VQL: A Query Language for Multiversion Databases

Talel Abdessalem and *Geneviève Jomier*
LAMSADE
Paris Dauphine University
75775 Paris cedex 16
E-mail : {abdessalem, jomier}@lamsade.dauphine.fr

November 6, 1997

Abstract

In this paper language VQL is proposed, devoted to querying data stored in multiversion databases. A multiversion database is assumed to represent different states of the modeled universe. A formal model of such a database is presented in this paper. VQL, which is based on a first order calculus, provides a user with a possibility of navigating through object versions, and through the states of the universe modeled by the multiversion database.

Keywords: *query language, versions, multiversion database, database formal model.*

1 Introduction

Numerous papers on versions, which appear since twenty years, are mainly devoted to their organization in file systems and databases [23, 16, 7]. Only few papers deal with retrieving versions from databases. They may be grouped in two families.

The first one concerns query languages for temporal databases [27, 20, 28, 15, 25]. In such databases timestamps are associated with object versions, tuple versions or attribute versions. Timestamps may be either mono or multidimensional, based on validation time, transaction time or user-defined time. A temporal database represents as many states of the modeled universe as the number of determined time units. Query languages for temporal databases are founded on the semantics of time [26, 24, 20, 28]. Their characteristic feature is exploitation of the total ordering of versions imposed by each temporal dimension. Note, however, that time is only one of multiple semantics which may be associated with versions. For instance, in design applications like CAD or CASE, versions may represent variants or alternative design choices. In such case version semantics is not related to time and version ordering is no more total.

A special case of the first family of papers is [3], where query language IQL(2) is proposed as a unique formal support for querying distributed databases and objects with multiple roles, views and versions. We classify it as belonging to the first family, because of its concept of *context*. A context is identified and may be perceived as the place where an object or a value appears, for instance, a site in a distributed database. Such contexts are to some extend similar to the states of the modeled universe appearing in temporal databases.

In the second family of papers [7, 22], object versions have no particular semantics associated with them. Object versions are implemented as objects. Query languages are extended by some primitives which permit version manipulation. However, from the operational point of view, navigation in such a database containing object versions does not differ much from the navigation in a database without versions. As a consequence of the lack of version semantics, expressiveness of query languages is reduced and a big part of the potential of the information contained in the database cannot be used.

In this paper we propose a data model and a query language *VQL* for multiversion databases. The data model is an extension of the one proposed in [2] by version control which in turn is based on the Database Version approach [6]. Query language *VQL* is conceived as an extension of the OQL standard [4, 5].

The main contributions of this paper are: (1) a formal model of a multiversion database, (2) a query language based on a first order calculus and (3) a navigation technique through object versions and through the states of the modeled universe.

The paper is organized as follows. In Section 2 the main concepts of the Database Version approach are presented. In Section 3 a formal model of a multiversion database is proposed. This model is the basis of the query language *VQL* described in Section 4. In Section 5 *VQL* is compared with other languages proposed in the literature. Section 6 concludes the paper.

2 Database Version Approach

In this section the database version approach is presented from the user point of view. We limit its description to the concepts necessary to understand the remainder of this paper: a database version (DBV) and a logical/physical object version.

Database Version. A conventional monoversion database (*i.e.* a database without versions) represents one state of the modeled universe. According to the database version approach, several states of the modeled universe are represented simultaneously in a *multiversion database*. Each state is called *database version* and denoted DBV (*cf.* Figure 1). A DBV has an identifier and contains one version, called a *logical version*, of each object of the modeled universe (we assume that objects are multiversion). A logical object version is similar to an object in a monoversion database: it has an identity and a value. The identifier of the logical version of an object o in a database version v is a couple (o, v). To represent the fact that an object does not exist in a DBV, its logical version contained in this DBV gets a special value $\perp$.

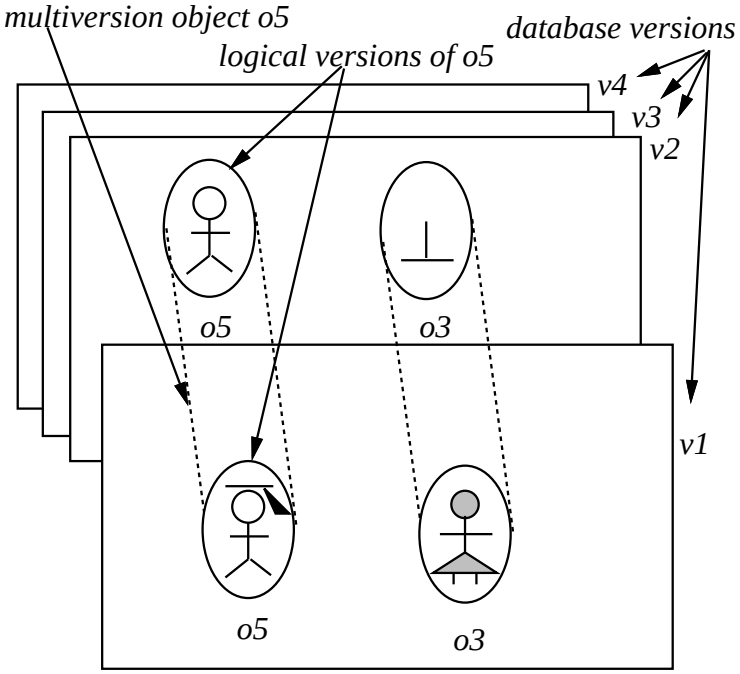

Figure 1: A multiversion Database.

Logical/physical version. When several logical object versions have the same value, it is stored only once, in a *physical version*. The version manager controls the association between logical and physical object versions. More details on the database version approach may be founded in [6, 13].

3 Data model

In this section the data model of a multiversion database is presented, which is an extension of the one proposed in [2] for the multiversion case.

3.1 Basic elements

We assume the following countably infinite and pairwise disjoint sets of atomic elements:

1. relation names $\{R_1, R_2, R_3, ...\}$,

2. class names $\{C_1, C_2, C_3, ...\}$,

3. attributes $\{A_1, A_2, A_3, ...\}$,

4. constants $D = \{d_1, d_2, d_3, ...\}$,

5. object identifiers $O = \{o_1, o_2, o_3, ...\}$,

6. DBV identifiers $V = \{v_1, v_2, v_3, ...\}$.

The product $O \times V$ denotes the set of logical object version identifiers (*cf.* Section 2). In the DBV approach, values are associated with logical object versions.

Definition 3.1 The set of o-values *O-Value* is the smallest set containing $D \cup O$ and such that if ov_1, ..., ov_k ($k \geq 0$) are o-values then $[A_1 : ov_1, ..., A_k: ov_k]$ and $\{ov_1, ..., ov_k\}$ are o-values.

Throughout this exposition, the generic notation $[A_1 : ..., ..., A_k: ...]$, where $k \geq 0$, is used for a tuple formed using any k distinct attributes $A_1, \ldots, A_k$; sets are represented using { } symbols and the empty set is denoted $\emptyset$.

Definition 3.2 The set of v-values *V-Value* is the smallest set containing V and such that if v_1, ..., v_k ($k \geq 0$) are v-values then $[A_1 : v_1, ..., A_k: v_k]$ and $\{v_1, ..., v_k\}$ are v-values.

Definition 3.3 Let **R** be a finite set of relation names and **C** be a finite set of class names.

1. **R** is composed of two disjoint subsets: $\mathbf{R} = \mathbf{Ro} \cup \mathbf{Rv}$, **Ro** for o-value relations, called *o-relations*, and **Rv** for v-value relations, called *v-relations*;

2. A *v-value assignment* for **Rv** is a function ρ_v: $\mathbf{Rv} \longrightarrow$ *V-Value* mapping each name in **Rv** to a v-value;

3. Let $\mathbf{V} = \{\rho_v(R_v) \mid R_v \in \mathbf{Rv}\}$. An *o-value assignment* for **Ro** is a patial function ρ_o mapping each couple (relation name in **Ro**, DBV identifier in **V**) to an o-value. ρ_o: $\mathbf{Ro} \times \mathbf{V} \longrightarrow$ *O-Value*;

4. An *oid assignment* for **C** is a function π: $\mathbf{C} \longrightarrow 2^O_{fin}$ mapping each name in **C** to a finite set of oids. π is called disjoint if $C \neq C'$ implies $\pi(C) \cap \pi(C') = \emptyset$, where $C, C' \in \mathbf{C}$.

In an object-oriented database, **C** is the set of all class names, **R** is the set of all persistent root names and, *O-Value* is the set of all the oids and possible object values. In the multiversion case, a new kind of relations is introduced: the v-relations, which are composed of DBVs. The traditional relations, composed of o-values, are called o-relations.

The assignment functions ρ_v and ρ_o allow getting relation values. An o-relation value may change from one DBV to another. The ρ_o function allows getting the value of an o-relation in each DBV of the multiversion database. When an o-relation R_o is not defined (does not exist) in a DBV v, its value $\rho_o(R_o, v)$ is $\{\}$ if R_o is of set type, otherwise $\rho_o(R_o, v)$ is undefined. The π function returns the oids of objects associated with a given class.

Remark 3.1 From the language point of view, persistent roots are used as entry points to get information from the database. Implementation aspects are beyond the scope of this paper.

Remark 3.2 The oid assignment function π defined above is global to all DBVs: an object belongs to the same class in every DBV. This hypothesis simplifies type checking. For this reason, we assume that the type of objects and o-relations does not change from one DBV to another.

3.2 Syntax and semantics of types

A new type *dbv* is used in the multiversion database case. The type of a v-relation is *dbv*, or *set of dbv-s*. Types associated with objects and values are similar to the ones used in monoversion database case. The set of type expressions $Types(\mathbf{C})$ is defined as: $Types(\mathbf{C}) = types(\mathbf{C}) \cup \{dbv, \{dbv\}\}$, $types(\mathbf{C})$ being the set of type expressions associated with o-values.

Types associated with o-values. Let $\mathbf{C}$ be a set of class names and π be an oid assignment for C. The set of type expressions $types(\mathbf{C})$ is defined as follows:

$\tau = \emptyset \mid D \mid C \mid [A_1 : \tau, \ldots, A_k : \tau] \mid \{\tau\} \mid (\tau \vee \tau) \mid (\tau \wedge \tau)$;
where τ is a type expression, $C \in \mathbf{C}$ and k ≥ 0.

Each type expression τ is given a set of o-values as its *interpretation* $\left[\!\left[\tau\right]\!\right]_\pi$, in the following natural manner:

- $\left[\!\left[\emptyset\right]\!\right]_\pi = \emptyset$, $\left[\!\left[D\right]\!\right]_\pi = D$, $\left[\!\left[C\right]\!\right]_\pi = \pi(C)$, for each $C \in \mathbf{C}$;
- $\left[\!\left[(\tau_1 \vee \tau_2)\right]\!\right]_\pi = \left[\!\left[\tau_1\right]\!\right]_\pi \cup \left[\!\left[\tau_2\right]\!\right]_\pi$ and $\left[\!\left[(\tau_1 \wedge \tau_2)\right]\!\right]_\pi = \left[\!\left[\tau_1\right]\!\right]_\pi \cap \left[\!\left[\tau_2\right]\!\right]_\pi$;
- $\left[\!\left[\{\tau\}\right]\!\right]_\pi = \left\{\{ov_1, \ldots, ov_j\} \mid \ j \geq 0, \text{ and } ov_i \in \left[\!\left[\tau\right]\!\right]_\pi, i = 1, \ldots, j\right\}$;
- $\left[\!\left[[A_1 : \tau_1, \ldots, A_k : \tau_k]\right]\!\right]_\pi = \left\{[A_1 : ov_1, \ldots, A_k : ov_k] \mid ov_i \in \left[\!\left[\tau_i\right]\!\right]_\pi, i = 1, \ldots, k\right\}$.

Types associated with DBVs. Let τ be a type expression belonging to $\{dbv, \{dbv\}\}$. The interpretation of τ, $\left[\!\left[\tau\right]\!\right]$, is defined as follows:

- $\left[\!\left[dbv\right]\!\right] = V$;
- $\left[\!\left[\{dbv\}\right]\!\right] = 2^V_{fin}$.

Remark 3.3 (1) The set of type expressions $Types(\mathbf{C})$ is depending on the set of class names $\mathbf{C}$. This is due to the fact that class names are used as type expressions, to designate abstract types. (2) In this paper we limit type construction over dbv to sets. This constraint will be removed in future work.

3.3 Database schema and instances

Definition 3.4 A multiversion database schema is a quadruple $S_{mv} = (\mathbf{Ro}, \mathbf{Rv}, \mathbf{C}, \mathbf{T})$, where $\mathbf{Ro}$ is a finite set of o-relation names; $\mathbf{Rv}$ is a finite set of v-relation names; $\mathbf{C}$ is a finite set of class names and $\mathbf{T}$ is a function from $\mathbf{Ro} \cup \mathbf{Rv} \cup \mathbf{C}$ to the set of type expressions $Types(\mathbf{C})$.

Definition 3.5 An instance I_{mv} of a schema $S_{mv} = (\mathbf{Ro}, \mathbf{Rv}, \mathbf{C}, \mathbf{T})$ is a quadruple $(\rho_o, \rho_v, \pi, \nu')$, where ρ_o is an o-value assignment for $\mathbf{Ro}$; ρ_v is a DBV assignment for $\mathbf{Rv}$; π is a disjoint oid assignment for $\mathbf{C}$ and ν': $\{\pi(C) \mid C \in \mathbf{C}\} \times \mathbf{V} \longrightarrow O\text{-}Value$ is a value assignment function such that:

1. $\rho_o(R_o, v) \subseteq \left[\!\left[\mathbf{T}(R_o)\right]\!\right]_\pi$, for each $R_o \in \mathbf{Ro}$ and $v \in \mathbf{V}$;
2. $\rho_v(R_v) \subseteq \left[\!\left[\mathbf{T}(R_v)\right]\!\right]$ for each $R_v \in \mathbf{Rv}$;
3. $\nu'(\pi(C), v) \subseteq \left[\!\left[\mathbf{T}(C)\right]\!\right]_\pi$, for each $C \in \mathbf{C}$ and $v \in \mathbf{V}$;
4. ν' is total on $\pi(C) \times \mathbf{V}$, for each $C \in \mathbf{C}$ with $\mathbf{T}(C) = \{\tau\}$.

Let $I_{mv} = (\rho_o, \rho_v, \pi, \nu')$ be an instance of schema $S_{mv} = (\mathbf{Ro}, \mathbf{Rv}, \mathbf{C}, \mathbf{T})$. Each oid occurring in I_{mv} (*i.e.* in the ranges of ρ_o, π and ν') belongs to some $\pi(C)$, where $C \in \mathbf{C}$. This follows from conditions (1) and (3) of the Definition 3.5 and from the semantics of types. A *set valued* oid in I_{mv} is an oid belonging to a class C, where $\mathbf{T}(C) = \{\tau\}$ for some $\tau \in types(\mathbf{C})$. In the same manner, a set valued o-relation R_o is an o-relation such that $\mathbf{T}(R_o) = \{\tau\}$, $\tau \in types(\mathbf{C})$. A set valued v-relation R_v is a v-relation such that $\mathbf{T}(R_v) = \{vbd\}$. The information contained in I_{mv} can be summarized as follows:

$$
\begin{aligned}
ground\text{-}facts(I_{mv}) \quad = \quad & \{v \mid v \in \rho_v(R_v), R_v \text{ is a set valued v-relation}\} \cup \\
& \{v \mid v = \rho_v(R_v), R_v \text{ is a non-set valued v-relation}\} \cup \\
& \{ov \mid ov \in \rho_o(R_o, v), R_o \text{ is a set valued o-relation, and } v \in \mathbf{V}\} \cup \\
& \{ov \mid ov = \rho_o(R_o, v), R_o \text{ is a non-set valued o-relation, and } v \in \mathbf{V}\} \cup \\
& \{o \mid o \in \pi(C), C \in \mathbf{C}\} \cup \\
& \{ov \mid ov \in \nu'(o, v), o \text{ is a set valued oid, and } v \in \mathbf{V}\} \cup \\
& \{ov \mid ov = \nu'(o, v), o \text{ is a non-set valued oid, and } v \in \mathbf{V}\}
\end{aligned}
$$

Condition 4 of Definitiony 3.5 specifies that ν' is total for set valued oids. We follow the convention that, given a DBV v, if for a set valued oid o there is no ground fact $ov \in \nu'(o, v)$ then $\nu'(o, v) = \{\}$, and if for a non-set valued oid o there is no ground fact $ov = \nu'(o, v)$ then $\nu'(o, v)$ is undefined.

In section 2 we specified that an object has a logical version and then a value in each DBV. This value being $\perp$ in some DBVs. Formally, $\perp$ is an empty set for set valued oids, and it is an undefined value for non-set valued oids.

Example 3.1 To illustrate the concepts set out in this section, let us consider the "imaginary family" multiversion database shown on figure 2. Three names of relations appear: *My_parents* and *My_friends* designate two o-relations and *My_DBVs* designates a v-relation. For My_parents the o-relation value is the same in every DBV. For My_friend the o-relation value varies according to the DBV.

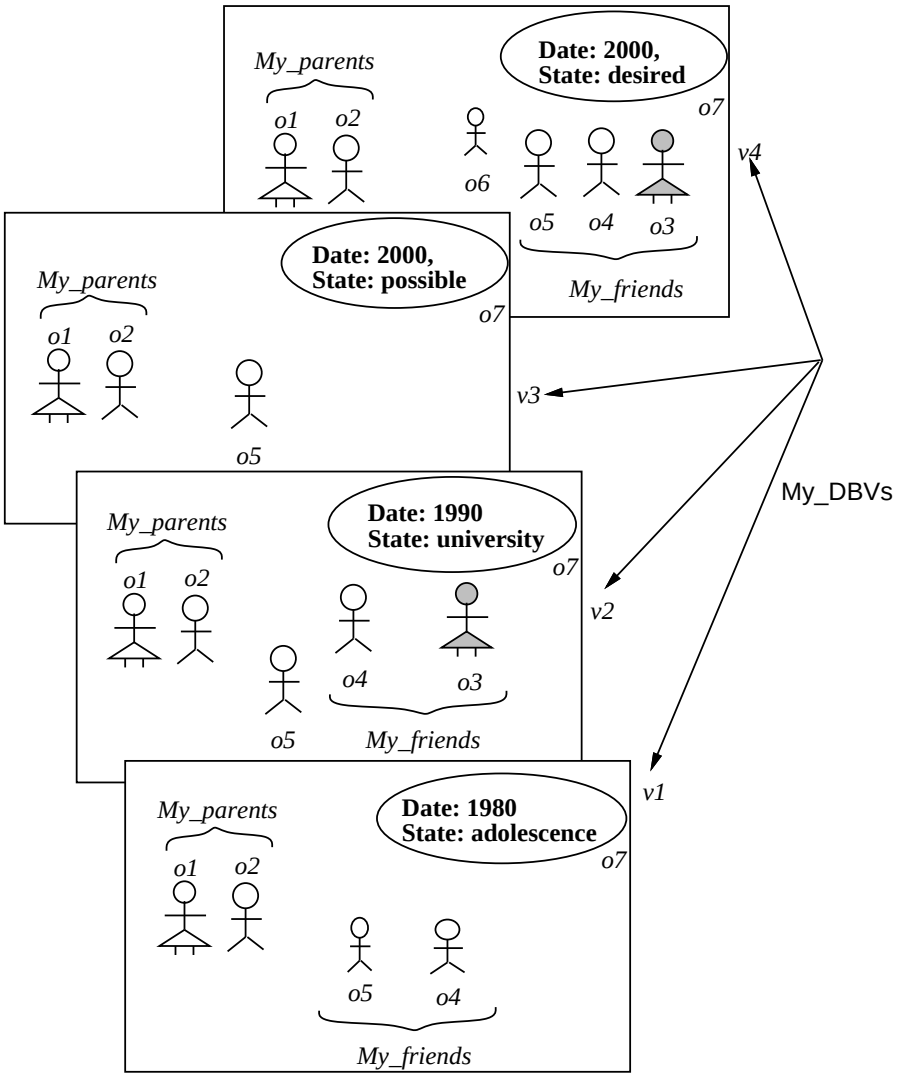

Figure 2: The imaginary family database.

In this example, each DBV represents the state of persons that are "close to me", at a given time. Time is represented as an attribute of the object named DBV_Desc (*i.e.* DBV descriptor). In each DBV, the value of this object describes the corresponding state of the modeled universe.

Database schema.
Ro = {*My_parents*, *My_friends*, *DBV_Desc*},
Rv = {*My_DBVs*},
C = {*Person*, *Descriptor*} and
T is defined by:

$class\ \ Person : [Name : string, City : string, Kids : \{Person\}]$
$class\ \ Descriptor : [Date : integer, State : string]$

$type\ \ My\_parents : \{Person\}$
$type\ \ My\_friends : \{Person\}$
$type\ \ DBV\_Desc : Descriptor$
$type\ \ My\_DBVs : \{dbv\}$

Instance description (see figure 2).

$$\pi(Person) = \{o_1, o_2, o_3, o_4, o_5, o_6\} \quad \pi(Descriptor) = \{o_7\}$$
$$\rho_v(My\_DBVs) = \{v_1, v_2, v_3, v_4\}$$

$DBV\ v_1$

$$\rho_o(My\_parents, v_1) = \{o_1, o_2\}$$
$$\rho_o(My\_friends, v_1) = \{o_4, o_5\}$$
$$\rho_o(DBV\_Desc, v_1) = o_7$$
$$\nu'(o_1, v_1) = [Name : Ben, City : Paris, Kids : \{\}]$$
$$\nu'(o_2, v_1) = [Name : Margareth, City : Paris, Kids : \{\}]$$
$$\nu'(o_3, v_1) \text{ is } undefined$$
$$\nu'(o_4, v_1) = [Name : Charles, City : Paris, Kids : \{\}]$$
$$\nu'(o_5, v_1) = [Name : Moustapha, City : St\ Denis, Kids : \{\}]$$
$$\nu'(o_6, v_1) \text{ is } undefined$$
$$\nu'(o_7, v_1) = [Date : 1980, State : adolescence]$$

$DBV\ v_2$

$$\rho_o(My\_parents, v_2) = \{o_1, o_2\}$$
$$\rho_o(My\_friends, v_2) = \{o_3, o_4\}$$
$$\rho_o(DBV\_Desc, v_2) = o_7$$
$$\nu'(o_1, v_2) = [Name : Ben, City : Marseille, Kids : \{\}]$$
$$\nu'(o_2, v_2) = [Name : Margareth, City : Marseille, Kids : \{\}]$$
$$\nu'(o_3, v_2) = [Name : Imen, City : Nice, Kids : \{\}]$$
$$\nu'(o_4, v_2) = [Name : Charles, City : Paris, Kids : \{\}]$$
$$\nu'(o_5, v_2) = [Name : Moustapha, City : St\ Denis, Kids : \{\}]$$
$$\nu'(o_6, v_2) \text{ is } undefined$$
$$\nu'(o_7, v_2) = [Date : 1990, State : university]$$

$DBV\ v_3$

$$\rho_o(My\_parents, v_3) = \{o_1, o_2\}$$
$$\rho_o(My\_friends, v_3) = \{\}$$
$$\rho_o(DBV\_Desc, v_3) = o_7$$
$$\nu'(o_1, v_3) = [Name : Ben, City : Tunis, Kids : \{\}]$$
$$\nu'(o_2, v_3) = [Name : Margareth, City : Tunis, Kids : \{\}]$$
$$\nu'(o_3, v_3) \text{ is } undefined$$
$$\nu'(o_4, v_3) \text{ is } undefined$$
$$\nu'(o_5, v_3) = [Name : Moustapha, City : St\ Denis, Kids : \{\}]$$
$$\nu'(o_6, v_3) \text{ is } undefined$$
$$\nu'(o_7, v_3) = [Date : 2000, State : possible]$$

$DBV\ v_4$

$\rho_o(My\_parents, v_4) = \{o_1, o_2\}$
$\rho_o(My\_friends, v_4) = \{o_3, o_4, o_5\}$
$\rho_o(DBV\_Desc, v_4) = o_7$
$\nu'(o_1, v_4) = [Name : Ben, City : Tunis, Kids : \{\}]$
$\nu'(o_2, v_4) = [Name : Margareth, City : Tunis, Kids : \{\}]$
$\nu'(o_3, v_4) = [Name : Imen, City : Paris, Kids : \{\}]$
$\nu'(o_4, v_4) = [Name : Charles, City : LosAngeles, Kids : \{o_6\}]$
$\nu'(o_5, v_4) = [Name : Moustapha, City : St\ Denis, Kids : \{\}]$
$\nu'(o_6, v_4) = [Name : David, City : LosAngeles, Kids : \{\}]$
$\nu'(o_7, v_4) = [Date : 2000, State : desired]$

3.4 Summary of the introduced concepts

The model presented in this section extends the one presented in [2] by the concept of a DBV. As a consequence, (1) a new set V has been added to contain DBV identifiers, and (2) the set of relation names has been partitioned in two disjoint subsets, one for o-relation names and the other for v-relation names. The o-value assignment function ρ_o has been modified to take into account the fact that an o-relation value may change from one DBV to another. The DBV assignment function ρ_v has been defined for mapping v-relation names to DBV identifiers.

Two new type expressions have been added: dbv and $\{dbv\}$. The interpretation of dbv is the set of DBV identifiers V, and the interpretation of $\{dbv\}$ is the finite powerset of V.

At the schema level, a finite set of v-relation names has been added to the schema definition used in the monoversion database case, and the type function **T** has been modified to take into account this new kind of relations. At the instance level, a DBV assignment function has been added and the value assignment function, denoted ν in the monoversion database case, has been modified to take into account object value changements in different DBVs.

4 Query language

In this section a query language VQL based on a first order calculus is proposed. This calculus extends the proposals of [1, 3] to DBVs.

From the language point of view, there are two main contributions of VQL: (1) specific terms to denote the modeled universe states (DBVs), and (2) a dereferencing operation taking into account the "DBV dimension". The first one enables user to specify the states of the modeled universe he/she wants to query. The standard predicates $\in$ and $=$ are applied on terms denoting DBVs, and no restriction is placed on the quantification on DBVs. The second one makes it possible to keep the track of an object through DBVs (*i.e.* through different states of the modeled universe). A new predicate *Undef ()* is introduced to enable user to determine logical object versions having $\perp$ as a value.

This section is organized as follows. In section 4.1, a complex value calculus integrating DBVs is proposed followed by some examples of queries explaining the use of VQL language. In section 4.3, the semantics of VQL queries is defined. In section 4.4, a navigation technique for multiversion databases is proposed. In section 4.5, the use of quantifiers over DBVs is analyzed and in section 4.6, the use of $\perp$ values is described.

4.1 Calculus

Terms. There are two categories of terms: terms denoting DBVs and terms denoting o-values.

- Terms denoting DBVs (DBV terms)
  1. R_v, for each $R_v \in \mathbf{Rv}$;
  2. x_τ, where x is a variable of type $\tau \in \{dbv, \{dbv\}\}$.
- Terms denoting o-values (o-value terms)
  1. d, for each d in D;
  2. $R_o(t)$, for each $R_o \in \mathbf{Ro}$ and t a DBV term of type dbv;
  3. x_τ, where x is a variable of type τ, $\tau \in types(\mathbf{C})$;
  4. constructed terms such that:

- tuples $[A_1 : t_1, \ldots, A_n : t_n]$, where $A_1, \ldots, A_n$ are attribute names and $t_1, \ldots, t_n$ $(n > 0)$ are o-value terms;
- sets $\{t_1, \ldots, t_n\}$, where $t_1, \ldots, t_n$ $(n \geq 0)$ are o-value terms;
- projection $t.A$, where t is an o-value term of type tuple and A is an attribute name;

5. *dereferencing* $*(t_1, t_2)$, where t_1 is an o-value term denoting an oid and t_2 a DBV term of type *dbv*.

Each variable has a type, and each value (simple or complex) belongs to the interpretation of a type (*cf.* Definition 3.5). Therefore, each term has a type that can be determined easily. For example, the type of a tuple $[A_1 : t_1, ..., A_n : t_n]$, where $t_1, \ldots, t_n$ are terms of respective types $\tau_1, ..., \tau_n$, is $\tau = [A_1 : \tau_1, ..., A_n : \tau_n]$.

Formulas. Predicates $=$ and $\in$ applied to terms with the proper type restrictions yield *atomic formulas*: $t = t'$, $t \in t'$, with t and t' terms of compatible types.

In addition, we define a new predicate *Undef*(). Applied to dereferencing terms, *Undef*$(*(t_1, t_2))$, where t_1 denotes an oid o and t_2 denotes a DBV v, indicates whether $\nu'(o, v)$ is defined or not. The use of this predicate is extended to o-relation terms : *Undef*$(R_o(t_2))$, where R_o is an o-relation name, indicates whether $\rho_o(R_o, v)$ is defined or not. *Undef*$(*(t_1, t_2))$ and *Undef*$(R_o(t_2))$ are atomic formulas.

Formulas are obtained from atomic formulas by application of the connectives $\wedge$, $\vee$, $\neg$, and the quantifiers $\exists$ and $\forall$. If L_1 and L_2 are two formulas, then $L_1 \wedge L_2, L_1 \vee L_2, \neg L_1$, are formulas. If x_τ is a free variable of $L_1(x_\tau)$, then $\exists x_\tau(L_1(x_\tau))$ and $\forall x_\tau(L_1(x_\tau))$ are formulas.

Queries. A query on a multiversion database can be expressed in two ways:

1. $\{ x \mid \varphi \}$, where φ is a formula and x the free variable of φ;
2. $\{ (x, y) \mid \varphi \}$, where φ is a formula and x, y are the free variables of φ, x is of type $\tau \in types(\mathbf{C})$ and y is of type *dbv*.

The result of the first kind of queries is a selection of DBVs, objects or values. When DBVs are selected, the execution of a query gives as a result a multiversion instance composed of the selected DBVs and all the logical object versions and values they contain. When o-values (oids or values) are selected, the output multiversion instance is composed of the selected o-values and as many DBVs as the queried database contains.

The second kind of queries selects couples (o-value, DBV). In this case, the output multiversion instance is composed of the selected DBVs with the selected logical object versions or values as a content. Let (o_1, v_1) and (o_2, v_2) be the result of a query. The output multiversion instance is therefore composed of two DBVs : v_1 and v_2. The only object existing in v_1 is o_1, and the only one existing in v_2 is o_2.

4.2 Examples of VQL queries

4.2.1 DBV selection

Consider the "imaginary family" database (*cf.* Example 3.1). The following query selects the DBV representing the year 1990.

Q_1 : `{ y | y ∈ My_DBVs ∧ *(DBV_Desc(y), y).Date = 1990 }`

The term `DBV_Desc(`y`)` denotes object o_7 for each y in `My_DBVs`. The dereferencing of this object `*(DBV_Desc(`y`), `y`)` allows the access to the value of o_7 in each of `My_DBVs`.

Object o_7 is of the tuple type, the projection on attribute Date allows to find out the corresponding year for each DBV. Only the DBV corresponding to 1990 is selected. In the following, the selected DBV is denoted *Current_DBV*. The expression of query Q_1 in an OQL-like syntax is:

```
define Current_DBV as element  ( SELECT  y
                                 FROM    y in My_DBVs
                                 WHERE   (DBV_Desc, y).Date = 1990 )
```

Here, the term `DBV_Desc(`y`)` has been simplified in `DBV_Desc`. This syntactic simplification is considered in the following, each time the designated o-relation does not vary from a DBV to another.

4.2.2 Object selection

Query Q_2 retrieves an object representing "my friend Charles" in the "past DBV". In the following, this object will be denoted *My_friend*.

Q_2 : { x | $x \in$ My_friends(Current_DBV) $\wedge$ *(x, Current_DBV).Name = "Charles" }

in an OQL-like syntax:

```
define My_friend as element   ( SELECT   x
                                FROM     x in My_friends(Current_DBV)
                                WHERE    (x,Current_DBV).Name = "Charles" )
```

Remark 4.1 A selection made in a fixed DBV is equivalent to the one in a monoversion database. For instance, if the current DBV is considered as a default DBV, query Q_2 can be written in OQL [4, 18] as follows:

```
SELECT   x
FROM     x in My_friends
WHERE    x.Name = "Charles"
```

4.2.3 Value selection

Query Q_3 retrieves the "possible" and the "desired" residence cities of "my friend Charles" in 2000.

Q_3 : { x | $\exists y$ ($y \in$ My_DBVs $\wedge$ *(DBV_Desc(y), y).Date=2000 $\wedge$
x = *(My_friend(y), y).City) }

in an OQL-like syntax:

```
SELECT   (My_friend, y).City
FROM     y in My_DBVs
WHERE    (DBV_Desc, y).Date = 2000
```

Remark 4.2 This query can be expressed in a simpler way if we look for the residence city in the "current DBV": `(My_friend, Current_DBV).City`, or `My_friend.City`, if the past DBV is the one taken by default by the system.

4.2.4 (o-value, DBV) couple selection

The purpose of the (o-value, DBV) couple selection is to allow a user to know which DBV contains which logical object version or value selected.

The following query looks for "my friends" in different DBVs. Each retrieved object is associated with the DBV in which it represents a "friend of mine".

Q_4 : { (x, y) | $y \in$ My_DBVs $\wedge$ $x \in$ My_friends(y) $\wedge$
(*(DBV_Desc(y), y).Date=1980 $\vee$
*(DBV_Desc(y), y).Date=1990) }

in an OQL-like syntax:

```
SELECT   (x, y)
FROM     y in My_DBVs, x in My_friends(y)
WHERE    (DBV_Desc, y).Date = 1980  or
         (DBV_Desc, y).Date = 1990
```

In this example, selected couples identify logical object versions. The selection is restricted to the DBVs of 1980 and 1990 (v_1 and v_2). Also, the content of these DBVs is restricted to the logical versions of objects representing "my friends" (o_4 and o_5 in v_1 and o_3, o_4 in v_2, as presented in Figure 3).

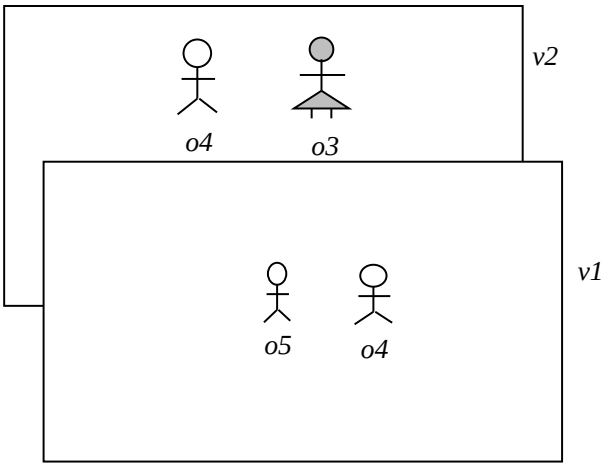

Figure 3: Query Q_4 output instance.

4.3 Semantics

The notions presented in this section are staightforward extensions of those used for the semantics of IQL in [2]. They are slightly complicated by the use of DBVs in VQL.

For the comprehension of VQL query semantics, we need to define projections of multiversion database schemas an instances. Let S_{mv}=(**Ro**, **Rv**, **C**, **T**) be a multiversion database schema.

> A schema S'_{mv}= (**Ro**′, **Rv**′, **C**′, **T**′) is a *projection* of schema S_{mv} if **Ro**′ ⊆ **Ro**, **Rv**′ ⊆ **Rv**, **C**′ ⊆ **C** and **T**′ is the mapping of **T** on **Ro**′ ∪ **Rv**′ ∪ **C**′.
>
> Given a multiversion instance $I_{mv} = (\rho_o, \rho_v, \pi, \nu')$ of S_{mv}, its projection on S'_{mv}, denoted $I_{mv}[S'_{mv}]$, is defined as a mapping of ρ_o, ρ_v, π and ν' on **Ro**', **Rv**' and **C**'. $I_{mv}[S'_{mv}]$ is an instance of S'_{mv}.

A VQL query, denoted $\Gamma_{mv}(S_{mv}, S_{mv-in}, S_{mv-out})$, is composed of formulas over a schema S_{mv}. Its semantics is a binary relation between multiversion instances. This relation, denoted γ_{mv}, associates a multiversion instance over the *output schema* S_{mv-out} to each instance over the *input schema* S_{mv-in}, where S_{mv-in} and S_{mv-out} are two projections of S_{mv}. Intuitively, S_{mv-in} contains the names of the queried relations, the class names associated with the queried objects and the corresponding type association function, and S_{mv-out} contains the names of the retrieved relations, the class names of the retrieved objects and the corresponding type function. The input of a query is a multiversion instance I_{mv} over S_{mv-in}, the computation of the query defines a multiversion instance J_{mv} over S_{mv}, and the output is $J_{mv}[S_{mv-out}]$.

Valuation. Given a multiversion instance $I_{mv} = (\rho_o, \rho_v, \pi, \nu')$, the valuation of VQL queries is done using two disjoint functions: θ_v for DBV valuation, and θ_o for o-value valuation.

A *DBV valuation function* θ_v is a partial function from variables y of type $\tau \in \{dbv, \{dbv\}\}$ to $V \cup 2^V_{fin}$ such that: if $\theta_v\left(y\right)$ is defined, then $\theta_v\left(y\right) \in \left[\!\left[\tau\right]\!\right]$. A DBV valuation can be extended to DBV terms as follows:

$\theta_v\left(R_v\right) = v$ such that $v = \rho_v(R_v)$.

An *o-value valuation function* is a partial function from variables x of type $\tau \in types(\mathbf{C})$ to o-values such that: if $\theta_o\left(x\right)$ is defined, then $\theta_o\left(x\right) \in \left[\!\left[\tau\right]\!\right]_\pi$). An o-value valuation can be extended to o-value terms as follows:

$\theta_o\left(d\right) = d$;

$\theta_o\left(R_o(t)\right) = ov$ such that $ov = \rho_o(R_o, \theta_v\left(t\right))$;

$\theta_o\left(\{t_1, \ldots, t_k\}\right) = \{\theta_o\left(t_1\right), \ldots, \theta_o\left(t_k\right)\}$,
$\theta_o\left([A_1 : t_1, \ldots, A_k : t_k]\right) = [A_1 : \theta_o\left(t_1\right), \ldots, A_k : \theta_o\left(t_k\right)]$,
where $k \geq 0$;

$\theta_o\Big(t.A\Big) = ov$ such that $ov = \theta_o\Big(t\Big).A$;

$\theta_o\Big(*(t_1, t_2)\Big) = ov$ such that $ov = \nu'(\theta_o\Big(t_1\Big), \theta_v\Big(t_2\Big))$.

Satisfaction. Let I_{mv} be a multiversion instance, θ_v a DBV valuation that must be defined on DBV terms t_1, t_2, and θ_o an o-value valuation that must be defined on o-value terms t'_1, t'_2. The following rules allow to determine whether I_{mv} *satisfies* ($\models$) a VQL query or not.

$I_{mv} \models t_1 \in t_2$ if $\theta_v\Big(t_1\Big) \in \theta_v\Big(t_2\Big)$; $\quad$ $I_{mv} \models t'_1 \in t'_2$ if $\theta_o\Big(t'_1\Big) \in \theta_o\Big(t'_2\Big)$;

$I_{mv} \models t_1 = t_2$ if $\theta_v\Big(t_1\Big) = \theta_v\Big(t_2\Big)$; $\quad$ $I_{mv} \models t'_1 = t'_2$ if $\theta_o\Big(t'_1\Big) = \theta_o\Big(t'_2\Big)$;

$I_{mv} \models \neg(t_1 \in t_2)$ if $\theta_v\Big(t_1\Big) \notin \theta_v\Big(t_2\Big)$; $\quad$ $I_{mv} \models \neg(t'_1 \in t'_2)$ if $\theta_o\Big(t'_1\Big) \notin \theta_o\Big(t'_2\Big)$;

$I_{mv} \models \neg(t_2 = t_1)$ if $\theta_v\Big(t_1\Big) \neq \theta_v\Big(t_2\Big)$; $\quad$ $I_{mv} \models \neg(t'_2 = t'_1)$ if $\theta_o\Big(t'_1\Big) \neq \theta_o\Big(t'_2\Big)$;

$I_{mv} \models \mathit{Undef}(*(t'_1, t_1))$ if $\nu'(\theta_o\Big(t'_1\Big), \theta_v\Big(t_1\Big))$ is undefined;

$I_{mv} \models \neg\ \mathit{Undef}(t'_1, t_1)$ if $\nu'(\theta_o\Big(t'_1\Big), \theta_v\Big(t_1\Big))$ is defined;

$I_{mv} \models \mathit{Undef}(R_o(t_1))$ if $\rho(R_o, \theta_v\Big(t_1\Big))$ is undefined;

$I_{mv} \models \neg\ \mathit{Undef}(R_o(t_1))$ if $\rho(R_o, \theta_v\Big(t_1\Big))$ is defined.

In addition, let L_1, L_2 be two formulas. We say that:

$I_{mv} \models L_1 \wedge L_1$ if $I_{mv} \models L_1$ and $I_{mv} \models L_2$;
$I_{mv} \models L_1 \vee L_2$ if $I_{mv} \models L_1$ or $I_{mv} \models L_2$;

$I_{mv} \models \exists x_\tau(L_1(x_\tau)), \tau \in \{dbv, \{dbv\}\}$, if it exists $v = \theta_v\Big(x\Big)$ such that $I_{mv} \models L_1(v)$;

$I_{mv} \models \exists x_\tau(L_1(x_\tau)), \tau \in types(\mathbf{C})$, if it exists $ov = \theta_o\Big(x\Big)$ such that $I_{mv} \models L_1(ov)$;

$I_{mv} \models \forall x_\tau(L_1(x_\tau)), \tau \in \{dbv, \{dbv\}\}$, if for each $v = \theta_v\Big(x\Big)$, $I_{mv} \models L_1(v)$;

$I_{mv} \models \forall x_\tau(L_1(x_\tau)), \tau \in types(\mathbf{C})$, if for each $ov = \theta_o\Big(x\Big)$, $I_{mv} \models L_1(ov)$;

$I_{mv} \models \neg(L_1 \wedge L_1)$ if $I_{mv} \models \neg L_1$ or $I_{mv} \models \neg L_2$;
$I_{mv} \models \neg(L_1 \vee L_2)$ if $I_{mv} \models \neg L_1$ and $I_{mv} \models \neg L_2$;
$I_{mv} \models \neg(\exists x_\tau(L_1(x_\tau)))$ if $I_{mv} \models \forall x_\tau(\neg L_1(x_\tau))$;
$I_{mv} \models \neg(\forall x_\tau(L_1(x_\tau)))$ if $I_{mv} \models \exists x_\tau(\neg L_1(x_\tau))$.

4.3.1 Example of valuation

Lets consider query Q_2, presented in section 4.2. FI_{in} denotes the input instance of this query. The computation of Q_2 is done following the next valuation steps :

$FI_{in} \models Q_2$ if $FI_{in} \models x \in$ My_friends(Current_DBV) and
$FI_{in} \models$ *(x, Current_DBV).Name = "Charles"

i.e., if $FI_{in} \models \theta_o\Big(x\Big) \in \theta_o\Big($ My_friends(Current_DBV) $\Big)$ and
$FI_{in} \models \theta_o\Big($ *(x, Current_DBV).Name $\Big) = \theta_o\Big($ "Charles" $\Big)$

i.e., if $FI_{in} \models \theta_o\Big(x\Big) \in \rho_o($ My_friends, $\theta_v\Big($Current_DBV$\Big)$) and
$FI_{in} \models \theta_o\Big($*($x$, Current_DBV)$\Big)$.Name = "Charles"

i.e., if $FI_{in} \models \theta_o\Big(x\Big) \in \rho_o($ My_friends, ρ_v(Current_DBV)) and
$FI_{in} \models \nu'(\theta_o\Big(x\Big), \theta_v\Big($ Current_DBV $\Big)$).Name = "Charles"

i.e. if $FI_{in} \models \theta_o\Big(x\Big) \in \rho_o($ My_friends, v_2) and
$FI_{in} \models \nu'(\theta_o\Big(x\Big),$ ρ_v(Current_DBV)).Name = "Charles"

i.e., if (1) $FI_{in} \models \theta_o\Big(x\Big) \in \{o_3, o_4\}$ and
(2) $FI_{in} \models \nu'(\theta_o\Big(x\Big),$ v_2).Name = "Charles"

Query Q_2 looks for o-values $ov = \theta_o\Big(x\Big)$ such that conditions (1) and (2) are satisfied. Instance FI_{in} satisfies $\nu'(o_4,$ v_2).Name = "Charles", but it does not satisfy $\nu'(o_3,$ v_2).Name≠"Charles". So, o_4 is the only object retrieved by Q_2, *i.e.* o_4 is the only oid occurring in Q_2 output instance.

Treatment of $\perp$ values. Lets imagine that object o_4 does not exist in DBV v_2, *i.e.* it has $\perp$ as a value in DBV v_2. Formally this means that $\nu'(o_4, v_2)$ is undefined and, consequently, FI_{in} doesn't satisfy $\nu'(o_4,$ v_2).Name = "Charles".

4.4 Navigating through object versions

The queries described in Section 4.1 take DBVs one at a time. Thus, they are similar to a queries devoted for monoversion databases. In some cases, the selection predicate has to be computed on several DBVs. For example, query Q_5 retrieves - from among "my friends" in 1990 - those that did change their residence between 1980 and 1990. Note that "my friends" in 1990 are different from those in 1980. Query Q_5 checks for each of "my friends" in 1990 if he/she had the same residence in 1980.

$$Q_5: \quad \{\ (x,y)\ \mid\ y \in \texttt{My\_DBVs} \ \wedge\ \texttt{*(DBV\_Desc(}y\texttt{), }y\texttt{).Date = 1990} \ \wedge\ x \in \texttt{My\_friends(}y\texttt{)} \ \wedge\ \exists z\ (z \in \texttt{My\_DBVs} \ \wedge\ \texttt{*(DBV\_Desc(}z\texttt{), }z\texttt{).Date = 1980} \ \wedge\ \neg(\texttt{*(}x\texttt{, }z\texttt{).City = *(}x\texttt{, }y\texttt{).City}) \ )\ \}$$

in an OQL-like syntax:

```
SELECT   (x,y)
FROM     y in My_DBVs, x in My_friends(y),
         z in My_DBVs
WHERE    (DBV_Desc, y).Date = 1990 and (DBV_Desc, z).Date = 1980 and
         (x,y).City != (x,z).City
```

In this example, the selected objects are followed in two DBVs corresponding to 1990 and 1980. Each queried object is first selected in the DBV of 1990, then its values in both the VBDs of 1990 and 1980 are reached by the use of the dereferencing operation. The comparison of these values makes it possible to deduce the "evolution" of the object through these two DBVs. We assume that no changes occurred in between 1980 and 1990, otherwise they would be represented in the database as separate DBVs.

Paths through DBVs

In a declarative language, a query states the required information and the path to reach it. In a monoversion database, a path goes through a finite sequence of objects/values [9, 8, 17]. In a multiversion database, a path may either be limited to one DBV only - as in the case of a monoversion database - or it may go across DBVs (*cf.* Figure 4).

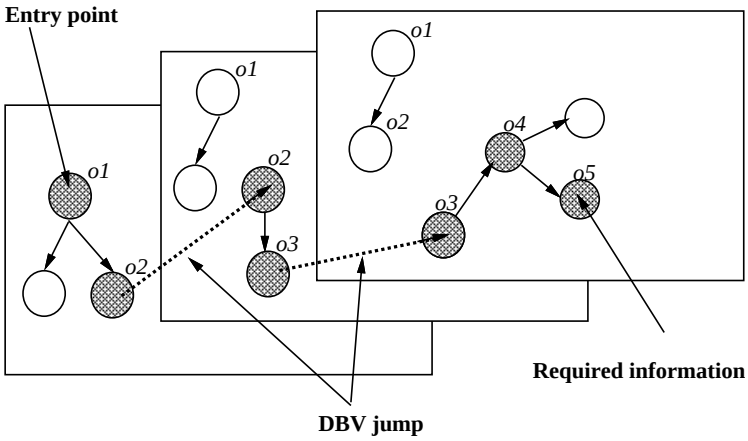

Figure 4: A path going across several DBVs.

The dereferencing operation can be used to "jump" from one logical version of an object to another logical version of the same object, *i.e.* from one DBV to another. A path beginning in a DBV v_1 can continue in a DBV v_2, after an object dereferencing in v_1. So, a user may navigate in a multiversion database through a finite sequence of logical object versions/values, possibly contained in different DBVs.

The following example concerns paths across DBVs. We consider the "imaginary" family database with the following schema and instance additions.

$$\begin{array}{ll} class & Person : [Name : string, City : string, Kids : \{Person\}, Diploma : Certificate] \\ class & Certificate\text{: [} Title\text{: } string, Institute : string] \\ type & My\_best\_friend : Person \end{array}$$

$$\begin{array}{l} \rho_o(My\_best\_friend, v_1) = o_5 \\ \rho_o(My\_best\_friend, v_2) = o_4 \\ \rho_o(My\_best\_friend, v_3) \ is\ undefined \\ \rho_o(My\_best\_friend, v_4) = o_3 \end{array}$$

$$\begin{array}{l} \nu'(o_5, v_2) = [Name : Moustapha, City\text{: } St\ Denis, Kids : \{\}, Diploma : o_9] \\ \nu'(o_9, v_2) = [\ Title\text{: } MD\ 93 \text{ - } Computer\ Science, Institute\text{: } Univ.\ of\ St\ Denis\ ] \\ \nu'(o_9, v_3) \ is\ undefined \\ \nu'(o_9, v_4) = [\ Title\text{: } MD\ 93 \text{ - } DB\ and\ AI, Institute\text{: } Univ.\ of\ St\ Denis\ ] \end{array}$$

The title in 2000 of the diploma obtained in 1990 by the best friend y had in 1980 is required. Query Q_6 retrieves it. In this query, paths expressed by the term `*(*(My_best_friend(`t`), `z`).Diploma, `y`).Title`, go through three DBVs. Each path begins at the o-relation `My_best_friend` in the 1980 DBV, continues through the logical version of object o_5 in the 1990 DBV, then through the logical version of object o_9 in the 1990 DBV, and ends at a logical version of object o_9 in one the DBVs representing year 2000. This example shows that it is possible to select information contained in one DBV, starting from an entry point contained in another.

$$\begin{array}{lll} Q_6: & \{\ (x,\ y)\ \mid & y \in \texttt{My\_DBVs} \quad \wedge \quad \texttt{*(DBV\_Desc(}y\texttt{),}\ y\texttt{).Date = 2000} \quad \wedge \\ & & \exists z\ \exists t\ (\ z \in \texttt{My\_DBVs} \quad \wedge \quad \texttt{*(DBV\_Desc(}z\texttt{),}\ z\texttt{).Date = 1990} \quad \wedge \\ & & \qquad t \in \texttt{My\_DBVs} \quad \wedge \quad \texttt{*(DBV\_Desc(}t\texttt{),}\ t\texttt{).Date = 1980} \quad \wedge \\ & & \qquad\qquad x = \texttt{*(*(My\_best\_friend(}t\texttt{),}\ z\texttt{).Diploma,}\ y\texttt{).Title}\ \ )\ \} \end{array}$$

in an OQL-like syntax:

```
SELECT   (x, y)
FROM     y in My_DBVs, z in My_DBVs, t in My_DBVs
WHERE    (DBV_Desc,y).Date=2000 and
         (DBV_Desc,z).Date=1990 and
         (DBV_Desc,t).Date=1980 and
         x =*(*(My_best_friend(t), z).Diploma, y).Title
```

Paths expressed here go through two DBVs. Each path begins at a logical object version representing one of "my friends" in the 1990 DBV, and ends at a logical object version representing a diploma in the 2000 DBV. This example shows that it is possible to select information contained in one DBV, starting from an entry point contained in another.

The valuation of variable t, z and y gives :

$$\theta_v\Big(t\Big) \;=\; v_1, \quad \theta_v\Big(z\Big) = v_2 \quad \text{et} \quad \theta_v\Big(y\Big) \in \{v_3, v_4\}.$$

Two path expressions are valuated: (1) `*(*(My_best_friend(`v_1`), `v_2`).Diploma, `v_3`).Title` and (2) `*(*(My_best_friend(`v_1`), `v_2`).Diploma, `v_4`).Title`.

Since, $\theta_o\Big($`*(*(My_best_friend(`v_1`), `v_2`).Diploma, `v_3`)`$\Big)$ is $\perp$, only the second path valuation returns a value for x : $\theta_o\Big(x\Big)$="DB et AI".

4.5 Quantifying on DBV

In VQL, the *quantification on DBVs* is done in the same way as on objects or values. Quantification on DBVs is usually associated with quantification on objects and/or values. Query Q_7 is an example of association in the form $\{\ldots \mid \forall x\ \exists v\ \ldots\}$, where x designates an o-value and v a DBV. In this association, the value of v depends on that of x.

Query Q_7 looks for "my friends" in 1990, whose all children own a PhD. Note that in each DBV only the diplomas obtained in the year corresponding to this DBV are represented.

$$\begin{array}{lllll} Q_7: \quad \{\ (x,y) \mid & y \in \texttt{My\_DBVs} & \wedge & \texttt{*(DBV\_Desc(}y\texttt{),}y\texttt{).Date=1990} & \wedge \\ & & & x \in \texttt{My\_friends(}y\texttt{)} & \wedge \\ & \forall t(t \in \texttt{*(}x\texttt{, }y\texttt{).kids} & \wedge & \exists z(z \in \texttt{My\_DBVs} & \wedge \\ & & & \quad \texttt{*(*(}t\texttt{,}z\texttt{).Diploma,}z\texttt{).Title = "PhD"))} & \quad \} \end{array}$$

in an OQL-like syntax:

```
SELECT   (x, y)
FROM     y in My_DBVs, x in My_friends(y)
WHERE    (DBV_Desc, y).Date = 1990 and
         for all t in (x, y).kids :
                exists z in My_DBVs :  ((t,z).Diploma, z).Title="PhD"
```

The PhD defense year is specific to each child. So, the DBV containing this information (PhD defense) depends on the child.

4.6 Using $\perp$ value

As mentioned in section 2, a logical version of an object o contained in a DBV v gets a special value $\perp$ to indicate that object o does not exist in DBV v. Formally, this value is $\{\}$ for set valued oids and it is undefined for non-set valued oids. Value $\perp$ can be used as a selection criterion. For example, query Q_8 looks for the kids of "my friends" in 2000 that (the kids) are "born" after 1990, *i.e.* wich are represented by objects having as a value $\perp$ in the DBV of 1990 and a value different from $\perp$ in one DBV of 2000.

$$\begin{array}{lllll} Q_8: \quad \{\ (x,y) \mid & y \in \texttt{My\_DBVs} & \wedge & \texttt{*(DBV\_Desc(}y\texttt{),}y\texttt{).Date=2000} & \wedge \\ & x \in \texttt{My\_friends(}y\texttt{)} & \wedge & t \in \texttt{*(}x\texttt{,}y\texttt{).Kids} & \wedge \\ & \exists z\ (z \in \texttt{My\_DBVs} & \wedge & \texttt{*(DBV\_Desc(}z\texttt{),}z\texttt{).Date=1990} & \wedge \\ & \quad \mathit{Undef}(\texttt{*(}t\texttt{,}z\texttt{)}) & \wedge & \neg\ (\mathit{Undef}(\texttt{*(}t\texttt{,}y\texttt{}))) &) \quad \} \end{array}$$

in an OQL-like syntax:

```
SELECT   (x, y)
FROM     y in My_DBVs, x in My_friends(y),
         z in My_DBVs
WHERE    (DBV_Desc, y).Date = 2000 and
         (DBV_Desc, z).Date = 1990 and
         t ∈ *(x,y).Kids and
         (t, z) = ⊥ and (t, y) != ⊥
```

5 Related work

In this section the main tools and languages are analysed, which are proposed in the literature for querying databases with versions. There are two categories of works which will be successively presented : works based on version models, and work based on temporal models.

5.1 Version approaches

Most of version models proposed so far provide manipulation primitives in order to create, read and update object versions [7, 16, 22, 23]. However, few papers are concerned with querying databases with versions [23, 14], and they don't allow querying the states of the modeled universe (*cf.* Table 1). The only concept used in these works is entity (or object) versioning. The limits of these models are shown in [12]. For application needs, in some version models the concept of a *context* is introduced, being as a set of object versions [7, 22, 23], in order to make the management of current (or default) versions easier. In [16] contexts are defined as a binding between a version of a composite object and a version of each of its components. In both cases, such concepts of context have not been proposed as an element of a multi-context query language.

Table 1 summarizes the previous contributions. The first column gives a paper reference. The second column indicates whether a query language is proposed or not. The next three columns give an idea on selection possibilities ("–" denotes unmentioned operations). The fifth column concerns keeping track of an object through its versions, and the last column indicates whether a formal support for a language is proposed or not.

Version model	Query language	Object version selection	Generic objet selection	Universe state selection	object tracking	Formal support
1	2	3	4	5	6	7
[7]	no	yes	-	no	-	no
[16]	no	mentioned	-	no	mentioned	no
[22]	no	yes	-	no	-	no
[23]	yes	yes	yes	no	-	no
[3]	yes	yes	no	yes	limited	yes
[14]	yes	yes	-	no	-	no
[28]	yes	yes	-	Temporal model	-	no
[20]	yes	-	-	Temporal model	-	no

Table 1: Version and temporal query tools

In contrast to previous approaches, IQL(2) [3] proposes a formal support integrating its own definition of context and combining different features such as distributed databases, objects with several roles, versions and views. In this model, a database is composed of objects contained in different *contexts*. For instance, in a distributed database with two sites Paris and Los Angeles, each site corresponds to a context.

Two objects contained in different contexts may represent the same entity of the modeled universe. Thus, a context may be compared to one state of the modeled universe. IQL(2) introduces an operator $\equiv$ in order to allow users to determine objects representing the same entity in different contexts. For instance, the following query looks for the phone number in Los Angeles of an employee existing in the Paris context, and called Mary.

```
SELECT  E'.phone
FROM    E in Emp(Paris), E' in Emp(LA)
WHERE   E ≡ E' and E.name = 'Mary'
```

Intuitively, the evaluation is the following. First the object representing employee Mary in the Paris context is selected from the persistent root `Emp(Paris)`. Then, the equivalent object is selected from `Emp(LA)`. However, if Mary is not considered as an employee in Los Angeles, the preceding query can't find her phone number. Moreover, if the user doesn't know the object representing Mary in Los Angeles is attached to which persistent root, querying Mary's phone number will be problematic.

So, in IQL(2), to be able to keep track of an entity through different contexts, user need to know how to access objects representing this entity in each context.

5.2 Temporal approaches

In temporal databases, data varying over time are time-stamped. Thus, implicitly, there are as many universe states represented in the database as there are time units. The common point between temporal languages and VQL is that they are devoted to databases representing simultaneously different states of

the modeled universe. However, temporal languages are based on temporal logic which is founded on the semantics and the specificity of time. A lot of temporal query languages have been proposed in the literature [26, 24, 20, 28]. Most of them are devoted to relational databases [27, 26, 24]. Those devoted to object-oriented databases may be separated in two categories: some of them, like TOOSQL, TOSQL [19] or TMQ [15], carry on the work done for the relational model; others, like OODAPLEX, propose new querying facilities due to object-oriented potential.

TOOSQL

TOOSQL [20] extends SQL to temporal object-oriented databases. It supports two time dimensions, valid time and transaction time, which appear on timestamps associated with attributes. Specific temporal constructs are used in TOOSQL. For instance, the next query retrieves the third change of manager for Mary and the duration over which he/she was Mary's manager.

```
SELECT  A.Manager.Nth(4), A.Manager.Nth(4).Duration(vt)
FROM    A: ADULT
WHERE   A.Name = "Mary"
```

Here, the `Nth(4)` operation returns the 4th manager of Mary. `Duration(vt)` is an operation defined on different time dimensions such as valid time (vt). In this example, it returns the length of the valid time interval during which the 4th manager of Mary didn't change. The FROM clause specifies that variable `A` ranges over class `ADULT`.

Other temporal clauses are defined in TOOSQL (e.g. `WHEN`, `TIME-SLICE`, `ROLLBACK TO`). For instance, the `WHEN` clause returns a time point or a time period over which other conditions specified in the query must hold. The `TIME-SLICE` clause selects only the objects valid during the time period specified in the clause. The `ROLLBACK TO` clause is used to determine the values of an object properties recorded sometime in the past.

OODAPLEX

In OODAPLEX [28], a query is a function mapping objects to objects. For example, the nested query `name(dept(`e`))` returns the name of the department where employee e works. In this model, properties of objects, relationships among objects, and operations on objects are all uniformly modeled by functions, which are applied to objects. Time-varying properties, relationships, or behavior are modeled by functions that return another function mapping time elements into snapshot values of the properties, as shown below.

`function salary (`e`: employee` $\longrightarrow$
`  `f`: ([valid_time: time, transaction_time: time]` $\longrightarrow$ s`: money))`

The following query `salary(`e`)(`t_1`,` t_2`)` returns the employee e's salary at time t_1, as recorded by the database at time t_2.

OODAPLEX introduces quantification over time. Time is treated as a first class object, and variables and quantifiers are allowed to range over time. For instance, the next query returns John's salary when he worked for the Shoes department.

```
for each e in Extent(employee) where name(e) = 'John'
     for each t where name(dept(e)(t)) = 'Chaussures'
           salary(e)(t)
     end
end
```

In contrast to a language such as TOOSQL, OODAPLEX proposes no special constructs. The retrieval of temporal and non-temporal information is uniformly expressed. By allowing variables and quantifiers to range over time, queries that require special operators, like *when* or *shift*, in other languages can be formulated naturally.

6 Conclusion - Future work

In this paper we have proposed a query language VQL for multiversion databases, based on a sound formalism. The DBV concept, corresponding to the modeled universe states, is integrated into both the underlying data model (Section 3) and VQL formal support (Section 4.1). This is done in order to allow users to query simultaneously object versions and the states of the modeled universe where these versions

appear. The result is a straightforward language allowing users to easily formulate queries which may be complex. An implementation of VQL on a version manager corresponding to the DBV model is under development.

An important characteristic of VQL is its generality: no special semantics is associated with versions, in contrast to temporal languages. For instance, in a CASE application where DBVs correspond to software configurations, DBVs may be associated with their creator, with their owner, with valid time, with hardware support, with software choices, etc. Thus, several version dimensions may be considered in an application. Some of these dimensions may be ordered (e.g. time), while others may be not ordered (e.g. hardware support). The DBV descriptor (DBV_Desc, *cf.* Example 3.1) can be used to combine information related to different version dimensions. It is also possible to have many DBV descriptors, each one related to a specific version dimension or combining some of them.

From the language point of view, there are two main contributions of VQL: (1) specific terms to denote the modeled universe states (DBVs), and (2) a dereferencing operation taking into account the "DBV dimension". The first one permits to query the states of a modeled universe. The standard predicates $\in$ and $=$ are applied on terms denoting DBVs, and no restriction is placed on the quantification on DBVs. The second one makes it possible to keep the track of an object through DBVs (*i.e.* through different states of the modeled universe). A new predicate *Undef* (t_1, t_2), where t_1 denotes an object and t_2 denotes a DBV, is introduced to enable user to determine logical object versions having a $\perp$ value.

VQL will be used as a support to several current works on multiversion databases such as constraint expression, view definition and querying in the case of a versioned schema. Constraints on a multiversion database may be "internal" to a DBV, as they may be applied on several DBVs [10]. For example, the fact that "my parents" are always the same, whatever is the DBV, is a constraint on several DBVs (*cf.* Example 3.1). Expressing such a constraint can be naturally done in VQL.

Queries on multiversion databases can be used for view definition, as queries on monoversion databases. A view may be used to restrict the vision of the user to some elements (DBVs, objects, logical object versions or values) stored in the database. It may also be used to construct new elements, imaginary elements [21], computed from the ones stored in the database. The output instance of query Q_4, described in Section 4.2, is an example of a view restrained to the logical object versions representing "my friends" (*cf.* Figure 3).

To take into account schema versioning, VQL requires extensions to the data model described in Section 3, as well as an appropriate type checking technique.

Moreover, manipulation operations will be added to VQL, in order to develop a complete database programming language. These operations have been presented in [11].

Finally, an implementation of the DBV model has been done on a relational system. In this case too, a manipulation language is required, as an extension to SQL. The concepts developed in VQL will be transposed to the relational framework.

References

[1] S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *The VLDB Journal*, Volume 4, Number 4, pages 727–794, October 1995.

[2] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 159–173, Portland, Oregon, 31 May–2 June 1989.

[3] S. Abiteboul and C. Souza. IQL(2): A model with ubiquitous objects. In *Fifth International Workshop on Database Programming Languages (DBPL 95)*, Gubbio, Umbria (Italy), 1995. Springer-Verlag, Workshops on Computing.

[4] F. Bancilhon, S. Cluet and C. Delobel. A Query Language for O2. In *Bulding an Object-Oriented Database System, the story of O2*, pages 234–277. Morgan Kaufmann Publishers, 1992.

[5] R.G.G. Cattell (editor). *The Object Database Standard: ODMG-93*, Chapter 4: Object Query Language, pages 53–85. Mogan Kaufmann Publishers, 1996. release 1.2.

[6] W. Cellary and G. Jomier. Consistency of Versions in Object-Oriented Databases. In *Proceedings of the 16th VLDB Conference*, pages 432–441, Brisbane, Australia, 1990.

[7] H. T. Chou and W. Kim. A unifying framework for version control in a CAD environment. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 336–344, Kyoto, August 1986.

[8] V. Christophides, S. Abiteboul, S. Cluet and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 313–324, Minneapolis, Minnesota, 24–27 May 1994.

[9] V. Christophides, S. Cluet and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 413–422, Montreal, Quebec, Canada, 4–6 June 1996.

[10] A. Doucet, S. Gançarski, G. Jomier and S. Monties. Maintien de la cohérence dans une base de données multiversion. In *Proc. 12-èmes journées Bases de Données Avancées (BDA 96)*, Cassis (France), 1996.

[11] S. Gançarski. Versions et bases de données : modèle formel, supports de langage et d'interface-utilisateur. Ph.D. thesis, Paris-Sud University, Centre d'Orsay, France, 1994.

[12] S. Gançarski and G. Jomier. Gestion des versions d'entités et de leur contexte : analyse et perspectives. *Ingénierie des Systèmes d'Information*, Volume 3, Number 6, pages 677–711, 1995.

[13] S. Gançarski and G. Jomier. Un Formalisme pour la Gestion de Versions d'Entité. In *Proc. 10-èmes journées Bases de Données Avancées (BDA 94)*, France, 1994.

[14] G. Hubert. Les versions dans les bases de données orientées objet : modélisation et manipulation. Ph.D. thesis, Paul Sabatier University, Toulouse (France), January 1997.

[15] W. Käfer and H. Schöning. Realizing a temporal complex-object data model. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, Volume 21(2) of *SIGMOD Record*, pages 266–275, New York, NY, USA, June 1992. ACM Press.

[16] R. H. Katz. Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, Volume 22, Number 4, pages 375–408, December 1990.

[17] M. Kifer, W. Kim and Y. Sagiv. Querying object oriented databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, Volume 21(2) of *SIGMOD Record*, pages 393–402, New York, NY, USA, June 1992. ACM Press.

[18] O2 Technology. *OQL User Manual, release 4.6*, 1996.

[19] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. 10th Int. Conf. on the Entity-Relationship approach*, 1991.

[20] E. Rose and A. Segev. TOOSQL - A Temporal Object-Oriented Query Language. In *Proceedings of the 12th International Conference on the Entity-Relationship approach*, LNCS series, pages 122–136, Arlington (Texas), December 1993. Springer-Verlag.

[21] C. Santos, C. Delobel and S. Abiteboul. Virtual Schemas and Bases. In *Proceedings of the 1994 International Conference on Extending Database Technology (EDBT 94*, Cambridge, March 1994.

[22] E. Sciore. Multidimensional versioning for object–oriented databases. In C. Delobel, M. Kifer and Y. Masunaga (editors), *Proceedings of Deductive and Object–Oriented Databases (DOOD 91)*, Volume 566 of *LNCS*, pages 355–370, Berlin, Germany, December 1991. Springer-Verlag.

[23] E. Sciore. Versioning and configuration management in an object-oriented data model. *The VLDB Journal*, Volume 3, Number 1, pages 77–106, January 1994.

[24] R. T. Snodgrass. An overview of the temporal query language TQuel. Technical Report TR 92-22, University of Arizona, August 1992.

[25] R. T. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim (editor), *Modern Database Systems, the object model, interoperability, and beyond*, Chapter 19, pages 386–408. Addison-Wesley Publishing Company, 1995.

[26] R. T. Snodgrass (editor). *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[27] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. T. Snodgrass (editors). *Temporal Databases: theory, design, and implementation.* Benjamin/Cummings, 1993.

[28] G. T. J. Wuu and U. Dayal. A uniform model for temporal object-oriented databases. In *Proceeding of the IEEE 8th International Conference on Data Engineering*, pages 584–593, Tempe (Arizona), February 1992. IEEE Computer Society Press.