Ersch. in: Objects and databases : Third International Conference, ICOODB 2010, Frankfurt/Main, Germany, September 28 - 30, 2010; proceedings / Alan Dearle ... (eds.). - Berlin [u.a.] : Springer, 2010. - S. 55-69. - (Lecture notes in computer science ; 6348). - ISBN 978-3-642-16091-2

A Flexible Object Model and Algebra for Uniform Access to Object Databases

Michael Grossniklaus[1], Alexandre de Spindler[2], Christoph Zimmerli[2], and Moira C. Norrie[2]

[1] Dipartimento di Elettronica e Informazione, Politecnico di Milano
I-20133 Milano, Italy
grossniklaus@elet.polimi.it

[2] Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{despindler,zimmerli,norrie}@inf.ethz.ch

Abstract. In contrast to their relational counterparts, object databases are more heterogeneous in terms of their architecture, data model and functionality. To this day, this heterogeneity poses substantial difficulties when it comes to benchmark or interoperate object databases. While standardisation proposals have been made in the past, they have had limited impact as neither industry nor research has fully adopted them. We believe that one reason for this lack of adoption is that these standards were too restrictive and thus not capable of dealing with the heterogeneity of object databases. In this paper, we propose a uniform interface for access to object databases that is based on a flexible object model and algebra.

1 Introduction

Since their emergence in the 1980s, object databases have always been heterogeneous to an extent far greater than their relational siblings. One reason for heterogeneity is the fact that object databases are situated at the intersection of database management and object-oriented systems [1]. As a consequence, different object databases provide different sets of capabilities depending on their origin. On a very general level, the two approaches can be characterised in terms of whether they aim at supporting the compile-time or the run-time of an object data management system. Typically, object-oriented systems focus on aspects related to the design and development, whereas database management systems also address issues related to operation and evolution.

This difference is most pronounced in the object data models on which these systems are based. Models originating from object-oriented systems emphasise aspects such as encapsulation and language integration [2] and, since their main goal is to persist the objects of a programming language, these data models are usually very similar to, or even tied in with, the one of the language. In contrast, models that emerged from database management are designed to support traditional database features such as concurrency and recovery through transactions and to efficiently query large object graphs. Additionally, these models tend to address issues related to the longevity of data and, therefore, provide features to support object and schema evolution such as roles and dynamic typing.

While the different origins have led to a diverse palette of systems that are all uniquely suited to address specific application requirements, they have also hindered interoperability, data exchange, performance evaluation and, as argued by Greene [3], ultimately market adoption. Early on, efforts to rectify this situation have been undertaken in terms of defining [1], benchmarking [4,5] and standardising [6] object databases. And even though these attempts have all made important contributions, they have failed to fully deliver on the hopes invested in them. Successful object databases have become so by occupying niche markets and expanding from there, rather than by following definitions and implementing standards. We believe that one reason for this lack of adoption is that the proposals were too restrictive in the sense that the trade-off between a common core and individual strengths was not well balanced.

Nevertheless, as object databases have recently gained importance in both academia and industry, it is critical to also resume these standardisation efforts. This requirement has also been identified by the Object Management Group (OMG) which recently formed a working group to develop the next-generation object database standard [7]. We believe that the current proposal is far too generic and, in this paper, propose an alternative object model and algebra that offers a better trade-off between diversity and specificity. In the context of this model, we have also defined an algebra that supports both unordered and ordered collections with or without duplicates. Based on this model and algebra, we propose an interface to provide uniform access to object databases.

We begin in Sect. 2 with the background and discussion of related work. The object data model and corresponding algebra are presented in Sect. 3 and Sect. 4, respectively. In Sect. 5, we discuss a prototype implementation of the proposed interface that serves as a proof-of-concept. The contributions of this work as well as open issues are discussed in Sect. 6 and we conclude in Sect. 7.

2 Background

Several efforts to standardise object databases in terms of object data models and algebras have been made in the past or are still ongoing. We start by summarising the most influential approaches, before introducing the background of the object representation used in our proposal.

The best-known object database standard was defined by the Object Data Management Group (ODMG) [6]. Its object data model is based on the OMG object model and distinguishes between modelling primitives with and without unique identifier, called objects and literals, respectively. An object has a state comprised by its attributes and relationships as well as behaviour given by its methods. Objects are defined by types that consist of a specification and an implementation part. The former defines the abstract state and behaviour, while the latter furnishes a concrete realisation of the specification through a language binding. Abstract types are specified in terms of interfaces that define abstract behaviour and classes that define abstract state and behaviour. For classes, the model supports only single inheritance, whereas for interfaces multiple inheritance is allowed. Finally, predefined collection types such as set, bag, list, array and dictionary are available both as objects and as literals.

Following the renewed interest in object databases, OMG recently resumed standardisation efforts and formed the object database technology working group. The proposal in the current white paper [7] is based on a Stack-Based Architecture (SBA) [8] that features a storage model and a query language. The storage model uses ⟨*subject*, *predicate*, *object*⟩ triples to represent objects. The formalisation of this model is straightforward and therefore its main advantage. However, we believe that the fact that it is not specific to object databases and hence does not capture their essential features makes it unsuitable as a standard. It has been shown that storages based on triples are generic to the point of being able to represent any data model [9]. As a consequence, the current proposal has to be considered a step backwards as its low level of granularity cannot compete with earlier and semantically richer models, such as OEM [10] that uses quadruples to represent objects or the previously discussed ODMG data model.

In order to interact with object data, algebras and query languages have been defined in addition to data models. The Object Query Language (OQL) [6] was defined within the ODMG standard. OQL is a declarative query language with a syntax similar to SQL. The semantics, however, is quite different as OQL operates on sets of objects and is capable of handling path expressions. Unlike the ODMG data model that is supported by some vendors, OQL has not seen widespread adoption. Today, the Versant Query Language (VQL) [11] represents the most complete implementation, even though it only supports a very limited subset of OQL. The Stack-Based Query Language (SBQL) [8] is based on an algebra that complements the stack-based architecture introduced above. SBQL queries can be expressed using its proprietary syntax or through SBQL4J, a language-integrated query interface for the Java programming language. The latter is again confirmation of the fact that there is a trend in object databases to integrate the query language with the programming language. This approach has been pioneered by Microsoft's Language-Integrated Query (LINQ) [12,13] which is capable of accessing object, relational and XML data uniformly. Other approaches that fall into this category are db4o's programmatic query interfaces [14], namely Native Query (NQ) and Simple Object Data Access (SODA). Acknowledging this development, we are convinced that a future object database standard should specify a programmatic or language-integrated query interface, rather than a stand-alone query language.

The object model and algebra that we propose as the theoretical foundation for building a standardised interface to object databases is based on object-slicing [15]. An object representation that uses the object-slicing technique is a suitable basis for a standard as it is flexible enough to capture the diversity of object databases while, at the same time, specific enough to address their unique requirements. For example, it can uniformly represent object models regardless of whether they use single or multiple inheritance and whether multiple instantiation is possible or not [16]. In the past, object-slicing has, therefore, been proposed as an implementation technique to support features such as views, schema evolution, versions and roles. MultiView [17,18] is an implementation of object-slicing on top of GemStone and has been applied both to object-oriented views and schema evolution. While MultiView implements object-slicing based on an object database, Iris [19] follows the same approach but uses a relational back-end to store its objects. This approach is similar to more recent Object-Relational Mapping (ORM) tools that also persist objects in relational databases using model mapping [20]. However, while MultiView and Iris assume a fixed mapping between classes and so-called implementation objects, Hibernate [21], for example, offers several mapping strategies to define how objects are stored.

In summary, previous work has focused on object-slicing at the implementation level to support advanced features and to store objects flexibly. In contrast, our proposal is to leverage object-slicing at the conceptual level to unify the different approaches that exist. Unlike earlier standards, our approach recognises the importance of having diverse object databases. Therefore, our main goal is not to limit these systems by forcing them to adopt a restrictive interface. On the contrary, we propose a uniform and consistent interface to object databases that could easily be implemented by existing systems. As a consequence, the focus of our interface is more on data exchange and benchmarking, rather than application development and portability.

3 Object Data Model

In this section, we present an object data model based on object-slicing [15]. Figure 1 introduces the example used to illustrate our approach. The left shows a class hierarchy, rooted at class `Contact` with subclasses `Organisation`, `Person` and `Private`. The graphical representation of two objects based on object-slicing is given on the right. Object id_1 is an instance of class `Person`, whereas object id_2 is an instance of class `Organisation`. As can be seen, both objects consist of two so-called *object slices* which we refer to as *information units*. Each information unit corresponds to exactly one class and stores the attribute values for the fields declared by that class. Object instantiation in our model is captured by the *dress* and *strip* primitives that add or remove information units, respectively. As shown in the figure, object id_1 can be instantiated with class `Private` using a *dress* operation, whereas object id_1 could be reclassified as an instance of `Contact` using a *strip* operation. Based on this representation, we now present the formal definition of the object data model.

The type system of our object data model distinguishes four different kinds of types—base, object, structured and extent types—that describe the domain $\mathbb{T}$ of all possible values $\mathbb{V}$. Let $\mathbb{T}^* = \{\mathbb{T}_{base}, \mathbb{T}_{obj}, \mathbb{T}_{struct}, \mathbb{T}_{ext}\}$, then $\forall\, \mathbb{T}_i, \mathbb{T}_j \in \mathbb{T}^* : \mathbb{T}_i \neq \mathbb{T}_j : \mathbb{T}_i \cap \mathbb{T}_j = \emptyset$ and $\mathbb{T} = \bigcup_{\mathbb{T}_i \in \mathbb{T}^*} \mathbb{T}_i$. We will describe each of these types in more detail.

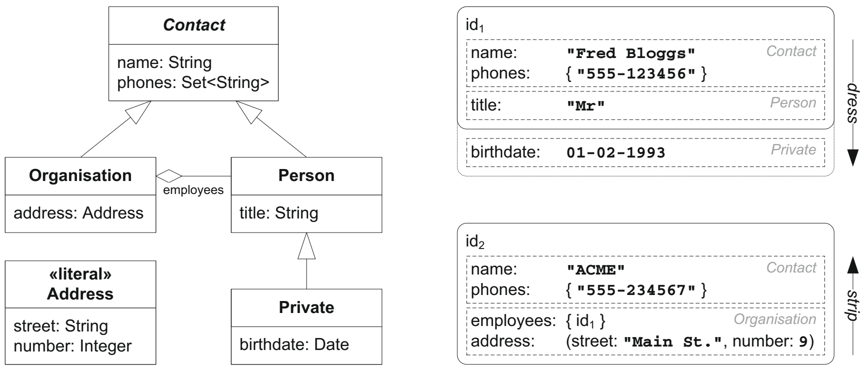


Fig. 1. Overview of our approach

Base Types. A base type $T_{base} \in \mathbb{T}_{base}$ defines the, possibly infinite, domain of a basic type that is predefined by the object database. As a consequence, $\mathbb{T}_{base}$ can change from one system to another. For the scope of this paper, we assume the following definition.

$$\mathbb{T}_{base} = \{\textsc{boolean}, \textsc{integer}, \textsc{real}, \textsc{date}, \textsc{string}\}$$

We use the names of the base type as a short-hand to denote their value domains. For example, BOOLEAN is used to denote $T_{boolean} = \{\mathsf{true}, \mathsf{false}\}$.

A *base value*, $v_{base} \in \mathbb{V}_{base}$, has no identity and is said to be an instance of a base type $T_{base} \in \mathbb{T}_{base}$, denoted as $v_{base} \vdash T_{base}$, iff $v_{base} \in T_{base}$. Generally, base types and their instances cannot be explicitly created, modified or deleted as their existence is taken for granted.

Object Types. An object type $T_{obj} \in \mathbb{T}_{obj}$ describes the properties of a class of objects.[1] It is defined as a set of field names $\{F_1, F_2, \ldots, F_n\}$ each of which is associated with a type $T_i \in \mathbb{T}$, where $1 \leq i \leq n$.

An object type T_{sub} can be a subtype of one or more object types T_{super}, denoted as $T_{sub} \sqsubset T_{super}$. The relation $\sqsubset$ is transitive, i.e. $T_1 \sqsubset T_2 \wedge T_2 \sqsubset T_3 \Rightarrow T_1 \sqsubset T_3$ and reflexive, i.e. $T_{obj} \sqsubset T_{obj}$. Based on these properties, we define

$$T^*_{obj} = \bigcup_{\forall\, T_i \in \mathbb{T}_{obj} \,:\, T_{obj} \sqsubset T_i} T_i$$

to be the set of all defined and inherited field names of an object type T_{obj}.[2]

An *object*, $v_{obj} \in \mathbb{V}_{obj}$, is defined as the structure $\langle id, \Omega \rangle$, where id is the object's unique and immutable identifier and $\Omega = \{\mu \mid \mu : T_{obj} \to \mathbb{V}\}$ is a set

[1] Due to space limitations, we omit the discussion of methods in this paper.

[2] Note that the precise definition of the set T^*_{obj} depends on the model of inheritance used by the object database. Since our object model does not preclude any inheritance model, different systems may return different sets.

of mappings. Each mapping $\mu : (F_1 = v_1, F_2 = v_2, \ldots, F_n = v_n)$ is a function relating field names $F_i \in T_{obj}$ to values $v_i \in \mathbb{V}$ with the restriction that $\mu(F_i) \vdash T_i$. We say a mapping μ satisfies T_{obj}, denoted by $\mu \models T_{obj}$, iff $\forall\, F_i \in T_{obj}, \exists\, v \in \mathbb{V} : \mu(F_i) = v$. An object $v_{obj} = \langle id, \Omega \rangle$ is said to be an instance of T_{obj}, denoted as $v_{obj} \vdash T_{obj}$, iff $\forall\, T_i \in \mathbb{T}_{obj} : T_{obj} \sqsubset T_i, \exists\, \mu \in \Omega : \mu \models T_i$. Mappings correspond to the information units introduced earlier.

Both object types and objects can be created, modified and deleted. Due to space limitations, we limit our presentation to the *dress*, *strip* and *browse* operations that are specific to our object data model. A more comprehensive discussion can be found in [22]. The *dress* and *strip* operations are used respectively to add or remove information units to or from an object, while the *browse* operation computes a mapping that represents the object in the context of the given type.

$$\begin{aligned}
dress(\langle id, \Omega\rangle, T_{obj}) &: \mathsf{if}\ \nexists\, \mu \in \Omega : \mu \models T_{obj}\ \mathsf{then}\ \Omega := \Omega \cup \{\mu_{new}\}\ \mathsf{end} \\
strip(\langle id, \Omega\rangle, T_{obj}) &: \mathsf{if}\ \exists\, \mu \in \Omega : \mu \models T_{obj}\ \mathsf{then}\ \Omega := \Omega \backslash \{\mu\}\ \mathsf{end} \\
browse(\langle id, \Omega\rangle, T_{obj}) &: \mathsf{return}\ \mu : \mu \models T^{*}_{obj}
\end{aligned}$$

Structured Types. A structured type $T_{struct} \in \mathbb{T}_{struct}$ describes the structure of literals. Similar to object types, structured types are defined as a set of field names $\{F_1, F_2, \ldots, F_n\}$ where each F_i is associated with a type $T_i \in \mathbb{T}$, where $1 \leq i \leq n$. In contrast to object types, structured types cannot define methods and there is no notion of subtyping or inheritance.

Since a *structured value* or *struct*, $v_{struct} \in \mathbb{V}_{struct}$, has no identity, it is simply defined as a mapping $\mu : T_{struct} \rightarrow \mathbb{V}$, denoted as $(F_1 = v_1, F_2 = v_2, \ldots, F_n = v_n)$, where $\mu(F_i) \vdash T_i$. We say a structured value $v_{struct} = \mu$ is an instance of T_{struct}, denoted as $v_{struct} \vdash T_{struct}$, iff $\mu \models T_{struct}$, where $\mu \models T_{struct} \Leftrightarrow \forall\, F_i \in T_{struct}, \exists\, v \in \mathbb{V} : \mu(F_i) = v$.

Extent Types. An extent type $T_{ext} \in \mathbb{T}_{ext}$ describes a collection of values in terms of its bulk behaviour and the type of its members. Accordingly, it is defined as a structure $\langle bulk, T\rangle$, where $bulk \in \{\mathsf{set}, \mathsf{bag}, \mathsf{ranking}, \mathsf{sequence}\}$ and $T \in \mathbb{T}$.

An *extent value* or *extent*, $v_{ext} \in \mathbb{V}_{ext}$, for an extent type $T_{ext} = \langle bulk, T\rangle$ is a finite collection of values, denoted as $v_{ext} = \langle\!\langle v_1, v_2, \ldots, v_n \rangle\!\rangle$. Corresponding to the four bulk behaviours introduced above, we distinguish set, bag, ranking and sequence extent values, depending on whether they are ordered and allow duplicates. We denote a set (unordered, no duplicates) as $v_{set} = \{v_1, v_2, \ldots, v_n\}$, a bag (unordered, duplicates) as $v_{bag} = \wr v_1, v_2, \ldots, v_n \int$, a ranking (ordered, no duplicates) as $v_{rnk} = \lceil v_1, v_2, \ldots, v_n \rceil$, and a sequence (ordered, duplicates) as $v_{seq} = [v_1, v_2, \ldots, v_n]$. An extent value v_{ext} is an instance of an extent type $T_{ext} = \langle bulk, T\rangle$, denoted as $v_{ext} \vdash T_{ext}$, iff its behaviour matches *bulk* and $\forall\, v \in v_{ext} : v \vdash T$. We will discuss the operations defined over collections of values in the next section.

Example. For the example introduced in Fig. 1, the representation of a database containing objects id_1 and id_2 based on the formal object data model is given by

$$\mathbb{V} = \{\langle id_1, \{\mu_1^{contact}, \mu_1^{person}\}\rangle, \langle id_2, \{\mu_2^{contact}, \mu_2^{organisation}\}\rangle\},$$

where

$$\begin{aligned}
\mu_1^{contact} &: (name = \texttt{"Fred Bloggs"}, phones = \{\texttt{"555-123456"}\}) \\
\mu_1^{person} &: (title = \texttt{"Mr"}) \\
\mu_2^{contact} &: (name = \texttt{"ACME"}, phones = \{\texttt{"555-234567"}\}) \\
\mu_2^{organisation} &: (address = (street = \texttt{"Main St."}, number = 9), employees = \{id_1\}).
\end{aligned}$$

4 Collection Algebra

We now present the algebra associated with our model. Since, for the most part, its operators apply to collections of values, i.e. extent values, we refer to it as a collection algebra. Our algebra is an extension of traditional set algebra as it introduces functionality specific to object data management and provides support for collections other than sets. However, in order to define how these operators manipulate collections of values, we first need to specify their behaviour in terms of the type system of our object data model.

Table 1. Most-specific types

(a) Base types

$\sqcup$	BOOLEAN	INTEGER	REAL	DATE	STRING
BOOLEAN	BOOLEAN	$\perp$	$\perp$	$\perp$	STRING
INTEGER		INTEGER	REAL	$\perp$	STRING
REAL			REAL	$\perp$	STRING
DATE				DATE	STRING
STRING					STRING

(b) Extent types

$\sqcup$	set	bag	ranking	sequence
set	set	set	set	set
bag		bag	set	bag
ranking			ranking	ranking
sequence				sequence

Most-specific Type. We define the most-specific type of two types T_1 and T_2, denoted as $\widehat{T} = T_1 \sqcup T_2$, where $T_1, T_2 \in \mathbb{T}_i$ and $\mathbb{T}_i \in \mathbb{T}^*$. In the case that $\mathbb{T}_i = \mathbb{T}_{base}$, the most-specific type of two base types is defined by Tab. 1(a), where $\perp$ stands for undefined. The most-specific type of two object types $T_1, T_2 \in \mathbb{T}_{obj}$ is defined as

$$\widehat{T} = T_1 \sqcup T_2 \Leftrightarrow T_1 \sqsubset \widehat{T} \wedge T_2 \sqsubset \widehat{T} \wedge (\nexists T_i \neq \widehat{T} : T_1 \sqsubset T_i \sqsubset \widehat{T} \wedge T_2 \sqsubset T_i \sqsubset \widehat{T}).$$

If $\mathbb{T}_i = \mathbb{T}_{struct}$, the most-specific type of two structured types T_1 and T_2 is defined as follows. Let $T_1 = \{F_1^1, F_2^1, \ldots, F_n^1\}$ with associated types T_i^1, where $1 \leq i \leq n$ and $T_2 = \{F_1^2, F_2^2, \ldots, F_m^2\}$ with associated types T_j^2, where $1 \leq j \leq m$. If $n = m$ and $\forall_{1 \leq k \leq n} F_k^1, F_k^2 : F_k^1 = F_k^2$, then $\widehat{T} = T_1 \sqcup T_2$ is given as the set of field names $\{F_1 = F_1^1, F_2 = F_2^1, \ldots, F_n = F_n^1\}$ with associated types $T_i = T_i^1 \sqcup T_i^2$, $1 \leq i \leq n$. Finally, in the case that $T_1 = \langle bulk_1, T_1' \rangle$ and $T_2 = \langle bulk_2, T_2' \rangle \in \mathbb{T}_{ext}$, the most-specific type of two extent types is given by the structure $\widehat{T} = \langle bulk, T' \rangle$, where $bulk = bulk_1 \sqcup bulk_2$, according to Tab. 1(b) and $T' = T_1' \sqcup T_2'$. In all other cases, the most-specific type of two types is undefined ($\perp$).

Type Compatibility. Two types T_i and T_j are said to be compatible, denoted as $T_i \sim T_j$, iff $T_i \sqcup T_j \neq \perp$.

Support Operations. Finally, we introduce the following operations to support the definition of operators over ordered collections. For an ordered collection $C = \langle\!\langle x|C' \rangle\!\rangle$, the $|$ operator decomposes C into its first element x and the ordered collection of the remaining elements C'. The operation $\mathit{append}(C, x) : (\mathsf{coll}[T], T) \rightarrow \mathsf{coll}[T]$ inserts an element x at the end of an ordered collection C. The operation $\mathit{remove}(C, x) : (\mathsf{coll}[T], T) \rightarrow \mathsf{coll}[T]$ removes the element x with the smallest index from the ordered collection C.

Note that we will use the set representation of bags in some of the following definitions, where $\wr 1, 1, 1, 2, 2, 3 \wr \equiv \{(1, 3), (2, 2), (3, 1)\}$. Then we use $x \in_{bag} B$ to denote the membership of x in a bag B and $(x, n) \in_{set} B$ to denote the membership of (x, n) in the set representation of B where n is an integer giving the number of occurrences of x. The full definition of collection membership $\in : (T, \mathsf{coll}[T]) \rightarrow \text{BOOLEAN}$, is given below.

$$\begin{array}{ll} x \in_{set} S \ = x \in S & x \in_{bag} B = \exists n : (x, n) \in_{set} B \wedge n > 0 \\ x \in_{rnk} R = \exists i : R[i] = x & x \in_{seq} Q = \exists i : Q[i] = x \end{array}$$

Finally, we also include a definition of bag addition here, which will be used to define other operators over bags that are part of the collection algebra.

$$B_1 \uplus B_2 = \{(x, y) \mid \exists n_1, n_2 : (x, n_1) \in_{set} B_1 \wedge (x, n_2) \in_{set} B_2 \wedge n = n_1 + n_2)\}$$

Collection Operations. The extent operation, $\otimes : T \rightarrow \mathsf{coll}[T]$, where $T \in \mathbb{T}_{obj}$, returns all objects v_{obj} in the databases, such that $v_{obj} \vdash T$.

The union, $\cup : (\mathsf{coll}[t_1], \mathsf{coll}[t_2]) \rightarrow \mathsf{coll}[t_1 \sqcup t_2]$, of two collections is defined as follows.

$$S_1 \cup_{set} S_2 = \{x \mid x \in_{set} S_1 \vee x \in_{set} S_2\}$$

$$B_1 \cup_{bag} B_2 = \{(x, n) \mid \exists n_1, n_2 : (x, n_1) \in_{set} B_1 \wedge (x, n_2) \in_{set} B_2 \wedge n{=}\mathit{max}(n_1, n_2)\}$$

$$R_1 \cup_{rnk} R_2 = \begin{cases} R_1 & \text{if } R_2 = \emptyset \\ \mathit{append}(R_1, x) \cup_{rnk} R_2', \text{where } R_2 = \lceil x|R_2' \rceil & \text{otherwise} \end{cases}$$

$$Q_1 \cup_{seq} Q_2 = \begin{cases} Q_1 & \text{if } Q_2 = \emptyset \\ \mathit{append}(Q_1, x) \cup_{seq} Q_2', \text{where } Q_2 = [x|Q_2'] & \text{otherwise} \end{cases}$$

The definition of the intersection, $\cap : (\mathsf{coll}[t_1], \mathsf{coll}[t_2]) \rightarrow \mathsf{coll}[t_1 \sqcup t_2]$, of two collections is given below.

$$S_1 \cap_{set} S_2 = \{x \mid x \in_{set} S_1 \wedge x \in_{set} S_2\}$$

$$B_1 \cap_{bag} B_2 = \{(x, n) \mid \exists n_1, n_2 : (x, n_1) \in_{bag} B_1 \wedge (x, n_2) \in_{bag} B_2 \wedge n{=}\mathit{min}(n_1, n_2)\}$$

$$R_1 \cap_{rnk} R_2 = \begin{cases} \emptyset & \text{if } R_1 = \emptyset \\ \lceil x|(R_1' \cap_{rnk} R_2) \rceil, \text{where } R_1 = \lceil x|R_1' \rceil & \text{if } x \in_{rnk} R_2 \\ R_1' \cap_{rnk} R_2, \text{where } R_1 = \lceil x|R_1' \rceil & \text{otherwise} \end{cases}$$

$$Q_1 \cap_{seq} Q_2 = \begin{cases} \emptyset & \text{if } Q_1 = \emptyset \\ [x|(Q_1' \cap_{seq} \mathit{remove}(Q_2, x))], \text{where } Q_1 = [x|Q_1'] & \text{if } x \in_{seq} Q_2 \\ Q_1' \cap_{seq} Q_2, \text{where } Q_1 = [x|Q_1'] & \text{otherwise} \end{cases}$$

The following definition specifies the difference, $- : (\mathsf{coll}[t_1], \mathsf{coll}[t_2]) \rightarrow \mathsf{coll}[t_1]$, of two collections.

$$\begin{aligned}
S_1 -_{set} S_2 &= \{x \mid x \in_{set} S_1 \wedge x \notin_{set} S_2\} \\
B_1 -_{bag} B_2 &= \{(x,n) \mid \exists n_1 : (x,n_1) \in_{set} B_1 \wedge \\
&\qquad ((x \notin_{bag} B_2 \wedge n = n_1) \vee \exists n_2 : (x,n_2) \in_{set} B_2 \wedge n = n_1 - n_2)\} \\
R_1 -_{rnk} R_2 &= \begin{cases} R_1 & \text{if } R_2 = \emptyset \\ \mathit{remove}(R_1, x) -_{rnk} R_2', \text{where } R_2 = \lceil x|R_2' \rceil & \text{if } x \in_{rnk} R_1 \\ R_1 -_{rnk} R_2', \text{where } R_2 = \lceil x|R_2' \rceil & \text{otherwise} \end{cases} \\
Q_1 -_{seq} Q_2 &= \begin{cases} Q_1 & \text{if } Q_2 = \emptyset \\ \mathit{remove}(Q_1, x) -_{seq} Q_2', \text{where } Q_2 = [x|Q_2'] & \text{if } x \in_{seq} Q_1 \\ Q_1 -_{seq} Q_2', \text{where } Q_2 = [x|Q_2'] & \text{otherwise} \end{cases}
\end{aligned}$$

Selection. The selection operation, $\sigma : (\mathsf{coll}[t], t \rightarrow \text{BOOLEAN}) \rightarrow \mathsf{coll}[t]$, forms a subcollection of a given collection C that only contains elements that satisfy a predicate p. Using the reduce operation ($\rhd$), which will be introduced later, it is defined as follows.

$$\begin{aligned}
\sigma_{set}\, p\, S &= \{x \mid x \in_{set} S \wedge p(x) = \mathsf{true}\} \\
\sigma_{bag}\, p\, B &= \{(x,n) \mid (x,n) \in_{set} B \wedge p(x) = \mathsf{true}\} \\
\sigma_{rnk}\, p\, R &= \rhd_{rnk}\ \lambda(x, R').(\mathsf{if}\ p(x)\ \mathsf{then}\ \lceil x \rceil \cup_{rnk} R'\ \mathsf{else}\ R')\ \emptyset\ R \\
\sigma_{seq}\, p\, Q &= \rhd_{seq}\ \lambda(x, Q').(\mathsf{if}\ p(x)\ \mathsf{then}\ [x] \cup_{seq} Q'\ \mathsf{else}\ Q')\ \emptyset\ Q
\end{aligned}$$

Map Operations. Our algebra also supports map operations that apply a given function f to all members of a collection C and return a new collection containing the results of this function application. The general map operator, $\lhd : (\mathsf{coll}[t_1], t_1 \rightarrow t_2) \rightarrow \mathsf{coll}[t_2]$, is given as follows.

$$\begin{aligned}
\lhd_{set}\ f\ S &= \{f(x) \mid x \in_{set} S\} \\
\lhd_{bag}\ f\ B &= \rhd_{bag}\ \lambda((x,n), B').(\{(f(x), n)\} \uplus B')\ \emptyset\ B \\
\lhd_{rnk}\ f\ R &= \rhd_{rnk}\ \lambda(x, R').(\lceil f(x) \rceil \cup_{rnk} R')\ \emptyset\ R \\
\lhd_{seq}\ f\ Q &= \rhd_{seq}\ \lambda(x, Q').([f(x)] \cup_{seq} Q')\ \emptyset\ Q
\end{aligned}$$

The navigation operation, $\cdot : (\mathsf{coll}[T], F_i) \rightarrow \mathsf{coll}[T_i]$, where $T \in \mathbb{T}_{obj}$, $F_i \in T$ and $\mu(F_i) \vdash T_i$, is a special case of a map operation that substitutes each object $x = \langle id, \Omega \rangle$ with the value of its field F_i, denoted as $x.F_i = \mu(F_i)$, where $\mu \in \Omega$.

$$\begin{array}{ll}
S\ \cdot_{set}\ F\ = \lhd_{set}\ \lambda x.(x.F)\ S & \quad B\ \cdot_{bag}\ F = \lhd_{bag}\ \lambda x.(x.F)\ S \\
R\ \cdot_{rnk}\ F = \lhd_{rnk}\ \lambda x.(x.F)\ S & \quad Q\ \cdot_{seq}\ F = \lhd_{seq}\ \lambda x.(x.F)\ S
\end{array}$$

Reduce Operations. The last group of operators provided in our algebra are reduce operations which, given an aggregation function f and a default value v, compute one or more aggregated values over a collection C. The general reduce operator, $\rhd : (\mathsf{coll}[t_1], ((t_1, t_2) \rightarrow t_2), t_2) \rightarrow t_2$ is defined as follows.

$$\begin{aligned}
\rhd_{set}\ f\ v\ S &= \mathsf{if}\ S = \emptyset\ \mathsf{then}\ v\ \mathsf{else}\ f(x, \rhd_{set}\ f\ v\ S'), \text{where } S = S' \cup_{set} \{x\} \\
\rhd_{bag}\ f\ v\ B &= \mathsf{if}\ B = \emptyset\ \mathsf{then}\ v\ \mathsf{else}\ f(x, \rhd_{bag}\ f\ v\ B'), \text{where } B = B' \uplus \{(x, 1)\} \\
\rhd_{rnk}\ f\ v\ R &= \mathsf{if}\ R = \emptyset\ \mathsf{then}\ v\ \mathsf{else}\ f(x, \rhd_{rnk}\ f\ v\ R'), \text{where } R = \lceil x \rceil \cup_{rnk} R' \\
\rhd_{seq}\ f\ v\ Q &= \mathsf{if}\ Q = \emptyset\ \mathsf{then}\ v\ \mathsf{else}\ f(x, \rhd_{seq}\ f\ v\ Q'), \text{where } Q = [x] \cup_{rnk} Q'
\end{aligned}$$

Examples. Based on the example given in Fig. 1, assume we want to find the names of all employees working for the `"ACME"` company. Then this query could be expressed as follows.

$$(\sigma_{\text{name}=\texttt{"ACME"}}(\otimes\text{ORGANISATION})) \cdot \text{employees} \cdot \text{name}$$

Another example is the following query to retrieve the names of the organisations for which `"Fred Bloggs"` works. Note that we have split it into two steps purely for the sake of legibility.

$$\textit{fred} := \sigma_{\text{name}=\texttt{"Fred Bloggs"}}(\otimes\text{PERSON})$$

$$\sigma_{\textit{fred}\subseteq\text{employees}}(\otimes\text{ORGANISATION}) \cdot \text{name}$$

Apart from the operators presented in this section, our algebra provides further functionality that had to be omitted due to space limitations. A complete overview of our collection algebra can be found in [22].

5 Implementation

Based on the formal definitions given in the previous sections, we have specified an application programming interface (API) and realised a proof-of-concept implementation. The aim of the proposed API is to serve as a standard for uniform access to object databases, rather than as a standard for application development. As a consequence, our API is quite low-level and procedural. Its main concepts are two interface classes that respectively define the methods to manage and query data according to the object data model and algebra. The signatures of the most commonly used methods of the first interface class are outlined in Tab. 2. These methods allow types to be created and instantiated, and their instances to be read, manipulated and deleted.

For example, an object type can be created with the `createObjectType` method by providing its name and a list of attribute types. Attributes may be of base, structured, object or extent types, which are commonly generalised as `Type`. An object is created using `createObject` and dressed with an object type using the `dress` method which takes the object to be dressed and an object type as argument. Given such an object and its type, attribute values may be read and written with the `get/setAttributeValues` methods. Finally, an object may be deleted with the `deleteObject` method. All other types and their instances are managed similarly.

Table 2. Signatures of API methods

createObjectType(Transaction, String, Type[]): ObjectType
createStructuredType(Transaction, String, Type[]): StructuredType
createExtentType(Transaction, String, BulkType, Type): ExtentType
getType(String): Type
createObject(Transaction): Identifier
dressObject(Transaction, Identifier, ObjectType)
stripObject(Transaction, Identifier, ObjectType)
deleteObject(Transaction, Identifier)
getAttributeValues(Transaction, Identifier, ObjectType): Object[]
setAttributeValues(Transaction, Identifier, ObjectType, Object[])
createExtent(Transaction, ExtentType): ExtentValue
deleteExtent(Transaction, ExtentValue)

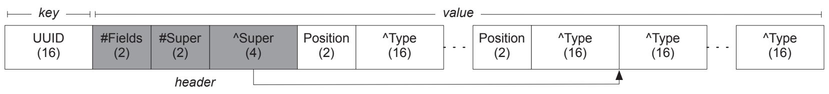
The interface of the algebra is based on the iterator model [23] and thus follows a language-integrated rather than a declarative approach. The signatures of the algebra operators defined in the previous section are shown in Tab. 3. Additionally, our interface provides a `scan` method that, given an `ExtentValue`, returns an iterator. Thus, the `scan` method interfaces between the collection representation of the object data model and the one used in the algebra. The signatures of the remaining operator methods closely correspond to the formal definitions of Sect. 4 and therefore require no further explanation. Note that all of these methods take one or more iterators as input and return one iterator as output. Therefore, operators can be arbitrarily nested to form complex queries.

As a proof-of-concept, we show how the API was implemented using Berkeley DB Java Edition which is a light-weight key-value database providing direct access to its data structures. Due to the nature of our interface, we wanted to avoid the complexity of interacting with a relational or object database system. While this might sound counter-intuitive, it is motivated by the fact that we do not propose an interface for application development and, therefore, do not believe it should be implemented "on top" of an existing database interface. Rather, vendors should offer the proposed interface as an alternative that supports use-cases such as benchmarking and data exchange.

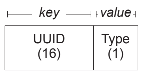
Table 3. Signatures of algebra operators

scan(ExtentValue): Iterator
map(Iterator, Function): Iterator
reduce(Iterator, Function, Object): Iterator
selection(Iterator, Predicate): Iterator
navigate(Iterator, String): Iterator
union(Iterator, Iterator): Iterator
intersection(Iterator, Iterator): Iterator
difference(Iterator, Iterator): Iterator

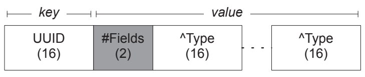
In order to store information about the different types, we use separate databases[3] for base, structured, object and extent types. These four databases constitute the metadata over the persistent data and their record layouts are shown in Fig. 2. As every type is identified by a unique name, we map these names to `UUID` values which are used as database keys. For object types shown in Fig. 2(a), we store a header (grey fields) containing the field and supertype count as well as the offset for the supertypes within the record. We then have a sequence of `(Position, ^Type)` pairs describing the type's attributes. `Position` is used for schema evolution, while `^Type` is a type reference represented as the `UUID` of the attribute type. A sequence of `UUID` values referring to a type's supertypes forms the end of such records. The numbers in parenthesis show the sizes of each record part in bytes. Figure 2(b) shows how base types are described by a `Type` which encodes the basic type from $\mathbb{T}_{base}$. A record describing a structured type is shown in Fig. 2(c). It consists of a header containing the field count and a sequence of `^Type` containing the `UUID` values of the field types. As shown in Fig. 2(d), extent types are stored as an encoded bulk type such as set, bag, sequence or ranking, and the `UUID` of the type describing the extent members.



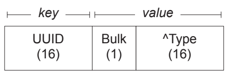
(a) Object type unit



(b) Base type unit



(c) Structured type unit



(d) Extent type unit

Fig. 2. Record layouts in the type metadata databases

In addition to the metadata, a user partition contains the objects, their information units and the extent values. For each object type, we create a separate database containing all of its instances. The entries of such databases start with the instance object's identifier encoded as a `UUID` key, followed by a value part as shown in Fig. 3. The value part contains the information unit's attribute values. Internally, we divide an entry's value part into a fixed-size and a variable-size part. For variable-size attributes such as strings, we store their length and a pointer to the beginning of the variable-size part following the fixed-size part (light grey). This record layout enables the execution of some schema evolution operations without having to re-write all instances of the type under change.

The dress types database shown in Fig. 4 is used to keep track of all information units that belong to an object. In this database, we map the object's `UUID`

[3] In Berkeley DB, the term *database* refers to what would be called a *relation* or *table* in the relational world.

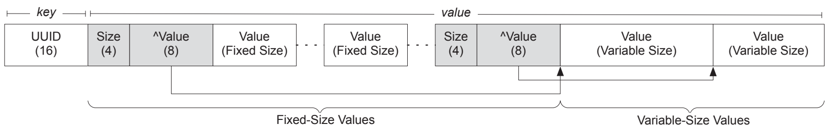


Fig. 3. Record layout for object information units

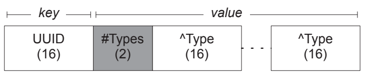


Fig. 4. Record layout for object dress types

to a sequence of UUID values referring to all types an object has been dressed with. This database duplicates information that could be found by accessing all type extents, however, we use it as an index to accelerate look-up operations.

Each extent is stored in its own database. Depending on the bulk type, additional index structures such as Berkeley DB's secondary databases are employed for fast access to extent members. The members are UUID values in the case of objects and extents or the actual values in the case of base and structured values.

6 Discussion

We now discuss and position our work with respect to the related approaches that were introduced in Sect. 2. The object data model that we presented in this paper can be classified as an evolution of the ODMG 3.0 data model. The ODMG modelling primitives of objects and literals correspond to object and structured types in our model. The distinction of whether information is modelled as an identifiable object or an inlined value is present in most object databases. For example, the Versant Object Database (VOD) introduced the notion of first-class and second-class objects, while Objectivity/DB uses the concept of embedded objects to support this feature. As a consequence, we believe that any new object database standard should also include these capabilities. Finally, we note that the collection types defined in our model are slightly different from the ones offered by the ODMG model. Nevertheless, we share the conviction that different collection types and their associated operations are an essential part of an object data model.

The approach that is currently proposed as the next-generation object databases standard takes an altogether different stance in this respect. Instead of defining the characteristics of a standard object data model, their data model decomposes objects into triples that are used to represent all information. While this model is very flexible and easily formalised, it is too general and lacks specificity for the domain of object databases. Our model acknowledges the importance of a formal specification as the foundation of consistent semantics, however we position it differently in terms of the trade-off between flexibility and specificity. Based on object-slicing, our approach supports different models of inheritance and instantiation. At the same time, its type model and collection algebra are truly object-oriented.

When defining a standard, there are different objectives that can be taken into consideration. For example, the ODMG 3.0 standard has been defined to provide better support for unified application development and portability. The goal of the interface proposed in this paper is different as it was designed to facilitate standardised evaluation of object databases in terms of benchmarking or as a format for data exchange. Consequently, our application programming interface does not provide transparent persistence that is nowadays the standard for object database application development. Nevertheless, we believe that the adoption of our proposal is likely as many vendors already offer lower-level interfaces to their databases, e.g. Versant's JVI Fundamental Binding [24].

7 Conclusion

We have presented an object data model that uses object-slicing to support different styles of inheritance and instantiation. We have defined the model formally and used this specification as the basis for a collection algebra that provides query facilities in the context of our object data model. Finally, we have proposed an interface that supports both uniform access and querying of object data that is represented according to the proposed model. As a proof-of-concept, the interface that is intended for benchmarking and data exchange has been implemented using Berkeley DB Java Edition.

As future work, we plan to experiment with different object-slicing strategies. In this paper, we have assumed a one-to-one correspondence between object classes and information units. However, if an object database does not provide multiple inheritance nor multiple instantiation, this assumption might be unreasonable and lead to increased complexity. To experiment with this, we plan to implement our interface based on different existing object databases. At the same time, this will help to demonstrate its value for benchmarking and data exchange.

References

1. Atkinson, M.P., Bancilhon, F., DeWitt, D.J., Dittrich, K.R., Maier, D., Zdonik, S.B.: The Object-Oriented Database System Manifesto. In: Building an Object-Oriented Database System: The Story of O_2, pp. 3–20. Morgan Kaufmann, San Francisco (1992)
2. Dearle, A., Kirby, G.N.C., Morrison, R.: Orthogonal Persistence Revisited. In: Proc. Intl. Conf. on Object Databases (ICOODB), pp. 1–23 (2009)
3. Greene, R.: OODBMS Architectures: An Examination of Implementations. Technical report, Versant Corp. (2006)
4. Cattell, R.G.G., Skeen, J.: Object Operations Benchmark. ACM Trans. Database Syst. 17(1), 1–31 (1992)

5. Carey, M.J., DeWitt, D.J., Naughton, J.F.: The OO7 Benchmark. In: Proc. Intl. Conf. on Management of Data (SIGMOD), pp. 12–21 (1993)
6. Cattell, R.G.G., Barry, D.K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F. (eds.): The Object Data Standard: ODMG 3.0. Morgan Kaufmann, San Francisco (2000)
7. Card, M.: Next-Generation Object Database Standardization. Technical report, Object Management Group (OMG) (2007)
8. Adamus, R., Habela, P., Kaczmarski, K., Lentner, M., Stencel, K., Subieta, K.: Stack-Based Architecture and Stack-Based Query Language. In: Proc. Intl. Conf. on Object Databases (ICOODB), pp. 77–95 (2008)
9. Frost, R.A.: Binary-Relational Storage Structures. Comput. J. 25(3), 358–367 (1982)
10. Papakonstantinou, Y., Garcia-Molina, H., Widom, J.: Object Exchange Across Heterogeneous Information Sources. In: Proc. Intl. Conf. on Data Engineering (ICDE), pp. 251–260 (1995)
11. Versant Corp.: Versant Object Database Fundamentals Manual, Release 8.0 (2009)
12. Box, D., Hejlsberg, A.: The LINQ Project. Technical report, Microsoft Corporation (2005)
13. Meijer, E., Beckman, B., Bierman, G.: LINQ: Reconciling Object, Relations and XML in the.NET Framework. In: Proc. Intl. Conf. on Management of Data (SIGMOD), pp. 706–706 (2006)
14. Paterson, J., Edlich, S., Hörning, H., Hörning, R.: The Definitive Guide to db4o. Apress (2006)
15. Martin, J., Odell, J.J.: Object-Oriented Analysis and Design. Prentice-Hall, Inc., Englewood Cliffs (1992)
16. Parsons, J., Wand, Y.: Emancipating Instances from the Tyranny of Classes in Information Modeling. ACM Trans. Database Syst. 25(2), 228–268 (2000)
17. Ra, Y.G., Kuno, H.A., Rundensteiner, E.A.: A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views. Technical Report CSE-TR-215-94, University of Michigan (1994)
18. Kuno, H.A., Ra, Y.G., Rudensteiner, E.A.: The Object-Slicing Technique: A Flexible Object Representation and its Evaluation. Technical Report CSE-TR-241-95, University of Michigan (1995)
19. Fishman, D.H., Beech, D., Cate, H.P., Chow, E.C., Connors, T., Davis, J.W., Derrett, N., Hoch, C.G., Kent, W., Lyngbæk, P., Mahbod, B., Neimat, M.A., Ryan, T.A., Shan, M.C.: Iris: An Object-Oriented Database Management System. ACM Trans. Office Info. Syst. 5(1), 48–69 (1987)
20. Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A Vision for Management of Complex Models. SIGMOD Rec. 29(4), 55–63 (2000)
21. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications Co. (2006)
22. Würgler, A.P.: OMS Development Framework: Rapid Prototyping for Object-Oriented Databases. PhD thesis, ETH Zurich (2000)
23. Graefe, G.: Volcano–An Extensible and Parallel Query Evaluation System. IEEE Trans. on Knowl. and Data Eng. 6(1), 120–135 (1994)
24. Versant Corp.: Java Versant Interface Usage Manual, Release 8.0 (2009)