of attributes. Legacy records have null values, but this is not “incomplete information”, because in the past the information was irrelevant or unnecessary.
- The result of some processing is an intermediary data structure containing null-valued elements; e.g. outer joins.

Nowadays, there are several reasons to deal with semi-structured data. The following aspects of database technologies may require them:

- Object-oriented analysis and design. In many cases the admission of null values, unions and repeating data presents an important facility for the analyst or designer. For example, the designer can stick two classes into one introducing some null-valued attributes; and vice versa.
- Schema evolution. As a result of schema evolution some new attributes may appear, some may disappear, some may change type, some single attributes can be changed into repeating ones. This naturally leads to unions and optional data.
- Interoperability. Heterogeneous and/or distributed databases require “resolution of missing or conflicting data values that occur when semantically identical data items may have some attribute values different of missing in some data sources” [Brei90]. Similar arguments can be found in [Kent91].
- Web, the most influential achievement of current computer technology. It has broken a seemingly firm stereotype according to which databases must be well typed, formatted, documented and consistent. WWW databases are untyped, weakly formatted, with dangling links as a rule, etc. There is a lot of valuable research devoted to querying semi-structured data on the Web; however, there is little doubt that they will require efficient programming techniques to deal with their irregularities and inconsistencies (e.g. for implementing active agents).
- Data warehouses, which assume collecting decision-support data from various sources. There are two tendencies. The first one, related to OLAP and “data cubes”, assumes very regular and formatted data, originating mainly from relational databases. The second one assumes collecting information concerning a particular topic (e.g. the stock market) from various heterogeneous sources (e.g. Web). Analytical processing of such heterogeneous information (statistical analysis, overview reports, data mining, etc.) requires capabilities of integrated query/programming languages that will make possible to query and process absent, alternative or repeating information.

## 2.6. Null Values in SQL

While professionals easily agree that null values and other data irregularities cannot be avoided, the question concerns functionalities in database languages that are appropriate to serve them. The current view on this issue is materialized in relational databases and SQL. A null value can be stored in a database and returned by an SQL expression if it cannot be evaluated correctly. This may happen because of null-valued arguments of an operation, as well as in the case of wrong arguments of operations. For example, function *sum* returns *NULL* for an empty argument table. SQL does not allow explicit comparisons of nulls and ordinary values but involves a special predicate *is [not] null*. A special function *if\_null* returns an attribute's value if it is not null, or some constant value otherwise. Comparison of an expression which returns *NULL* with any other value returns a third truth value *UNKNOWN*; in the *WHERE* clause, however, it is equivalent to *FALSE*. In embedded SQL the admission of null values requires for every such attribute two variables in an underlying host language program. An extra *indicator variable* is used to store boolean information determining if the particular value of an attribute is null or not.

The above presented solutions have their flaws. Several authors (notably Date and Darwen [Date86, DaDa92a, DaDa95]) point out difficulties related to null values, which make the semantics of user/programmer interfaces obscure and inconsistent. Date [Date86] presents a severe criticism of null values in the relational model (“...the null value concept is far more trouble than it is worth”, “...the SQL null value concept introduces far more problems than it solves”). He gives many striking examples of anomalies implied by null values. For instance, although in SQL “officially” every null value is distinct (thus  $A = B$  returns *UNKNOWN* if both *A* and *B* return nulls), the *group by* and *unique* operators treat them as identical, and aggregate functions *sum*, *avg*, *min*, *max*, *count* totally ignore them. Another striking example: if relation *R* contains numerical attributes *A* and *B* which could be null valued, than in general *select sum(A+B) from R* may return the result different from *select sum(A) + sum(B) from R*. We can easily

imagine very difficult bugs generated by such an inconsistency. Date concludes that these flaws are not only the property of an inadequate SQL design, but they are the inevitable consequence of the idea of null values.

The descendant model of SQL also promotes the use of null values. The SQL3 working draft [ANSI94] proposes to introduce an arbitrary number of application specific null values such as “Unknown”, “Missing”, “Not Applicable”, “Pending”, etc. [Gall94]. It is not clear, however, how this feature will be reflected in the semantics of particular query/manipulation constructs. Because of the assumed compatibility with the SQL-92 standard we cannot expect that the above mentioned flaws will be removed. Rather, there is a suspicion that new inconsistencies will be introduced to various SQL3 ideas (ADTs, inheritance, user-defined row types, reference types, collection types, procedures, quantifiers, path expressions, triggers, control statements, temporal capabilities, families of tables, stratified deductive rules, etc.).

## 2.7. Default Values

As an alternative to null values Date and other authors [Date86c, DaDa92c, DPT88, Gess91] propose the concept of *default values*. They are ordinary values that are filled in into a tuple in the case of absent information. Default values can be determined in a relational schema. For example, the schema

```
declare SUPPLIER relation
  SNO      char(4)      nodefault,
  SNAME    char(15)    default( “      ”),
  STATUS   int         default (-1 ),
  CITY     char(20)    default (“???” ),
primary key SNO;
```

presents defaults for attributes *SNAME*, *STATUS*, and *CITY* as a string of spaces, integer -1, and a string of three question marks, correspondingly.

An advantage of default values over null values is that they need not any special options in a query/programming language to process them. However, default values do not solve all problems. When numerical domains are concerned it may happen that there is no good default value. In such cases the database designer is forced to solve the problem in another way, e.g., by introducing an additional column of a table (c.f. the “hidden byte” approach of SQL). Default values are also error-prone. Assuming -1 is a default for salaries of employees, if one apply directly the function *avg*, then the result will be inconsistent. One need to use more complicated statements such as

```
select avg(SAL) from EMP where SAL ≠ -1
```

If the programmer forgets about this then there will be no indication that the result is wrong. As a convenience Date proposes special aggregate functions that automatically omit default values (similarly as SQL aggregate functions omit nulls). This proposal associates two unrelated features, thus the PLs’ orthogonality would be violated. Moreover, it is unclear how such functions can be consistently defined if their argument would be calculated by some queries.

Unions pose another problem: the motivation for introducing them, in contrast to nulls, concerns saving storage space. Default values, which consume the space as ordinary values, could be unacceptable. Moreover, for complex or pointer-valued attributes (links) there may not be any acceptable way to determine default values. Taking into account all of the above difficulties we conclude that the idea of default values solves some problems related to semi-structured data, but not all.

## 2.7. Current Approaches to Semi-structured Data

In comparison to classical database models (network, hierarchical, relational, object-oriented) the main novelty of the approaches to semi-structured data is that data names are kept together with data values. A data structure is viewed as a graph with nodes representing some values (or internal identifiers of data structures) and labeled directed edges representing data names (i.e. logical access paths). Every instance of some conceptual entities can be assigned different attributes. In particular, in Tsimmis [Tsim95] objects are triples  $\langle identifier, label, value \rangle$ , where *identifier* is a unique internal object identification, *label* is a data name, and *value* is some atomic value (*integer*, *string*, etc.) or a set of objects. The database is a pair  $\langle O, N \rangle$ , where O is a set of objects, and N (a subset of O) is a set of entry points to the database. A similar approach (but without identifiers) is assumed in [BDS95, BDHS96, BDFS97]. Almost identical

assumptions (but a bit more general) are presented in [Subi78, Subi83, Subi85, Subi87, SuMi87, SuMi88, SMA90, Subi91, SBMS93, Subi94, SBMS95, SKL95] and several other papers.

Essentially, this view of data structures is not new and is not revolutionary. In [Subi87] we argued that data structures of practically any data model (hierarchical, network, relational) on some abstraction level can be represented in this way. This view of data structures was also considered during the relational model development, e.g. to express such relational operators as a natural join. The essential difference concerns the attitude to typing and to the representation of data. Semi-structured data assume no type constraints. For example, every instance of objects *Person* can be assigned a different collection of attributes. Moreover, the information on data representation is kept together the data itself. These assumptions create a new quality concerning the freedom in changes not only values of attributes, but also in dynamic insertion new attributes or removing them. We note that similar assumptions are taken by OMG CORBA [CORBA95] in the *Property Service*.

While type-less structures have their advantages (dynamic flexibility) and disadvantages (error prone, storage overhead), there is a bit misunderstanding concerning the terms “type-less” and “schema-less”. We can easily imagine a database without types if the information on representation is kept together with data objects. The persistent object store based on this assumption is implemented by the first author [Subi94] mainly for the purpose of expert systems which require flexible data structures. However, “type-less” does not mean “schema-less”. An important aspects of types concerns conceptual modeling. Types not only constraint data: they also inform the user what the database contains; i.e. they represent informal data semantics. It is rather uncommon that the user sees the whole database during writing queries or programs acting on it and is able to predict precisely how it could be changed. The user must be informed what the database *could* contain by a sufficiently precise statement. This statement presents the database schema. It is possible that the schema is not implemented in the system (thus do not present a typing constraint), but it must be known for the user for correct formulation of queries.

Moreover, such a schema is also necessary for writing wrappers that convert plain text data (e.g. stored in a file or in Web pages) into a structured form of labeled hierarchies or graphs. For example, the designer of a wrapper must take decisions concerning data names (labels of graph edges). It is difficult to imagine that such decisions are not supported by a data schema. We note that this misunderstanding of the role of a schema caused illogical assertions in some papers devoted to semi-structured data; we gently skip citations.

Assuming that a formal schema is an inevitable property of any data store, the question is what a new quality is introduced by semi-structured data. The answer is the following:

- A schema should be able to express the fact that some data are optional and may not be present in a particular data instance. This issue is known as “null values”.
- A schema should be able to express the fact that some data are alternative. This issue is known as “variants” or “unions”.
- A schema should be able to express the fact that some data can be repeated some number of times, starting from zero. The issue is known as “repeating groups” or “collections”.
- The schema language should allow orthogonal combination of the above features on any data hierarchy level.

We argue that the problems related to semi-structured data are well-known for many years. This does not mean that they are solved. Despite a lot of research, the proper treatment of them in query/programming languages is still an open problem. Query languages developed specifically to this end are at the beginning of a long way. In this paper we present some alternative paths (in our opinion, more powerful) of further development.

## 2.8. Summary of Our Approach

The approach that we advocate is based on the concept of absent objects. The idea is to treat null-valued data as absent, non-existing. Consequently, any binding to such a datum results in a “nothing” entity rather than in a special value “null”. Although this looks as a very subtle change, there are essential consequences. We show that the idea makes it possible to overcome difficulties encountered in SQL and can be smoothly combined with unions, default values and object-oriented concepts. It can also be consistently incorporated in query languages developed in the spirit of OQL of ODMG, and can be smoothly adopted by programming abstractions and interfaces. As far as we are aware this model is the first, simple, intuitive and universal idea, which is capable to express uniformly all features related to semi-structured information and which is acceptable to an average database engineer. The idea is consistently incorporated into data structures, a query language, programming interfaces and other capabilities of the experimental

object-oriented system LOQIS [SMA90,Subi91,Subi94]. The LOQIS query language [SKL95] is much more powerful than Lorel and UnQL. Moreover, it has very precise, intuitive semantics and is seamlessly integrated with programming constructs, programming abstractions and object oriented notions.

The characteristic points of our approach are as follows:

- Following our experience and suggestions of others, e.g. [Buff92, Date86b, Date86c, DaDa92b, DPT88] we propose that a special null value, understood as a generic property of a database system, should be avoided.
- Default values [Date86c, DaDa92c, DPT88, Gess91] are useful but do not solve the problem of semi-structured data.
- Our approach unifies null values, unions and repeating data. These cases irregularities are modeled by absent objects.
- Every kind of objects (atomic, complex, reference objects) can be absent. Such irregularities are served by standard query facilities.
- The query/programming language for semi-structured data is based on the stack-based approach, where the classical environmental stack is used to express the semantics of query operators in terms of the naming, scoping and binding mechanism.
- We present functionalities that make it possible to achieve an effect that is referred to as “outer join”.
- Default values are properties of classes; they are inherited or overridden by the class members.
- We assume orthogonality of unions and nulls and propose techniques of typing semi-structured data.

### 3. An Abstract Object-Oriented Store Model

A similar idea of a store model has been independently adopted in the Stanford Tsimmis project [Tsim95] as a “lightweight” model, which is able to capture in a uniform way various situations that can occur in heterogeneous data repositories. In comparison with Tsimmis our model additionally contains pointer objects and object-oriented concepts.

#### 3.1. Basics

We repeat some definitions from [Subi91, Subi94, SBMS95,SKL95]. A store contains a collection of *store objects* (*objects* for short) which can be volatile or persistent. There are three components of an object:

- its *internal identifier* (the identifiers cannot be directly written in queries and are not printable);
- the *external name* invented by the programmer or database designer that can be used to access the object from a program;
- the *content* of the object which can be a value, a pointer or a set of objects.

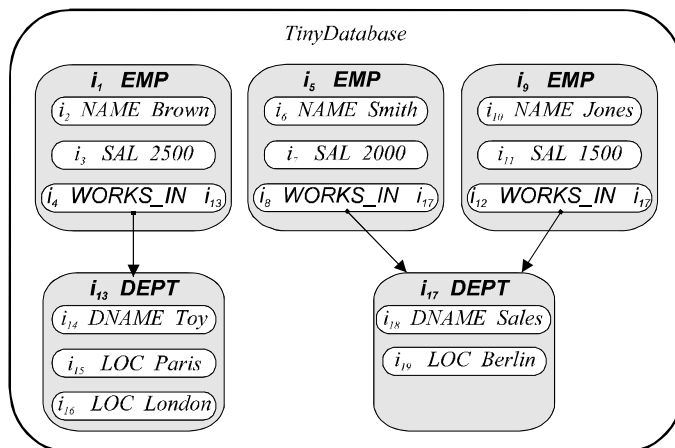


Fig.1. An example database

We refer to these types of objects as *value objects*, *pointer objects*, and *complex objects* respectively. A *store* is a set of objects and a set of identifiers of designated *root* objects. Note that the concept of a value we consider very generally; in particular, it can be a number, a string, a text, a graphics, a method, etc. Fig. 1 depicts an example store. The root objects of the store are identified by  $i_1, i_5, i_9, i_{13}, i_{17}$ . Henceforth, in order to simplify the figures, we will usually omit object identifiers assuming that every depicted object has the unique internal identifier.

### 3.2. Optional, Alternative and Repeating Data

We do not impose constraints which require that objects with the same name should have the same structure. This allows to represent optional, alternative and repeating data uniformly. Optional data (null values) are represented by the absence of a corresponding sub-object. Unions are represented as different collections of sub-objects within the same object type. Repeating data is represented as several sub-objects having the same name. This is illustrated in Fig.2. This approach makes it possible to deal with absent complex sub-objects. A complex repeating attribute *PREV\_JOB* (previous job) of *EMP* may occur any number of times, including zero; inside this attribute we have an optional sub-sub-object *WHEN*. There is a problem of a typing system dealing with such irregularities: this will be discussed later. As we stated previously, we do not associate *a priori* any conceptual semantics with absent, alternative or repeating data. Such capabilities are purely technical means, which can be used for different purposes.

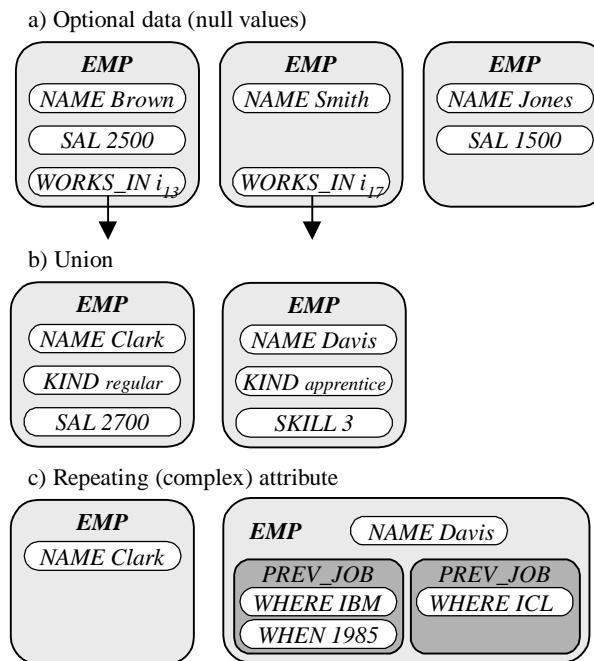


Fig.2. Optional, alternative and repeating data

### 3.3. Object-Oriented Concepts

#### 3.3.1. Encapsulation and Information Hiding

As indicated, complex objects may consist not only of passive sub-objects, but also of procedures, functional procedures, rules, constraints, etc. As in Modula-2, we assume that any kind of such a component can be exported outside the object, as *NAME*, *WORKS\_IN*, *Age*, *ChangeSal* and *SalNet* in Fig. 3, or can be private, as *BIRTH\_DATE*, *SAL* and *Tax* in Fig. 3.

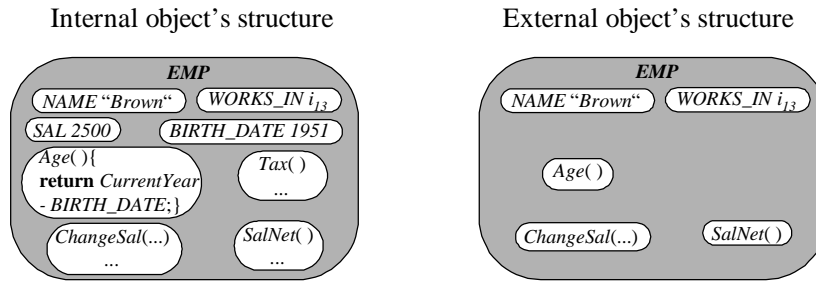


Fig.3. Internal and external object structure

The approach follows the principle of *orthogonality of encapsulation to a kind of object*. Object-oriented models usually assume that objects export *methods* and all attributes of objects are private. Each of the attributes has to be served by two methods: one for reading and another one for updating. We argue that this assumption is too restrictive, makes asymmetry in applying generic capabilities of a query/programming language w.r.t. internal and external objects' properties, and (for repeating attributes) it is even illogical [SKL95b].

We extend our basic model, which we described above, by assuming that all components of a complex object are subdivided into *exported* (visible from the object's outside) and *private* (invisible from the outside). A complex object is defined as a quadruple  $\langle i, n, Te, Tp \rangle$ , where  $i$  is an identifier,  $n$  is a name,  $Te$  is a set of exported elements, and  $Tp$  is a set of private ones. Consider  $O = \langle i_o, n_o, Te_o, Tp_o \rangle$  be an object containing a procedure (method)  $p$  as a sub-object. A name occurring inside the body of the procedure can be bound to any direct component of  $O$ , including private components  $Tp_o$ . For example, consider the procedure *Age* in Fig.3. The private sub-object *BIRTH\_DATE* can be bound, thus the query *CurrentYear - BIRTH\_DATE* is valid. It is invalid outside the object, because *BIRTH\_DATE* is not exported. In general, private sub-objects of  $O$  are visible only within a body of a procedure, which is a sub-object of  $O$ , or is inherited by  $O$ . See the next paragraph.

### 3.3.2. Classes and Inheritance

Fig. 3 illustrates an object *EMP* which contains procedures (methods) *Age* and *ChangeSal*. Other *EMP* objects would also contain those procedures. We may avoid this redundancy by introducing an additional "master" object, which stores all components common to a collection of (similar) objects. We assume that each object in the collection treats the components as its own (i.e. imports them). To make this import possible, we assume that each object  $o$  of the collection has a special *inheritance link* to the master object, Fig.4. We call such a master object a *class*. This understanding of classes (presented in [SMSRW93] and implemented in LOQIS [SMA90,Subi91]) makes the scoping and binding rules semantically clear and more intuitive. This approach easily incorporates optional, alternative and repeating data.

In the rest of the paper we depict inheritance links as dashed arrows. As seen in Fig.4, classes form a hierarchy and inheritance of invariants is transitive: an *EMP* object inherits invariant properties from *CLASS\_EMP* and from the super-class *CLASS\_PERSON*. In [SKL95b] we have shown how scoping and binding rules have to be modified to deal with such structures. Note that overriding appears naturally as a side effect of these rules.

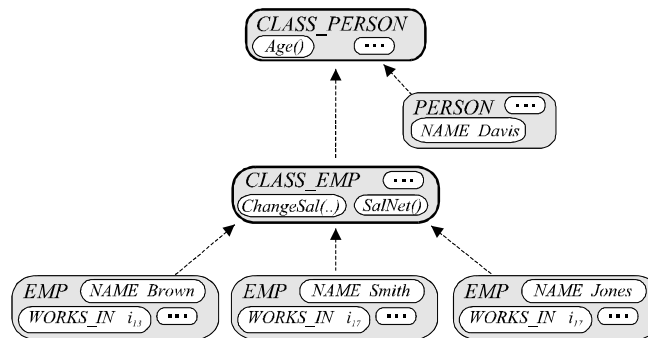


Fig.4. Objects, classes and inheritance

The above approach is also convenient to deal with many features of object-oriented structures such as roles, object migration, attributes having own classes, and multi-inheritance. Another structure accomplishing the objects discussed in Fig.4 is given in Fig.5. In the latter figure *PERSON* and *EMP* objects are separated; an *EMP* object inherits attributes together with values from a *PERSON* object. This view of object-oriented structures (known as *delegation*) is more convenient for introducing roles: *EMP* is one of possible many roles of *PERSON*.

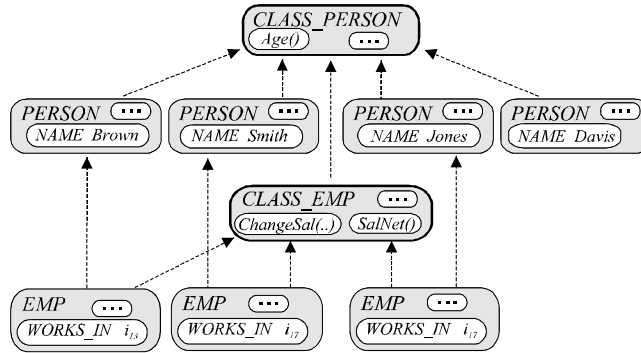


Fig.5. Another view on objects, classes and inheritance

A class may also store some features common to the whole class; for example, a method for creating new objects or methods such as the *number-of-objects*, which act on the whole collection of objects linked to the class (i.e. the class *extent*). These properties should be distinguished from properties inherited by class members. (Inheritance of such properties leads to the concept of a *metaclass*: this is not discussed here due to the lack of space.)

### 3.4. Default values

A class in object-oriented PLs may store a few kinds of invariants which are inherited by the class members. Typically, these are the definitions of attributes (typing information) and the methods (procedures or functional procedures) which are local to classes. However, more kinds of such invariants can be considered in object-oriented databases, in particular, default values, integrity constraints, active rules, export/import lists, access rights, security rules, meta-data and helps. Here we only consider default values. We distinguish two different roles for them:

- Initial values for some attributes, which are used when a new object is created. They are physically copied into a created object, thus need not to be inherited.
- Common default values, which are dynamically inherited by class members and can be overridden by values within a particular member. Such values can be atomic or complex.

In Fig.6 we present the latter kind of default values for the example presented in Section 2.

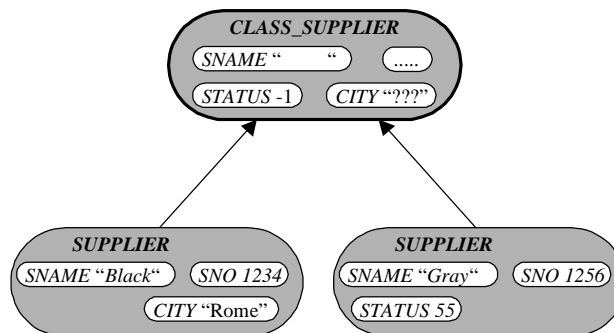


Fig.6. Inheritance and overriding of default values

Default values in the object-oriented approach do not imply special features in query languages, assuming the model and scoping/binding rules as explained in [SKL95a,SKL95b]. However, we must consider a new semantics of assignments (updating). When one updates a value which actually is inherited from a class, the semantics should not update the value, but create a new sub-object with the value determined by the right hand side of the assignment. For example, in the case shown in Fig.6 the assignment

**for each SUPPLIER where SNAME = "Black" do STATUS := 45;**

should result in creating a new atomic object  $\langle i_{new}, STATUS, 45 \rangle$  and then inserting it into the Black's object. (Obviously, updating of class properties via its members should not be allowed.)

The presented definitions of default values capabilities can be easily generalized, for example, in order to cater for cases when an inherited property can be not only a "static" object, but also a functional procedure. Assume that when the status of a supplier is not defined, it is calculated as the number of products she supplies. In this case instead of introducing the object  $\langle \dots, STATUS, -1 \rangle$  to the SUPPLIER\_CLASS we introduce the following functional procedure (a method), where  $SP(SNO, PNO, \dots)$  denotes the relation connecting suppliers and parts:

**integer procedure STATUS { return count((SP as x) where x.SNO = self.SNO); };**

There are many examples of situations in the conceptual database design, when such dynamic defaults might be useful. As far as we know they have not been considered in the database literature.

Note that the procedure, like all inherited procedures, is evaluated within an environment of a SUPPLIER object, thus the second occurrence of SNO in the procedure's body is properly bound to SNO of SUPPLIER (what is indicated by self). Semantically, the construct SP as x creates so-called binders, i.e., named values or identifiers. For detailed semantics related to binders see [SKL95a].

### 3.4.1. Default values and scoping/binding rules

Default values stored within classes require special attention concerning scoping/binding rules. In some situations it is a difficult to determine consistent scoping rules that obey natural expectations of the programmer. These expectations can be summarised in two points:

- First visit the internal properties of an object; if there is no proper attribute, then look for the default within its class, superclass, etc.
- During execution of a method first visit its internal environment, then visit its class, superclass, etc. looking for local properties; next, visit the internal properties of an object being a receiver of a message.

These rules are sometimes contradictory. For example, assume that a method  $m$  is stored within class  $c_1$  and refers to an attribute  $a$  stored within object  $O$ ;  $O$  is an instance of the class  $c_2$ , and  $c_2$  contains sub-object  $a$  storing a default value for the attribute  $a$ , Fig.7a.

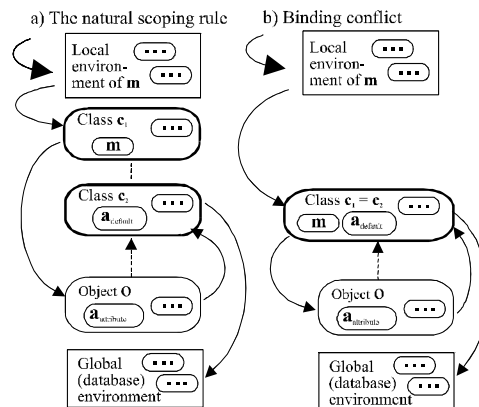


Fig.7. A conflict between natural scoping rules and default values

Assume that the object  $O$  received the message  $m$ , and that within the body of  $m$  there is a reference to the attribute  $a$ . The natural order of visiting particular environments during the binding name  $a$  occurring within the body of  $m$  is shown on Fig.7a as a sequence of curved arrows:

- Visit the local environment of  $m$ ;
- Visit private and exported properties of  $c_1$  (to bind some other methods, class variables or procedures that may be called within  $m$ );
- Visit the sub-objects of  $O$ ;
- Visit exported properties of the classes that  $O$  belongs to; in this case, the exported properties of  $c_2$ . The default  $a$  will be bound here if  $a$  was not bound in the previous step;
- Visit the global environment (database objects, global temporary objects, functions and procedures from available global libraries).

The questions arises what will happen if  $c_1$  and  $c_2$  denote the same class? In such a case the above rules make our idea of default values invalid for the method  $m$ , because the default  $a$  will be visited (and bound) before the attribute  $a$ , Fig.7b.

A possible method to avoid this conflict is an explicit syntactic distinction between attributes (and all inherited properties) of the object  $O$  that  $m$  operates on, and properties of the environment that  $m$  belongs to (or imports). There is several acceptable solutions, for example, the symbol “**self**” preceding an attribute name within the body of  $m$ , or a special preamble within  $m$  specifying names of  $O$  properties that are used within  $m$ .

The above case shows that for object-oriented database programming languages the scoping/binding rules is a non-trivial issue. For more complicated cases, which include the combination of complex attributes, multi-inheritance, roles, modules, ADTs, procedures local to procedures, higher order procedures, export/import lists, attributes belonging to own classes, nesting of method calls within queries (and *vice versa*), renaming of inherited objects, etc. the scoping/binding rules become quite complicated and need careful further investigations.

## 4. Querying Absent, Alternative and Repeating Data

In this section we consider how absent, alternative and repeating data can influence query languages. The presentation addresses the formalized query language SBQL that we have proposed in our previous publications [SBMS93, SBMS95, SKL95a, SKL95b, SKL95c].

### 4.1. Testing Absent Data

Absent data imply the necessity of special care in queries. Consider the query *EMP where SAL > 1000*. If for some employee the attribute *SAL* is absent, then binding name *SAL* will return the empty table, which has to be compared by  $>$  with 1000. Clearly, the operator  $>$  is not defined for this case. We therefore consider the following possible definitions of the semantics:

- Predefined truth value: in this case the formula will return *FALSE*.
- The formula will return the third logical value *UNKNOWN*.
- A run-time error or exception, i.e., the case is semantically forbidden.

The first approach leads to unreliable programs, because, e.g., the “apparently obvious” tautology

$$\text{not } (SAL > 1000) \Leftrightarrow SAL \leq 1000$$

does not hold. The second case requires an addition of many features to the integrated query/programming languages, which (as experience shows) is disproportionately difficult. Hence, only the last case is acceptable, although apparently “non-user-friendly”.

The standard query capabilities that exist in SQL and in the stack-based approach can be adopted to handle the last case. Depending on preferences, the designer or programmer can choose between the options which we present below. For each described capability we list two options. In the first, marked by (a), an identifier of an *EMP* object is included in the result of the query. In the second option, marked by (b), the identifier is excluded.

- Quantifiers:
  - a) *EMP* **where forall** (*SAL as s*) *s* > 1000
  - b) *EMP* **where exists** (*SAL as s*) *s* > 1000
- The SQL function **exists** which tests if the table is non-empty:
  - a) *EMP* **where not exists**( *SAL* ) **or** *SAL* > 1000
  - b) *EMP* **where exists**( *SAL* ) **and** *SAL* > 1000
- The SQL function **count** which calculates the number of rows returned by a table:
  - a) *EMP* **where count**( *SAL* ) = 0 **or** *SAL* > 1000
  - b) *EMP* **where count**( *SAL* ) = 1 **and** *SAL* > 1000

Here the boolean operators **and** and **or** are “lazy”.

The absent data in object-oriented databases may concern not only atomic objects, but also complex ones: our approach can also deal with this case. Obviously, it covers alternative and possibly absent repeating data.

In some cases it is possible to deduce statically that a particular query can lead to a type error because of lack of some its part testing irregularities. Such queries can be rejected by the typing system, or at least generate some warning. This problem requires further investigation and it will be dealt with elsewhere.

## 4.2. Absent Data and Aggregate Functions

Aggregate functions have the same semantics as in SQL: absent data do not influence the result. Consider the query *avg( EMP.SAL )* (the average salary of all employees). For *EMP* objects with no *SAL* sub-object the name *SAL* occurring in the query will return an empty table. This table will be merged with other tables according to the union operator, hence it does not influence the result. This approach is simple, consistent and should be easily understood by the programmers. Consider the Date's example

```
select sum(A+B) from R
select sum(A) + sum(B) from R
```

where the first query may return a result different from the second one. In SBQL the first query cannot be formulated as *sum(R.(A+B))* because if *A* or *B* denote absent attributes the operator + fails. The correct formulation is:

$$\text{sum}(R.((A \text{ as } a) \times (B \text{ as } b)).(a+b))$$

The operator  $\times$  returns an empty table if any of the arguments is absent. Hence, *a+b* is never evaluated with *a* or *b* returning an empty table. The second query in SBQL

$$\text{sum}(R.A) + \text{sum}(R.B)$$

can hardly be confused with the first one.

## 4.3. Capabilities Equivalent to Outer Joins

In the relational model the outer join is an important operator that is relevant for semantic modeling and design methodologies, and convenient for programmers. In many cases the usual join is not satisfactory because it loses information. For example, the name and the department name of an employee can be obtained by the query:

```
select NAME, DNAME from EMP, DEPT where EMP.DNO = DEPT.DNO
```

The resulting table, however, will not contain information on such employees for which the *DNO* attribute is null valued. The outer join enables the user to formulate the query returning a tuple  $\langle emp\text{-}name, NULL \rangle$  for an employee having null valued *DNO* attribute. The operator is usually considered non-primitive, as it can be expressed by other operators of the relational algebra. Date [Date86] advocates the “default style” outer joins which avoids null values.

A similar operator could be also useful in object-oriented query languages. Consider the corresponding query in SBQL:

$EMP.(NAME \times (WORKS\_IN.DEPT.DNAME))$

It will return no information on employees for which the *WORKS\_IN* sub-object is absent. In many cases such a problem is automatically solved by introducing inherited default values. In the presented example the solution is cumbersome, as we must introduce a complex object representing a dummy department, and then introduce to the *CLASS\_EMP* a default *WORKS\_IN* pointer pointing to the object.

We can avoid special options because the existing SBQL operators are sufficiently powerful. For example, assuming that the user would like to return the string “???” for *EMP* objects with absent *WORKS\_IN* the above query can be formulated as:

$EMP.(NAME \times ((WORKS\_IN.DEPT.DNAME) \cup (“???” \textbf{ where not exists } (WORKS\_IN))))$

However, the solutions for more sophisticated cases are awkward. We propose to use a generic functional procedure *if\_absent*, similar to *if\_null* of SQL. The procedure takes any two queries as arguments and returns the result of the first query if the query is not empty, or the result of the second “substitute” query otherwise. Both are arbitrary “union-compatible” queries. In LOQIS the function can be written as the following generic procedure (we assume the macroscopic strict-call-by-value parameter passing method [WaGo84,SKL95c], which combines call-by-value with call-by-reference):

**procedure** *if\_absent*(*Query*, *Substitute*){ **if exists**(*Query*) **then return** *Query* **else return** *Substitute*;};

Thus, the last example can be written as follows:

$EMP.(NAME \times if\_absent(WORKS\_IN.DEPT.DNAME, “???”))$

If one wants to extend this query to deal with salaries which might be null valued, the solution will be the following:

$EMP.(NAME \times if\_absent(SAL, -1) \times if\_absent(WORKS\_IN.DEPT.DNAME, “???”))$

The function *if\_absent* covers all situations which in relational languages require the outer join operator. The function is more general than *if\_null* of SQL - its second argument can be not only constant value but also an arbitrary query, in particular, an invocation of a functional procedure. Such capabilities may be useful for some sophisticated cases which may occur in conceptual program modeling.

#### 4.4. A Problem of Binding

If a data environment is untyped (what may happen in WWW databases or heterogeneous resources) absent objects together with assumed full orthogonality of a query language may cause some binding problems. To explain it, consider the query “Get employees earning the same salary as Brown”:

$EMP \textbf{ where } NAME \neq \textit{“Brown”} \textbf{ and } SAL = ((EMP \textbf{ where } NAME = \textit{“Brown”}).SAL)$

If the Brown's salary is absent the query should cause a run-time error. During the binding of the second occurrence of *SAL* the environment stack is as in Fig.8.

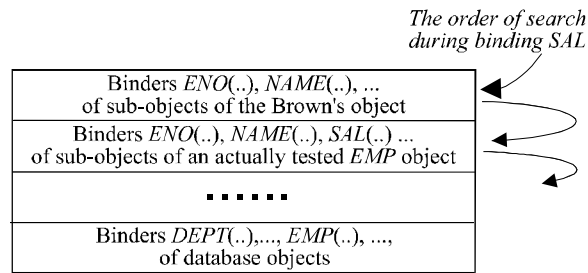


Fig.8. An example of a binding problem

Since the top of the environment stack contains no *SAL* binder the search will be continued in lower sections of the stack. But, the second section contains a *SAL* binder, thus the binding will succeed. As a result, the predicate after the first where will be *TRUE* for any *EMP* having a *SAL* sub-object.

Obviously, the result is wrong. Let us analyze the reason. The second *SAL* occurrence is referring to the Brown's object; if evaluated independently, the nested sub-query (*EMP where NAME = "Brown" SAL*) returns an empty table. However, because the query is nested, *SAL* is considered as referring to the outer query. In terms of our stack model, the search for *SAL* binder should be finished at the top of the stack, but because of absent data there is no such a binder and the search is continued in lower sections of the stack. Hence such a false binding is caused by the lack of information about the semantic qualification of the second occurrence of the name *SAL* in the query. This information should be somewhere given, and should influence the scoping rules and/or structures introduced in the model.

This information is usually present in an object's type. It can be used during compilation to determine statically which section of the environment stack (relatively to its top) is relevant for binding a particular name; so the problem does not occur. Analogously, the problem can be easily solved by introducing to the *EMP* class a special entity named *SAL*, which the only role is to stop the search in lower sections of the stack; the binding of name *SAL* to such objects will return an empty table.

If the data environment is untyped, and there is no possibility to introduce the corresponding information to a class, the programmer must be supported with special facilities to deal with this case. Among various options we present the following. We assume that every name occurring in a query can be augmented by an arbitrary number of names of its attributes. Each attribute name is preceded by a keyword *with* or *without*. For example,

***EMP with NAME with SAL without JOB***

This phrase is evaluated (bound) as a single semantic unit (i.e., *NAME*, *SAL* and *JOB* are not bound independently). It returns the references to all *EMP* objects which are present in the actual scope and satisfy the "with" and "without" conditions: that is, contain the attributes *NAME* and *SAL* but do not contain the attribute *JOB*. Inherited properties are not taken into account.

The described facility gives the programmer a full control over the binding, despite possibly defaults and absent data. For example, assuming *SAL* and *JOB* can be absent, and a default *JOB* is stored within the *EMP* class, the following query

**( *EMP without SAL with JOB* ) where *JOB* ≠ "clerk"**

returns references to all employees containing a job subobject, have no salary and are not clerks. Note that the default jobs and salaries do not influence the result.

## 5. Typing Semi-Structured Data

Current typing systems based on the universal polymorphism [AtBu87, AtMo95, CaWe85, Card89] (parametric, inclusion) are well-developed and valuable theoretical artifacts. However, if a polymorphic typing system is used to express a database schema then, due to its specific syntax and complex semantics, the result can be very difficult to read and understand. Polymorphic type systems imply well-known problems related to multi-inheritance, dynamic

binding, schema/type evolution, overriding of methods and dynamic creation and modification of such database features as views, database procedures, integrity constraints and active rules. Moreover, they are weakly tolerant to other programming interfaces (perhaps, untyped) that may act on the same database.

Persistent polymorphic PLs (Napier [MBCD89], Galileo [ACO85], Machiavelli [OBB89], Fibonacci [AGO95], Tycoon [Matt95], and others) retain the classical one-threaded program control: they provide iterators for bulk types which conceptually assume the one-data-at-a-time processing. Some of them are also equipped with query capabilities, but rather as add-ons to the above mentioned philosophy. In consequence, program expressions and queries are different notions. However, because they cover the same language constructs (e.g.,  $2+2$ ), in some further versions of these languages they should be unified. In contrast, the advocated stack-based approach assumes the many-data-at-a-time processing mode and the unification of program expressions and queries from the very beginning.

Typing semi-structured data presents another difficult problem, especially challenging if we assume processing through query languages. Below we shortly summarize the current view on the topic.

Some authors extend every type with a special null (or nil) element which represents absent or uninitialized value of this type. As Date and others suggest and as follows from our previous considerations, there are strong arguments to reject this idea.

In general, unions make static type checking impossible; hence, it must be delegated to the run-time. A special *union discriminator* (switch) attribute must be introduced, which makes it possible to distinguish dynamically which union case is actually processed. The type checker raises an exception (a run time error) if the program refers to an attribute that does not exist in a particular union case. In this way some type safety is achieved at the cost of performance.

Assuming one-record-at-a-time processing this technique can also be applied to absent sub-objects (i.e., "null values"). In this case every object must store the information telling which attributes are absent. It can be automatically used, analogously to union discriminators, by a fragment of code generated during compilation to detect illegal references to absent attributes. To enable the programmer to write programs with no illegal references a special testing facility should be provided; for example, with/without options considered previously.

The situation becomes different in query languages, mainly because they assume macroscopic processing and require conceptual simplicity. Consider an ODMG OQL query:

```
select * from Persons as x where x.spouse != nil and x.spouse.name = "Carol"
```

If in the particular union case the Persons object does not contain the spouse attribute, then the part `x.spouse` will cause a typing error. The document [ODMG97] contains no explanation how this situation has to be dealt with by systems' developers.

The problem can be solved by the mentioned special facility. However, it is caused by the fact that strong typing does not tolerate situations where a particular name cannot be successfully bound. It follows from our previous discussion that such situations appear in cases of absent (sub) objects, unions and repeating attributes.

## 5.1. Types as Context-Free Grammars

In the LOQIS project we made first steps in developing a polymorphic typing system that would be a trade-off between a full type safety, simplicity and flexibility in the case of semi-structured and evolving data. Our intention was to create simple and flexible types which can express object-oriented database schemata and enable static typing of integrated query/programming languages.

In programming languages types have several roles [CaWe85]:

- They keep information about the representation of data. For example, typing some variable as *integer* means that the variable is stored as 16 bits, with the precise meaning concerning the representation.
- They keep information about the format or structure of data. For example, a record type keeps information about the order and sizes of record's attributes; the record itself is a sequence of bytes with no boundaries between particular values.
- They are components of the binding mechanism. For example, a record type bears the information about names of the record's attributes; these names are required during binding of attribute names occurring in a program to run-time programming entities.

- They present constraints on the use of programming objects within expressions of a language, within procedures, functions, modules, etc.
- Informally, types are used for conceptual modeling of data and programs.

This overloading of type roles causes problems when semi-structured data are involved. Indeed, for unions some information about the structure of a record must be kept outside its type - usually within the record itself, e.g. a switch attribute. In relational databases the information about nulls is kept within each tuple as a “hidden byte”. Moreover, databases can be accessed from many application programming interfaces (perhaps, some of them are untyped, e.g., Smalltalk), while a type is a property of a single language (e.g., C++). In such a case there must be some language-independent information about the representation, structure, order and names of run-time program entities, available for compilers/interpreters of other languages.

These observations have led us to the conclusion that a proper treatment of semi-structured data requires the information about the representation, names and structure of data to be independent of its type. This is an implicit assumption of approaches to unstructured data [Tsim95, BDHS96]; [SiZd96] presents arguments that many database applications require light-weight data models, not constrained *a priori* by a database schema. This in particular concerns Web, scientific databases and data warehouses. Lack of such flexibility is an obstacle in wide use of database technologies for many application domains.

In LOQIS the information about the representation, structure and names of data is kept together with the data itself. Apparently, this assumption implies some additional consumption of the storage. We argue that for modern databases, which include very big data values such as graphic files, this overhead is acceptable. Moreover, it can be optimized by various methods, including some of them that are based on types. On the other hand, there are many advantages of such an assumption. For example, it is possible to leave for the system the decision how particular data has to be represented (e.g. an integer can be represented as a string of digits), or to make an efficient garbage collector which browses the whole data structure from the root via all pointers to other data.

The separation of type from the information about the representation and structure of data makes it possible a new approach to typing semi-structured data. In LOQIS our basic assumption is that a database schema is a *context-free grammar*. Names of declared types are non-terminals in the grammar. To make the grammar more user-friendly, we introduce a special syntax denoting repetitions (iterations, in the terminology of regular expressions) and *options* (unions with the denotation of “empty”). Below we present an example of a type description in LOQIS:

```
PayType = AMOUNT( real ) optional WHEN( date );
```

```
EmpType = NAME( string ) AGE( integer) optional JOB( “designer” or “analyst” or ... )
          (SAL( integer) or optional repeating PAYMENT( PayType ));
```

“**repeating**” means that the data which follows it can be repeated one or more times; “**optional**” means that the number of occurrences of the data is zero or one; “**or**” means a union. The schema

```
optional repeating EMP( EmpType );
```

declares zero or more *EMP* objects. Each of them contains obligatory sub-objects *NAME* and *AGE*, an optional sub-object *JOB* and a union of sub-object *SAL* and repeating complex sub-objects *PAYMENT*. The following is an example of a database defined by such a schema:

```
EMP( NAME( "Clark" ) AGE( 25) SAL( 1000 ))
```

```
EMP( NAME( "Davis" ) AGE( 30) JOB( “analyst” )
     PAYMENT( AMOUNT( 300 ) WHEN( 1995 ))
     PAYMENT( AMOUNT( 500 )))
```

Such schemata can be extended to support structural inheritance, pointer links between objects, ordered structures and the methods, but due to a limited space we omit the details.

## 5.2. Dynamic Data Checking

The above introduced schemata can be used for dynamic data checking. To this end, we implemented a compiler [Subi91] which changes the schema into a non-deterministic checking automaton. Next, we briefly describe the working of the compiler.

Each type declaration is compiled into a separate sub-automaton with the stack-based semantics, hence recursive type declarations are supported. The automaton makes it possible to check the stored data for agreement with their types. Because the automaton is non-deterministic it can be simultaneously in many of its states. The transition from one collection of states into another collection of states is made according to the “signals” from the data scanner. If the automaton is not in any of the defined states, it means the type error; in this case the collection of previous states is used as the error diagnostics. After an error is detected the automaton states are resumed according to some policy. Some implementation peculiarity concerns disregarding the order of object's declarations (and correspondingly, disregarding the order of objects' attributes) on any level of data hierarchy.

Some PL experts argue that dynamic data checking is unnecessary, as the typing systems should be complete; the completeness means that it is impossible to write a correct program generating data values of wrong types. While this view is acceptable for programming languages such as Pascal, we argue that it is too idealistic for database systems, in particular for the following reasons:

- Most databases assume processing through a variety of programming languages and generic utilities; sometimes they are untyped or weakly typed.
- Input data are frequently written to a file which should be debugged in a way similar to programs. Run-time checks (which are in hands of the programmer) make it possible to relate the diagnostic information to this file, not to the place in a program where the type error has occurred.
- Dynamic checks make it possible to integrate type checks with (dynamically declared) integrity constraints and other processing. For example, in LOQIS we implemented a facility for automatic correction of clerical errors in values of enumerated types.
- Failures in data are inevitable consequence of the fact that no system is perfectly reliable. For all those reasons, we believe that the dynamic data checking is very useful in situations of unexpected data inconsistency.

## 5.3. Static Program Checking

The presented idea of a typing system can be extended to static type checking of programs with a kind of universal polymorphism. Here we only explain the basic ideas.

A static typing system requires two mechanisms: (1) type inference rules for all operators in the language; (2) typing input/output characteristics of operators, functions, procedures. Type inference rules are based on the maintenance of a static environment stack which consists of types of objects involved in a particular static scope. Consider, for example, the query *EMP.NAME*. According to the defined above schema involving *EMP* objects we can deduce that the query will return zero, one, or more references to the object named *NAME* of type string; hence its type can be described as

**optional repeating mutable** *NAME*( **string** )

The keyword **mutable** determines that the query returns L-values, i.e., values that can be used on the left side of assignments or as a *call-by-reference* parameter of a procedure. In contrast, when we have

**optional repeating** *NAME*( **string** )

then the result will have zero or more named values (i.e., binders), e.g. *NAME*( "Smith" ).

Next, consider the query *EMP.PAYMENT.( AMOUNT – 100 )*. In this case the user is warned that the query may result in a run time error because *PAYMENT* is only one of the two cases of a union. Then, the inferred type is

**optional repeating integer**

The query  $(EMP.SAL) + (EMP.NAME)$  obviously will result in a type error.

For typing of procedures we extend our schema language with special types *anyname* and *anyvalue*, which represent any name and any atomic value, respectively. Then we define types *anyatomic*, *anyobject* and *anyobjset* (any object set) as follows:

```
anyatomic = anyname( anyvalue );
anyobject = anyatomic or anyname( anyobjset );
anyobjset = optional repeating anyobject;
```

The introduced types enable the programmer to describe program entities with the required precision. This mostly concerns formal parameters of procedures. For example, assume that we would like to write a procedure *Proc* that has as a parameter a collection of references to arbitrarily named objects with attributes *NAME*, *AGE* and an arbitrary number of other attributes. Such parameter has the type:

```
ParType = optional repeating mutable anyname( NAME( string ) AGE( integer ) anyobjset );
```

In general, *anyobjset* may also generate *NAME* and *AGE* objects (thus their cardinalities and types will be undetermined). This can be forbidden by an additional context rule: if some object names are determined explicitly in an object set type then *anyobjset* occurring in this type specification cannot generate objects with these names. If one will consider such context dependencies undesirable, an explicit syntax can be introduced, for example:

```
anyname( NAME( string ) AGE( integer ) anyobjset without NAME without AGE )
```

Assume that a formal parameter of the procedure *Proc* is *F* (a name) and an actual one is *A* (a query). The type of *F* is declared explicitly, and for each procedure invocation the type of *A* can be statically inferred according to the type inference rules. Then, the type of *A* has to be compared with the type of *F*. Both types are grammars defining some formal languages (type extents)  $L_A$  and  $L_F$ . If  $L_A \subseteq L_F$ , then the situation is correct: any actual parameter satisfies the type of the formal parameter. If both  $L_A \cap L_F$  and  $L_A - L_F$  are non-empty, then the programmer should be warned about a possible run-time typing error. This is, in particular, the situation when unions and null values appear. If  $L_A \cap L_F$  is empty or it contains only a trivial element (e.g., an empty collection), then the situation is qualified as a type error.

For example, if the type inferred for *A* is

```
optional repeating mutable EMP( EmpType )
```

and *F* has the type *ParType*, then the situation is correct. Note that our procedure can be also applied to objects *PERSON*, *STUDENT*, etc., if they have the *NAME* and *AGE* attributes; hence the typing method covers the inclusion polymorphism.

The comparison of types of formal and actual parameters can be done by unfolding all nested types, and then, comparing type declarations according to names of involved objects, their cardinalities, and types of their values. The algorithm is rather simple and straightforward for non-recursive types. If we assume reasonable limitations to kinds of allowed recursions (e.g., a kind of “stratification” saying that an invocation of a recursive type should be on a lower level of objects’ hierarchy than its declaration), then recursive types do not present conceptual problems.

In general, the problem of intersection of languages defined by context-free grammars is known as undecidable. To avoid problems related to this issue two approaches are possible: (1) restrict kinds of recursion in the grammars; (2) if the type inference mechanism cannot decide if the intersection is empty or non-empty, then conclude there is a type error.

The presented method is more general than the inclusion polymorphism [AtMo95]. In the latter all types the procedure deals with must create a single inheritance hierarchy. Our method has no such requirement. For example, the procedure *Proc* is also correct for parameters of the type

```
repeating mutable PET( KIND( “dog” or “cat” or ...) NAME( string ) AGE( integer ) );
```

although *PET* objects may have no common super-class with *PERSON* objects. In comparison to the inclusion polymorphism our method seems to be simpler, more general and flexible, especially regarding multi-inheritance (in

fact, the method can be seen as a literal understanding of the Cardelli's approach [Card88]), schema evolution and program reuse.

## 6. Conclusions

Although it is a common belief that null values and other forms of semi-structured information are one of the inevitable features of database applications the object-oriented database research and technology tend to neglect the issue. Practical proposals, such as the ODMG standard, present very few examples of relevant capabilities. Theories proposed for object bases and their query languages, such as object algebras, F-logic, comprehensions and structural recursion, do not supply comprehensive solutions. Moreover, the discussed SQL approach ultimately leads to numerous flaws and inconsistencies.

In the paper we have presented a systematic approach to the problem based on the idea of absent (sub-) objects and briefly described our solutions. These can be summarized into the following recommendations:

- We agree with Ch. Date: a special generic value called *null* (*nil*) should be *absolutely* avoided. We argue that it acts like a little devil that is able to spoil the semantic clarity and consistency of all language constructs; this especially concerns query languages.
- Date suggests to use default values. In the paper we present simple methods to incorporate them into classes.
- In all cases when the use of default values is inconvenient or impossible (e.g., unions, repetitions, complex attributes) binding a name of an absent object should result in an empty table, which can be processed by standard language facilities, such as *exists*, *count*, and quantifiers.

Typing semi-structured data in the context of object-oriented query languages is neither a well understood nor solved issue in current typing systems and theories (including polymorphic ones). We have briefly outlined a new simple approach, which is based on considering types as context-free grammars over data. It presents a new kind of universal polymorphism. The idea is implemented and works in a prototype system LOQIS, but needs further research.

## References

- [AQMWW97] S. Abiteboul, D.Quass, L.McHugh, J.Widom, J.L. Weiner. The Lorel Query Language for Semistructured Data. Technical report, Department of Computer Science, Stanford University, 1996; <ftp://db.stanford.edu/pub/papers/lorel96.ps>; J.Digital Libraries, to appear.
- [Abit97] S. Abiteboul. Querying Semi-Structured Data. Proc. of the International Conference on Database Theory (ICDT'97), 1997, pp.1-17.
- [AbVi97] S. Abiteboul, V. Vianu. Queries and Computations on the Web. Proc. of the International Conference on Database Theory (ICDT'97), 1997, pp.262-275.
- [AKG91] S. Abiteboul, P. Kanellakis, G. Grahne. On the Representation and Querying of Sets of Possible Worlds. Theoretical Computer Science 78(1), 1991, pp.159-187
- [ACO85] A. Albano, L. Cardelli, R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. ACM Transactions on Database Systems, Vol.10, No 2, pp.230-260, 1985.
- [AGO95] A. Albano, G. Ghelli, R. Orsini. Fibonacci: A Programming Language for Object Databases. The VLDB Journal, 4(3), 1995, pp.403-444.
- [ANSI94] American National Standards Institute (ANSI) Database Committee (X3H2). Database Language SQL3. J.Melton, Editor. August 1994.
- [AtBu87] M.P. Atkinson, O.P. Buneman. Types and Persistence in Database Programming Languages. ACM Computing Surveys, Vol.19, No.2, pp.105-190, 1987.
- [ABDD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. Proc. 1-st DOOD Conf., Kyoto, pp.40-57, 1989.
- [AtMo95] M Atkinson, R. Morrison. Orthogonally Persistent Object Systems. The VLDB Journal, 4(3), 1995, pp.319-401.
- [BDK92] F. Bancilhon, C. Delobel, P. Kanellakis. Building an Object-Oriented Database System, The Story of O2. Morgan Kaufmann 1992.

- [BGP90] D. Barbara, H. Garcia-Molina, D. Porter. A Probabilistic Relational Data Model. Proc. Conf. on Extending Database Technology, Venice, Italy 1990, pp.60-74.
- [BGP92] D. Barbara, H. Garcia-Molina, D. Porter. The Management of Probabilistic Data. IEEE Transactions on Knowledge and Data Engineering, 4(5), 1992, pp.487-502.
- [Brei90] Y. Breitbart. Multidatabase Interoperability. SIGMOD Record 19(3), 1990. pp.53-60.
- [Buff92] H.W. Buff. The Relational Model contra Entity Relationship? SIGMOD Record 21(3), Sep.1992, pp.33-34.
- [BDS95] P. Buneman, S.B. Davidson, D. Suciu. Programming Constructs for Unstructured Data. Proc. of 5-th Workshop on DBPL, Sep. 1995, Gubbio, Italy.
- [BDW90] P. Buneman, S.B. Davidson, A. Watters. A Semantics for Complex Objects and Approximate Answers. Journal of Computer and System Sciences, Aug.1990, pp.170-218.
- [BLSTW94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, L. Wong. Comprehension Syntax. SIGMOD Record, Vol. 23, No. 1, pp.87-96, March 1994.
- [BDHS96] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. Proc. of ACM SIGMOD Conf., Montreal, Canada, June 1996. SIGMOD Record, Vol. 25, Issue 2, June 1996, pp.505-516.
- [BDFS97] P. Buneman, S.B. Davidson, M. Fernandez, D. Suciu. Adding Structure to Unstructured Data. Proc. of the International Conference on Database Theory (ICDT'97), 1997, pp.336-350
- [CaWe85] L. Cardelli, P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, Vol.17, No.4, 1985, pp.471-522.
- [Card88] L. Cardelli. A semantics of multiple inheritance. Information and Computation, Vol.76, 1988, pp.138-164.
- [Card89] L. Cardelli. Typeful Programming. DIGITAL Systems Research Center, Palo Alto, Report No 45, May 1989
- [CaPi87] R. Cavallo, M. Pittarelli. The Theory of Probabilistic Databases. Proc. 13<sup>th</sup> VLDB Conf., Brighton, 1987, pp.71-81
- [CoYo91] P. Coad, E. Yourdon. Object-Oriented Design. Prentice Hall, 1991.
- [Codd79] E.F. Codd. Extending Database Relations to Capture More Meaning. ACM Transactions on Database Systems, Vol.4, No 4, 1979, pp.397-434
- [CORBA95] Object Management Group. CORBA: The Common Object Request Broker: Architecture and Specification, July 1995, Release 2.0.
- [Date86] C.J. Date. Relational Database: Selected Writings. Addison-Wesley, 1986.
- [Date86a] (In [Date86], Chapter 13) Some Principles of Good Language Design.
- [Date86b] (In [Date86], Chapter 14) A Critique of the SQL Database Language.
- [Date86c] (In [Date86], Chapter 15) Null Values in Database Management.
- [DaDa92] C.J. Date, H. Darwen. Relational Database Writings 1989-1991. Addison-Wesley, 1992.
- [DaDa92a] (In [DaDa92], Chapter 17) Three-Valued Logic and the Real World.
- [DaDa92b] (In [DaDa92], Chapter 18) Oh No Not Nulls Again.
- [DaDa92c] (In [DaDa92], Chapter 21) The Default Values Approach to Missing Information.
- [DaDa95] H. Darwen, C.J. Date. The Third Manifesto. SIGMOD Record, 24(1), 1995, pp.39-49.
- [DrCh89] H.M. Dreizen, S.K. Chang. Imprecise Schema: A Rationale for Relations with Embedded Subrelations ACM TODS 14(4), Dec.1989, pp.447-479
- [DPT88] D. Dubois and H. Prade and C. Testamale. Handling Incomplete or Uncertain Data and Vague Queries in Database Applications. (In) Possibility Theory: An Approach to Computerized Processing of Uncertainty. Plenum Press, New York and London, 1988, Chapter 6, pp.217-257
- [Dyre95] C. Dyreson. Bibliography Relating to Incomplete Information in Information Systems. Proc. Workshop on Uncertainty Management in Information Systems: From Needs to Solutions, Avalon, Santa Catalina, California, Apr. 1993, pp.187-220 (An updated version: <http://liinwww.ira.uka.de/bibliography/Database/incomplete.html>)
- [FiEy95] D.G. Firesmith, E.M. Eykholt. Dictionary of Object Technology - The Definitive Desk Reference. SIGS Books, New York 1995.
- [Fuhr90] N. Fuhr. A Probabilistic Framework for Vague Queries and Imprecise Information in Databases. Proc. of 16<sup>th</sup> VLDB Conf., Brisbane, Australia, 1990.
- [Gall94] L. Gallagher. Influencing Database Language Standards. SIGMOD Record 23(1), 1994, pp.122-127
- [GYPB92] R. George, A. Yazici, F.E. Petry, B.P. Buckles. Uncertainty Modeling in Object-Oriented Geographical Information Systems. Proc. Conf. on Database and Expert Systems Applications (DEXA'92), Valencia, Spain, 1992.
- [Gess90] G.H. Gessert. Four Value Logic for Relational Database Systems. SIGMOD Record 19(1), 1990, pp.29-35.
- [Gess91] G.H. Gessert. Handling Missing Data by Using Stored Truth Values. SIGMOD Record 20(3), 1991, pp.30-42.

- [INV91] T. Imielinski, S. Naqvi, K. Vadaparty. Incomplete Objects - A Data Model for Design and Planning Applications. Proc. SIGMOD Conf., Denver, CO, May 1991, pp.288-297.
- [Kent91] W. Kent. Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language. Proc. 17<sup>th</sup> VLDB Conf., Barcelona, Spain, 1991, pp.147-160.
- [KiLa89] M. Kifer, G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and Scheme. Proc. of SIGMOD Conf., 1989.
- [Kim90] W. Kim. Introduction to Object-Oriented Databases. The MIT Press, Cambridge, Massachusetts, 1990.
- [Lenz91] M. Lenzerini. Type Data Bases with Incomplete Information. Information Sciences 53, 1991, pp.61-87.
- [LeLo91] M. Levene, G. Loizou. Correction to Null Values in Nested Relational Databases by M.A. Roth, H.F.Korth, and A. Silberschatz. Acta Informatica 28, 1991, pp.603-605.
- [Libk94] L. Libkin. Aspects of Partial Information in Databases. PhD. Dissertation. University of Pennsylvania, 1994 (via <http://www.cis.upenn.edu>)
- [LiSu90a] K.C. Liu, R. Sunderraman. Indefinite and Maybe Information in Relational Databases. ACM TODS 15(1), 1990, pp.1-39.
- [LiSu90b] K.C. Liu, R. Sunderraman. On Representing Indefinite and Maybe Information in Relational Databases: a Generalization. Proc. 6th Conf. on Data Engineering, 1990, pp.495-502.
- [Loom95] M.E.S. Loomis. Object Databases: The Essentials. Addison Wesley, 1995
- [Mart93] J. Martin. Principles of Object-Oriented Analysis and Design. Prentice hall, 1993.
- [Matt95] F. Matthes. Higher-Order Persistent Polymorphic Programming in Tycoon. In M.P. Atkinson, editor, Fully Integrated Data Environments. Springer-Verlag , 1995.
- [MBCD89] R. Morrison, F. Brown, R. Connor, A. Dearle. The Napier88 Reference Manual. Universities of St Andrews and Glasgow, Departments of Comp. Science, Persistent Programming Report 77, July 1989.
- [Motr90] A. Motro. Accommodating imprecision in database systems: issues and solutions. SIGMOD Record 19(4), 1990, pp.69-74.
- [ODMG97] The Object Database Standard ODMG, Release 2.0. R.G.G. Cattell, Ed., Morgan Kaufman, 1997.
- [OBB89] A. Ogori, P. Buneman, V. Breazu-Tannen. Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference. Proc. of ACM SIGMOD 89 Conf., pp.46-57, 1989.
- [PiZi92] A. Pirotte and E. Zimanyi. Imperfect Knowledge in Databases. Research Report 92-36, Unite d'Informatique, Universite de Louvain, Belgium, Oct. 1992.
- [QRSUW95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom. Querying Semistructured Heterogeneous Information. Proc. of the International Conference on Deductive and Object-Oriented Databases (DOOD95) Springer Lecture Notes in Computer Science 1013, 1995, pp.319-344 (<http://www-db.stanford.edu/pub>)
- [RKS89] M. A. Roth and H. F. Korth and A. Silberschatz. Null Values in Nested Databases. Acta Informatica 26, 1989, pp.615-642.
- [RKS91] M. A. Roth and H. F. Korth and A. Silberschatz. Addendum to Null Values in Nested Relational Databases. Acta Informatica 26, 1991, pp.607-610.
- [RBPEL91] J. Rumbaugh, M. Blaha, W. Premarlani, F. Eddy, W. Lorenson. Object-Oriented Modeling and Design. Prentice Hall, 1991.
- [Sadr91] F. Sadri. Modeling Uncertainty in Databases. Proc. 7th Conf. on Data Engineering, Kobe, Japan, 1991, pp.122-131.
- [Semi97] Papers presented at the Workshop on Management of Semistructured Data, Tucson, Arisona, May 1997. Available via <http://www.research.att.com/~suciu/workshop-papers.html>
- [SiZd96] A. Silberschatz, S. Zdonik. Database Systems - Breaking Out of the Box. Unpublished Report, Sept. 1996 (via <http://www.medg.lcs.mit.edu/sdcr/groups.html>)
- [Simo95] A.R. Simon. Strategic Database Technology: Management for the Year 2000. Morgan Kaufmann 1995.
- [SRLG+90] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech. The Committee for Advanced DBMS Function. Third-Generation Data Base System Manifesto. ACM SIGMOD Record 19(3), pp.31-44, 1990.
- [Subi78] K. Subieta. Linguistic Approach to the Database Theory: DDL-s for Hierarchical Models. Information Systems, Vol.3, No.3, 1978, pp.203-208
- [Subi83] K. Subieta. High-Level Navigational Facilities for Network and Relational Databases. Proc. of 9<sup>th</sup> VLDB Conf., Florence, Italy, 1983, pp. 380-386
- [Subi85] K.Subieta. Semantics of Query Languages for Network Databases. ACM Transactions on Database Systems, Vol.10, No.3, 1985, pp.347-394.
- [Subi87] K. Subieta. Denotational Semantics of Query Languages. Information Systems, Vol.12, No.1, 1987, pp.69-82

- [SuMi87] K. Subieta, M. Missala. Semantics of Query Languages for the Entity-Relationship Model. Proc. 5<sup>th</sup> Conf. on Entity-Relationship Approach, Dijon, France, 1987; Elsevier Science Publishers (North-Holland), 1987, pp.197-216
- [SuMi88] K. Subieta, M. Missala. Data Manipulation in NETUL. Proc. of 6<sup>th</sup> Conf. on Entity-Relationship Approach. New York, USA 1988; Elsevier Science Publishers (North-Holland), 1988, pp.391-407
- [SMA90] K. Subieta, M. Missala, and K. Anacki. The LOQIS System. Institute of Computer Science Polish Academy of Sciences, Report 695, Warszawa, Nov. 1990.
- [Subi91] K. Subieta. LOQIS: The Object-Oriented Database Programming System. Proc. 1<sup>st</sup> East/West Database Workshop on Next Generation Information System Technology, Kiev, USSR, 1990, Springer LNCS, Vol.504, 1991, pp.403-421.
- [SMSRW93] K. Subieta, F. Matthes, J.W. Schmidt, A. Rudloff, I. Wetzal. Viewers: A Data-World Analogue of Procedure Calls. Proc. 19<sup>th</sup> VLDB Conf., Dublin, Ireland, 1993, pp.269-277.
- [SBMS93] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt. A Stack- Based Approach to Query Languages. Institute of Computer Science Polish Academy of Sciences, Report 738, Warszawa, Dec. 1993, <http://www.ipipan.waw.pl/~subieta>
- [Subi94] K. Subieta. A Persistent Object Store for the LOQIS Programming System. International Journal on Microcomputer Applications, 13(2), 1994, pp. 50-61
- [SBMS95] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt. A Stack- Based Approach to Query Languages. Proc. 2nd East-West Database Workshop, Klagenfurt, Austria, September 1994, Springer Workshops in Computing, 1995.
- [SKL95a] K. Subieta, Y. Kambayashi, J. Leszczy<sup>3</sup>owski. Procedures in Object-Oriented Query Languages. Proc. 21<sup>st</sup> VLDB Conf., Zurich, 1995, pp.182-193.
- [TYI89] K. Tanaka, M. Yoshikawa, K. Ishihara. Schema Design, Views and Incomplete Information in Object-Oriented Databases. Journal of Information Processing, 12(3), pp.239-250, 1989.
- [Tsim95] S.S. Chawathe, H. Garcia-Molina, A. Gupta, J. Hammer, K. Ireland, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom. The TSIMMIS Project, 1994-95; <http://www-db.stanford.edu/pub>.
- [VrLi91] S.V. Vrbsky and J.W.S. Liu. An Object-Oriented Query Processor that Produces Monotonically Improving Approximate Answers. Proc. Conf. on Data Engineering, Kobe, Japan, 1991.
- [VrLi93] S.V. Vrbsky and J.W.S. Liu. APPROXIMATE: A Query Processor that Produces Monotonically Improving Approximate Answers. IEEE Trans. on Knowledge and Data Engineering 5(6), 1993, pp. 1056-1068.
- [WaGo84] W.M. Waite, G. Goos. Compiler Construction. Springer 1984.
- [WoLe90] M.H. Wong and K.S. Leung. A Fuzzy Database-Query Language. Information Systems 15(5) 1990, pp.583-590.
- [Yazi90] A. Yazici. Representing Imprecise Information in NF2 Relations. IEEE Proc. of South East Conf. '90 - Technologies Today and Tomorrow, New Orleans, LA, 1990, pp.1026-1030.
- [Zade89] L. Zadeh. Knowledge Representation in Fuzzy Logic. IEEE Trans. on Knowledge and Data Engineering 1(1), 1989, pp.89-100.
- [Zani84] C. Zaniolo. Database Relations with Null Values. Journal of Computer and System Sciences 28, 1984, pp.142-166.
- [ZdMa90] S.B. Zdonik, D. Maier. Fundamentals of Object-Oriented Databases. Readings in Object-Oriented Database Systems (S.B. Zdonik, D. Maier, Eds.). Morgan Kaufman, pp.1-32, 1990.
- [Zica90] R. Zicari. Incomplete Information in Object-Oriented Databases. SIGMOD Record 19(3), 1990, pp.5-16.