iBLOB: Complex Object Management in Databases Through Intelligent Binary Large Objects

Tao Chen, Arif Khan, Markus Schneider[⋆] and Ganesh Viswanathan

Department of Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611, USA
{tachen, ahkhan, mschneid, gv1}@cise.ufl.edu

Abstract. New emerging applications including genomic, multimedia, and geospatial technologies have necessitated the handling of complex *application objects* that are highly structured, large, and of variable length. Currently, such objects are handled using filesystem formats like HDF and NetCDF as well as the XML and BLOB data types in databases. However, some of these approaches are very application specific and do not provide proper levels of data abstraction for the users. Others do not support random updates or cannot manage large volumes of structured data and provide their associated operations. In this paper, we propose a novel two-step solution to manage and query application objects within databases. First, we present a generalized conceptual framework to capture and validate the structure of application objects by means of a *type structure specification*. Second, we introduce a novel data type called *Intelligent Binary Large Object* (*iBLOB*) that leverages the traditional BLOB type in databases, preserves the structure of application objects, and provides smart query and update capabilities. The iBLOB framework generates a type structure specific application programming interface (API) that allows applications to easily access the components of complex application objects. This greatly simplifies the ease with which new type systems can be implemented inside traditional DBMS.

1 Introduction

Many fields in computer science are increasingly confronted with the problem of handling large, variable-length, highly structured, complex *application objects* and enabling their storage, retrieval, and update by application programs in a user-friendly, efficient, and high-level manner. Examples of such objects include biological sequence data, spatial data, spatiotemporal data, multimedia data, and image data, just to name a few. Traditional database management systems (DBMS) are well suited to store and manage large, unstructured alphanumeric data. However, storing and manipulating large, structured application objects at the low byte level as well as providing operations on them are hardly supported. *Binary large objects* (*BLOBs*) are the only means to store such objects. However, BLOBs represent them as low-level, binary strings and do not preserve their structure. As a result, this database solution turns out to be unsatisfactory.

[⋆] This work was partially supported by the National Aeronautics and Space Administration (NASA) under the grant number NASA-AIST-08-0081.

Hence, scientists have designed special file formats like *NetCDF* (*network Common Data Form*) and *HDF5* (*Hierarchical Data Format*) to store such objects in files. Unfortunately, without the support of a DBMS, standard features like a query language, concurrency control, transaction management, security, and recovery are unavailable (*data management problem*). A widely accepted approach to handling complex data in databases is to model and implement them as values of *abstract data types* (*ADT*) in a type system, or *algebra*, which is then embedded into an extensible DBMS and its query language. This enables their use as attribute data types in a database schema without disclosing the implementation details of their complex internal structure to the user and DBMS. At the type system level, extensible DBMS enable the specification of new ADTs like spatial, image, and XML data types. However, these ADTs have DBMS specific implementations and are not universally deployable (*generality problem*). On the other hand, BLOBs are not well suited for structured object management. They have originally been designed for storing unstructured data as byte sequences and offer a low-level interface for simple read/write access to byte ranges. Thus BLOBs do not understand the semantics of the internal structure of the application objects stored in them and therefore do not include methods to access internal components of them (*abstraction problem*). This makes the access to a component of an application object rather expensive since the entire object needs to be loaded into main memory to understand its structural semantics and get access to the component of interest. Further, BLOBs typically allow data to be appended, truncated, and modified through the overwriting of bytes. However, general data insertions and deletions are not supported unless the user explicitly shifts data (*update problem*).

In this paper, we present a novel, generic model for complex object management that focuses on providing the required functionality to address the data management, generality, abstraction, and update problems. We first propose a generalized method, named *type structure specification*, for representing and interpreting the structure of application objects. This specification provides an interface for the ADT implementer to describe the structure of complex objects at the conceptual level. Based on this specification, we employ a generalized framework, called *intelligent binary large objects* (*iBLOBs*), for the efficient and high-level storage, retrieval, and update of hierarchically structured complex objects in databases. iBLOBs store complex objects by utilizing the unstructured storage capabilities of DBMS and provide component-wise access to them. In this sense, they serve as a communication bridge between the high-level abstract type system and the low-level binary storage. This framework is based on two orthogonal concepts called *structured index* and *sequence index*. A *structured index* facilitates the preservation of the structural composition of application objects in unstructured BLOB storage. A *sequence index* is a mechanism that permits full support of *random updates* in a BLOB environment.

Section 2 describes relevant research related to the iBLOB concept. In Section 3, we describe the applications that involve large structured application objects, the existing approaches to handling them, and our approach to dealing with structured objects in a database context. We introduce the concept of type structure specification and the iBLOB framework in Sections 4 and 5. Finally, in Section 6, we draw some conclusions and discuss future work.

2 Related Work

The need for extensibility in databases, in general, and for new data types in databases [11], in particular, has been the topic of extensive research from the late eighties. In this section, we review work related to the storage and management of structured large application objects. The four main approaches can be subdivided into *specialized file formats*, new *DBMS prototypes*, *traditional relational DBMS*, and *object-oriented extensibility mechanisms in DBMS*.

The *specialized file formats* can be further categorized into text formats and binary formats [8]. Text formats organize data as a stream of Unicode characters whereas binary formats store numbers in "native" formats. XML [4] is a universal standard text data format primarily meant for data exchange. A critical issue with all text data formats is that they make the data structure visible and that one cannot randomly access specific subcomponent data in the middle of the file. The whole XML file has to be loaded into the main memory to extract the data portion of interest. Moreover, the methods used to define the legal structure for a XML document such as Document Type Definition (DTD) and XML Schema Definition (XSD) have several shortcomings. DTD lacks support for datatypes and inheritance, while XSD is really over-verbose and unintuitive when defining complex hierarchical objects. On the other hand, binary data formats like NetCDF [8, 9] and HDF [1, 8] support random access of subcomponent data. But updating an existing structure is not explicitly supported in both formats. Further, since HDF stores a large amount of internal structural specifications, the size of a HDF file is considerably larger than a flat storage format. Further, these file formats do not benefit from DBMS properties such as transactions, concurrency control, and recovery.

The second approach to storing large objects is the development of new *DBMS prototypes* as standalone data management solutions. These include systems such as *BSSS* [7], *DASDBS*, [10], *EOS* [3], *Exodus* [5], *Genesis* [2], and *Starburst* [6]. These systems operate on variable-length, uninterpreted byte sequences and offer low-level byte range operations for insertion, deletion, and modification. However, these systems do not manage structural information of large application objects and are hence unable to provide random access to object components.

The third approach taken to store large objects is the *use of tables and BLOBs* in traditional object-relational database management systems. Any hierarchical structure within an object can be incorporated in tables using a separate attribute column that cross-references tuples with their primary keys. Some database such as Oracle even support hierarchical SQL queries on such tables. However, the drawback of this method is that the querying becomes unintuitive and has to be supported by complex procedural language functions inside the database. Further, these queries are slow because of the need of multiple joins between tables. Binary Large OBjects (BLOBs) provide another means to store large objects in databases. However, this is a mechanism for storing unstructured, binary data. Hence, the entire BLOB has to be loaded into main memory each time for processing purposes.

The fourth approach to storing large objects is the use of *object-oriented extension mechanisms* in databases. Most popular DBMS support the CREATE TYPE construct to create user-defined data types. However, the type constructors provided (like array constructors) do not allow to create large *and* variable-length application objects.

3 Problems with Handling Structured Application Objects in Database Systems and Our Solution

Application objects like DNA structures, 3D buildings, and spatial regions are complex, highly structured, and of variable representation length. The desired operations on the application objects usually involve high complexity, long execution time and large memory. For example, *region* objects are complex application objects that are frequently used in GIS applications. As shown in Figure 1, a region object consists of components called *faces*, and *faces* are enclosed by *cycles*. Each *cycle* is a closed sequence of connected *segments*. Applications that deal with regions might be interested in numeric operations that compute the *area*, the *perimeter* and the *number of faces* of a region. They might also be interested in geometric operations that compute the *intersection*, *union*, and *difference* of two regions. Many more operations on regions are relevant to applications that work with maps and images. In any case, the implementation of an operation requires easy access to components of structured objects (e.g., segments, cycles, and faces of a region) that uses less memory and runs in less time.

Since database systems provide built-in advanced features like the SQL query language, transaction control, and security, handling complex objects in a database context is an expedient strategy. Most approaches are built upon two important architectures that enable database support for applications involving complex application objects.

Early approaches apply a *layered architecture* as shown in Figure 2a, in which a *middleware* that handles complex application objects is clearly separated from the *application front-end* that provides services and analysis methods to its users. In this architecture, only the underlying primitive data are physically stored in traditional RDBMS tables. The knowledge about the structure of complex objects is maintained in the middleware. It is the responsibility of the middleware to load the primitive data from the underlying database tables, to reconstruct complex objects from the primitive data, and to provide operations on complex objects. The underlying DBMS in the layered architecture does not understand the semantics of the complex data stored. In this sense, the database is of limited value, and the burden is on the application developer to implement a middleware for handling complex objects. This complicates and slows down the application development process.

A largely accepted approach is to model and implement complex data as *abstract data types* (*ADTs*) in a type system, or *algebra*, which is then embeded into an exten-

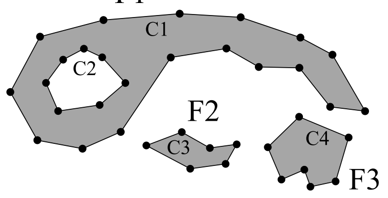


Fig. 1. A region object as an example of a complex, structured application object. It contains the faces F1, F2, and F3, which consist of the cycles C1 and C2 for F1, C3 for F2, and C4 for F3.

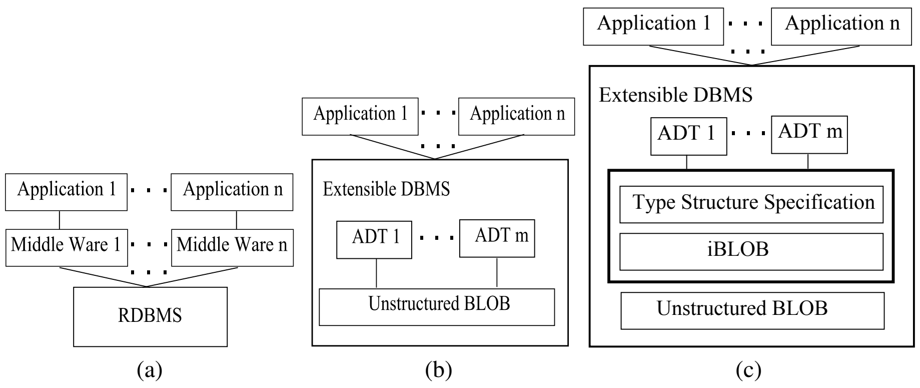


Fig. 2. The layered architecture (a) and the integrated architecture (b) and our solution (c).

sible DBMS and its query language. This approach employs an *integrated architecture* (Figure 2(b)), where the applications directly interact with the extended database system, and use the ADTs as attribute data types in a database schema. Some commercial database vendors like Oracle and Postgres have included some ADTs like spatial data types as built-in data types in their database products. Extensible DBMS provides users the interfaces for implementing their own ADT so that all types of applications can be supported. Since the only available data structure for storing complex objects with variable length is BLOB, the implementations of ADTs for complex objects are generally based on BLOBs. The implementation of an abstract data type involves three tasks, the design of binary representation, the implementation of component retrieval and update, and the implementation of high level operations and predicates.

The integrated architecture has obvious advantages. It transfers the burden of handling complex objects from the application developer to databases. Once abstract data types are designed and integrated into a database context, applications that deal with complex objects become standard database applications, which require no special treatment. This simplifies and speeds up the development process for complex applications. However, the drawback of this approach is that ADTs for structured application objects rely on the unstructured BLOB type, which provides only byte level operations that complicate, or even foil, the implementation of component retrieval and update. Byte manipulation is a redundant and tedious task for *type system implementers* who want to implement a high-level type system because they want to focus on the design of the data types and the algorithms for the high-level operations and predicates.

In this paper, we propose a new concept that extends the integrated architecture approach, provides the type system implementers with a high level access to complex objects, and is capable of handling any structured application objects. In our concept, we apply the integrated architecture approach and extend it with a generalized framework (Figure 2c) that consists of two components, the *type structure specification* (Section 4) and the *intelligent BLOB* concept (Section 5). The type structure specification consists of algebraic expressions that are used by type system implementers to specify the internal hierarchy of the abstract data type. It is later used as the meta data for the intelligent

BLOB to identify the semantic meaning of each structure component. Further, as part of the type structure specification we provide a set of high-level functions as interfaces for type system implementers to create, access, or manipulate data at the component level. To support the corresponding interfaces, we propose a generic storage method called *intelligent BLOB* (*iBLOB*), which is a binary array whose implementation is based on the BLOB type and which maintains hierarchical information. It is “intelligent” because, unlike BLOBs, it understands the structure of the object stored and supports fast access, insertion and update to components at any level in the object hierarchy.

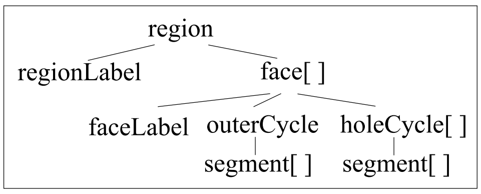
The type structure specification in the framework provides an abstract view of the application object which hides the implementation details of the underlying data structure. The underlying intelligent BLOBs ensure a generic storage solution for any kinds of structured application objects, and enable the implementation of the high-level interfaces provided by the type structure specification. Therefore, the type structure specification and the concept of intelligent BLOBs together enable an easy implementation for abstract data types. type system implementers can be released from the task of interpreting the logical semantics of binary unstructured data, and the component level access is natively supported by the underlying iBLOB.

4 Representing and Interpreting Structured Application Objects with Type Structure Specifications

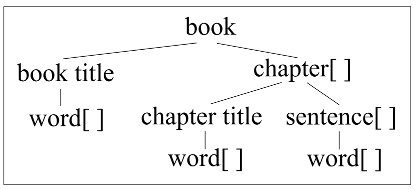
The structures of different application objects can vary. Examples are the structure of a region (Figure 1) and the structure of a book. We aim at developing a generic platform that accommodates all kinds of hierarchical structures. Thus, the first step is to explore and extract the common properties of all structured objects. Unsurprisingly, the hierarchy of a structured object can always be represented as a tree. Figure 3a shows the tree structure of a *region* object. In the figure, *face*[], *holeCycle*[], and *segment*[] represent a list of faces, a list of hole cycles and a list of segments respectively. In the tree representation, the root node represents the structured object itself, and each child node represents a component named *sub-object*. A sub-object can further have a structure, which is represented in a sub-tree rooted with that sub-object node. For example, a region object in Figure 3a consists of a label component and a list of face components. Each face in the face list is also a structured object that contains a face label, an outer cycle, and a list of hole cycles, where both the outer cycle and the hole cycles are formed by segments lists. Similarly, the structure of a book can also be represented as a tree (Figure 3b).

Further, we observe that two types of sub-objects can be distinguished called *structured objects* and *base objects*. Structured objects consist of sub-objects, and base objects are the smallest units that have no further inner structure. In a tree representation, each leaf node is a base object while internal nodes represent structured objects.

A tree representation is a useful tool to describe hierarchical information at a conceptual level. However, to give a more precise description and to make it understandable to computers, a formal specification would be more appropriate. Therefore, we propose a generic *type structure specification* as an alternative of the tree representation for describing the hierarchical structure of application objects.



(a)



(b)

Fig. 3. The hierarchical structure of a *region* object and the hierarchical structure of a *book* object.

We first introduce the concept of *structure expressions*. Structure expressions define the hierarchy of a structured object. A structure expression is composed of *structure tags* (*TAGs*) and *structure tag lists* (*TAGLISTs*). A structure tag (TAG) provides the *declaration* for a single component of a structured object, whereas a structure tag list (TAGLIST) provides the declaration for a list of components that have the same structure. The declaration of a TAG, named *tag declaration*, is $\langle NAME : TYPE \rangle$, where $NAME$ is the identifier of the tag and the value of $TYPE$ is either SO, which is a flag that indicates a structured object, or BO, which is a flag that indicates a base object. An example of a *structured object tag* is $\langle region : SO \rangle$, and $\langle segment : BO \rangle$ is an example of a *base object tag*. We first define a set of terminals that will be used in structure expressions as constants. Then, we show the syntax of structure expressions.

$$Terminal\ Set\ S = \ \{:=,\ \langle,\ \rangle,\ |,\ [,\ ]\ ,\ SO,\ BO,\ :\}$$

$$\begin{array}{ll} Expression & ::= TAG := \langle TAG \mid TAGLIST \rangle^{+}; \\ TAGLIST & ::= TAG\,[\,] \\ TAG & ::= \langle NAME : TYPE \rangle \\ TYPE & ::= \langle SO \mid BO \rangle \\ NAME & ::= IDENTIFIER \end{array}$$

In the region example, we can define the structure of a region object with the following expression: $\langle region : SO \rangle := \langle regionLabel : BO \rangle \langle face : SO \rangle [\,]$. In the expression, the left side of := gives the tag declaration of a region object and the right side of := gives the tag declarations of its components, in this case, the region label and the face list. Thus, we say the region object is *defined* by this structure expression.

With structure expressions, the type system implementer can recursively define the structure of structured sub-objects until no structured sub-objects are left undefined. A list of structure expressions then forms a specification. We call a specification that consists of structure expressions and is organized following some rules a *type structure specification* (*TSS*) for an abstract data type. Three rules are designed to ensure the *correctness* and *completeness* of a type structure specification when writing structure expressions: (1) the first structure expression in a TSS must be the expression that defines the abstract data type itself (*correctness*); (2) every structured object in a TSS has to be defined with one and only one structure expression (*completeness* and *uniqueness*); (3) none of the base objects in a TSS is defined (*correctness*). By following these rules, the type system implementer can write one type structure specification for each abstract data type. Further, it is not difficult to observe that the conversion between a tree representation and a type structure specification is simple. The root node in a tree maps to the first structure expression in the TSS. Since all internal nodes are structured sub-objects and leaf nodes are base sub-objects, each internal node has exactly one corresponding structure expression in the TSS, and leaf nodes require no structure expressions. The type structure specification of the abstract data type *region* corresponding to the tree structure in Figure 3a is as follows:

$$\begin{array}{ll}
\langle region : SO\rangle & := \langle regionLabel : BO\rangle\langle face : SO\rangle[\,];\\
\langle face : SO\rangle & := \langle faceLabel : BO\rangle\langle outerCycle : SO\rangle\langle holeCycle : SO\rangle[\,];\\
\langle outerCycle : SO\rangle & := \langle segment : BO\rangle[\,];\\
\langle holeCycle : SO\rangle & := \langle segment : BO\rangle[\,];
\end{array}$$

The next step after specifying the structure is to create and store the application object into the database. The TSS provides a workable interface for the type system implementer to create, access and navigate through the object. This higher-level interface is the abstraction of the iBLOB interface. This abstraction along with the specification, frees the type system implementer from understanding the underlying data type iBLOB that is used for finally representing the application object in the database. Navigating through the structure of the object is done by specifying a path from the root to the node by a string using the *dot-notation*. For example, to point to the first segment of the outer cycle of the third face of a region object can be specified by the string *region.face*[3].*outerCycle.segment*[1]. A component number (e.g., *first* segment, *third* face) is determined by the temporal order when a component was inserted. An important point to mention is that the structural validity of a path (e.g., whether an outer cycle is a subcomponent of a face) can be verified by parsing the TSS. However, the existence of a third face can only be detected during runtime. The set of operators which are defined by the interface are given below:

$$\begin{array}{ll}
create & :\rightarrow SO\\
get & : path \rightarrow BO[]\\
set & : path \rightarrow bool\\
set & : path \times char* \rightarrow bool\\
baseObjectCount & : path \rightarrow int\\
subObjectCount & : path \rightarrow int
\end{array}$$

An application object can be created by the operator *create*() which generates an empty *application object*. The operator $get(p)$ returns all base objects at leaf nodes under the node specified by any valid path p. Since no data types are defined for the structured objects in intermediate nodes, these objects are not accessible, and paths to them are undefined. Hence, paths to intermediate nodes are interpreted differently in the sense that the operator $get(p)$ recursively identifies and returns all base objects under p. The operator $set(p)$ creates an intermediate component. The operator $set(p,s)$ inserts a base object given as a character string s at the location specified by the path p. The last two operators $baseObjectCount(p)$ and $subObjectCount(p)$ return the number of base objects and the number of sub-objects under a node specified by the path p. As an example, for a region object with one face that contains an outer cycle with three segments, the corresponding code for creating the region object is given below:

```
region r = create(); r.set(region.regionLabel,"MyRegion");
r.set(region.face[1]); r.set(region.face[1].faceLabel,"Face1");
r.set(region.face[1].outerCycle);
r.set(region.face[1].outerCycle.segment[1],seg1);
r.set(region.face[1].outerCycle.segment[2],seg2);
r.set(region.face[1].outerCycle.segment[2],seg3);
```

The first line of the code shows how the type system implementer can create a region object based on the specified type structure specification. The second line creates the first face and the third line its outer cycle as intermediate components. The following three lines store the three segments *seg*1, *seg*2, *seg*3 as components of the outer cycle.

5 Intelligent Binary Large Objects (iBLOBs)

In this section, we present the conceptual framework for a new database data type called *iBLOB* for *Intelligent Binary Large Objects*. This type enhances the functionality of traditional binary large objects (BLOBs) in database systems. Our concept also helps to solve the generality, abstraction and update problems (described in Section 1) that are exhibited by current approaches (see Section 2) to manage large application objects. BLOBs serve currently as the only means to store large objects in DBMS. However, they do not preserve the structure of application objects and do not provide access, update and query functionality for the sub-components of large objects. *iBLOBs* help to smartly extend traditional BLOBs by preserving the object structure internally and providing application-friendly access interfaces to the object components. All this is achieved while maintaining low level access to data and extending existing database systems using object-oriented constructs and *abstract data types* (*ADTs*).

The iBLOB framework consists of two main sections called the *structure index* and the *sequence index* (Figure 4). The first section contains the *structure index* which helps us represent the object structure as well as the base data. The second section contains the *sequence index* that dictates the sequential organization of object fragments and preserves it under updates. Since the underlying storage structure of an iBLOB is provided through a BLOB, which is available in most DBMSs, the iBLOB data type can be registered as a user-defined data type and be used in SQL.

5.1 iBLOB Structure Index: Preserving Structure in Unstructured Storage

A structure index is a mechanism that allows an arbitrary hierarchical structure to be represented and stored in an unstructured storage medium. It consists of two components for, first, the representation of the structure of the data and, second, the actual data

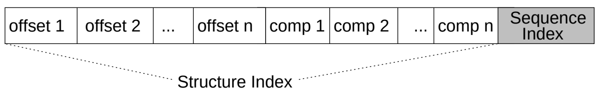


Fig. 4. Illustration of an iBLOB object consisting of a structure index and a sequence index.

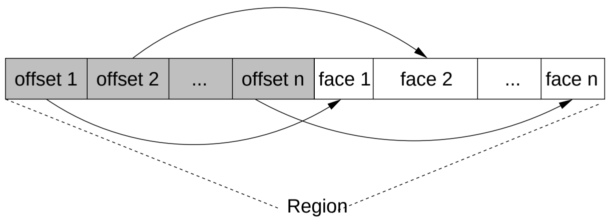


Fig. 5. A structured object consisting of *n* sub-objects and *n* internal offsets

themselves. The structural component is used as a reference to access the data's structural hierarchy. The mechanism is not intended to enforce constraints on the data within it; thus, it has no knowledge of the semantics of the data upon which it is imposed. This concept considers hierarchically structured objects as consisting of a number of variable-length sub-objects where each sub-object can either be a *structured object* or a *base object*. Within each structured object, its sub-objects reside in sequentially numbered slots. The leaves of the structure hierarchy contain base objects.

To illustrate the concept of a structure index, we show an example how to store a spatial region object with a specific structure in a database. A region data type may be described by a hierarchical structure as shown in Figure 3a. Consider a region made up of several faces. If we needed to access the 50th face of a region object using a traditional BLOB storage mechanism, one would have to load and sequentially traverse the entire BLOB until the desired face would be found. Further, since the face objects can be of variable length, the location of the 50th face cannot be easily computed without extra support built in to the BLOB. In order to avoid an undesirable sequential traversal of the BLOB, we introduce the notion of *offsets* to describe structure. Each hierarchical level of a structure in a structure index stored in a BLOB is made up of two components (corresponding to the two components of the general structure index described above). The first component contains offsets that represent the location of specific sub-objects. The second component represents the sub-objects themselves. We define offsets to have a fixed size; thus, the location of the *i*th face can be directly determined by first calculating the location of the *i*th offset and then reading the offset to find the location of the face. Figure 5 shows a structured object with internal offsets.

The recursive nature of hierarchical structures allows us to generalize the above description. Each sub-object can itself have a structure like the region described above. Objects at the same level are not required to have the same structure; thus, at any given level it is possible to find both structured sub-objects and base objects (raw data). For example, we can extend the structure of a region object so that it is made up of a collection of faces each of which contains an outer cycle and zero or more hole cycles, which in turn are made up of a collection of segments. Segments can be implemented as a pair of (x,y)-coordinate values. This example is illustrated in terms of structured and base objects in Figure 6 where the top level object represents a region with an information part, a label, and one of its face sub-objects.

In general, a specific structure index implementation must be defined with respect to the underlying unstructured storage medium. Because we have to use BLOBs as

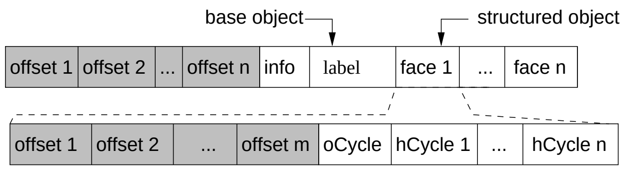


Fig. 6. A structured object consisting of a base object and structured sub-objects

the only alternative in a database context, we are forced to embed both the structure index and the data into a BLOB. Thus, using an offset structure embedded within the data itself is an appropriate solution. However, this may not be ideal in all cases. For instance, one could implement a structure index for data stored in flat files. In this case, the structure index and the data could be represented in seperate files. In general, the structure index concept must be adapted to the capabilities of the user's desired storage medium for implementation.

5.2 iBLOB Sequence Index: Tracking Data Order for Updates

Different DBMSs provide different implementations of the BLOB type with varied functionalities. However, most advanced BLOB implementations support three operations at the byte level, namely, random read and append (write bytes at end of BLOB), truncate (delete bytes at end) and overwrite (replace bytes with another block of bytes of the same or smaller length).

Structured large objects require the ability to update sub-objects within a structure. Specifically, they require *random updates* which include insertion, deletion and the ability to replace data with new data of arbitrary size. Examples are the replacement of a segment by several segments in a cycle of a region object, or adding a new face. Given a large region object, updating it entirely for each change in a face, cycle or segment becomes very costly when stored in BLOBs (update problem). Thus, it is desirable to update only the part of the structure that needs updating. For this purpose, we present a novel *sequence index* concept that is based on the random read and data append operations supported by BLOBs Extra capabilities provided by higher level BLOBs are a further improvement and serve for optimization purposes. The sequence index concept is based on the idea of physically storing new data at the end of a BLOB and providing an index that preserves the logically correct order of data.

Consequently, data will have internal fragmentation and will be physically stored out-of-order, as illustrated in Figure 7. In this figure, the data blocks (with start and end

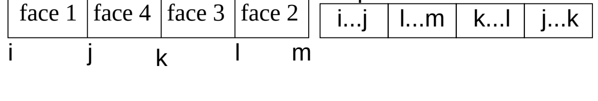


Fig. 7. An out-of-order set of data blocks and their corresponding sequence index

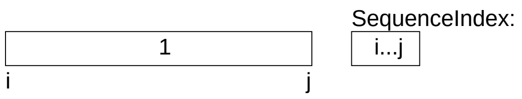


Fig. 8. The initial in-order and defragmented data and sequence index.

byte addresses represented by letters under each boundary) representing faces should be read in the order $1,2,3,4$, even though physically they are stored out-of-order in the BLOB (we will study the possible reasons shortly). By using an ordered list of physical byte address ranges, the sequence index specifies the order in which the data should be read for sequential access. The sequence index from Figure 7 indicates that the block $[i \dots j]$ must be read first, followed by the block $[l \dots m]$, etc.

Based on the general description of the sequence index given above, we now show how to apply it as a solution to the update problem. Assume that the data for a given structured object is initially stored sequentially in a BLOB, as shown in Figure 8. Suppose further that the user then makes an insertion at position k in the middle of the object. Instead of shifting data after position k within the BLOB to make room for the new data, we append it to the BLOB as block $[j \dots l]$, as shown in Figure 9. By modifying the sequence index to reflect the insertion, we are able to locate the new data at its logical position in the object.

Figure 10 illustrates the behavior of the sequence index when a block is intended to be deleted from the structured object. Even though there is no new data to append to the BLOB, the sequence index must be updated to reflect the new logical sequence. Because the BLOB does not actually allow for the deletion of data, the sequence index is modified in order to prevent access to the deleted block $[m \dots n]$ of data. This can result in internal fragmentation of data in the iBLOB which can be managed using a special *resequence* operation shown later in the iBLOB interface.

Finally, Figure 11 illustrates the case of an update where the values of a block of data $[o \dots p]$ as a portion of block $[j \dots l]$ are replaced with values from a new block $[l \dots q]$. For this kind of update, it is possible for the new set of values to generate a block size different from that of the original block being replaced.

iBLOBs enhance BLOBs by providing support for truncate and overwrite operations at the higher *component level* of an application object's structure. The *truncate* operation in BLOB (delete bytes at end) is enhanced in iBLOB with a *remove* function which can perform deletion of components at any location (beginning, middle or at the end of structure) as shown in Figure 10. The *overwrite* operation in BLOB (replace

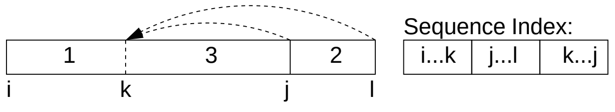


Fig. 9. A sequence index after inserting block $[j \dots l]$ at position k.

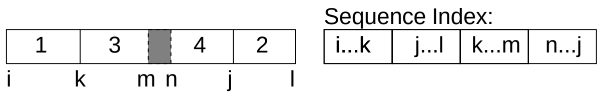


Fig. 10. A sequence index after deleting block $[m\ldots n]$.

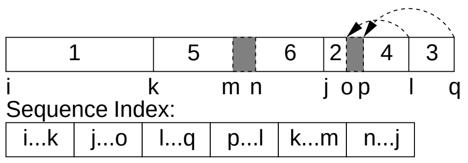


Fig. 11. A sequence index after replacing block $[o\ldots p]$ by block $[l\ldots q]$.

byte array with another of same length) is enhanced in iBLOB with a combination of *remove* and *insert* functions and sequence index adjustments, to perform the overwrite of components with other components of different sizes as shown in Figure 10.

5.3 The iBLOB Interface

In this section, we present a generic interface for constructing, retrieving and manipulating iBLOBs. Within this *iBLOB interface*, we assume the existence of the following data types: the primitive type *Int* for representing integers, *Storage* as a storage structure handle type (i.e., blob handle, file descriptor, etc.), *Locator* as a reference type for an iBLOB or any of its sub-objects, *Stream* as an output channel for reading byte blocks of arbitrary size from an iBLOB object or any of its sub-objects, and *data* as a representation of a base object. Figure 12 lists the operations offered by the interface. We use the term *l-referenced object* to indicate the object that is referred to by a given locator *l*. The following descriptions for these operations are organized by their functionality:

- **Construction and Duplication**: An iBLOB object can be constructed in three different ways. The first constructor $create()$ (1) creates an empty iBLOB object. The second constructor $create(sh)$ (2) constructs an iBLOB object from a specific storage structure handle *sh* such as a BLOB object handle or a file descriptor. The third constructor $create(s)$ (3) is a copy constructor and builds a new iBLOB object from an existing iBLOB object *s*. Similarly, an iBLOB object s_2 can also be copied into another iBLOB object s_1 by using the $copy(s_1, s_2)$ operator (4).
- **Internal Reference**: In order to provide access to an internal sub-object of an iBLOB object, we need a way to obtain the reference of such a sub-object. The sub-object referencing process must start from the topmost hierarchical level of the iBLOB object *s* whose locator *l* is provided by the operator $locateiBLOB(s)$ (5). From this locator *l*, a next level sub-object can be referenced by its slot *i* in the operator $locate(s, l, i)$ (6).

$$create : \rightarrow iBLOB \tag{1}$$
$$create : Storage \rightarrow iBLOB \tag{2}$$
$$create : iBLOB \rightarrow iBLOB \tag{3}$$
$$copy : iBLOB \times iBLOB \rightarrow iBLOB \tag{4}$$
$$locateiBLOB : iBLOB \rightarrow Locator \tag{5}$$
$$locate : iBLOB \times Locator \times Int \rightarrow Locator \tag{6}$$
$$getStream : iBLOB \times Locator \rightarrow Stream \tag{7}$$
$$insert : iBLOB \times data \times Int \times Locator \times Int \rightarrow iBLOB \tag{8}$$
$$insert : iBLOB \times iBLOB \times Locator \times Int \rightarrow iBLOB \tag{9}$$
$$remove : iBLOB \times Locator \times Int \rightarrow iBLOB \tag{10}$$
$$append : iBLOB \times data \times Int \times Locator \rightarrow iBLOB \tag{11}$$
$$append : iBLOB \times iBLOB \times Locator \rightarrow iBLOB \tag{12}$$
$$length : iBLOB \times Locator \rightarrow Int \tag{13}$$
$$count : iBLOB \times Locator \rightarrow Int \tag{14}$$
$$resequence : iBLOB \rightarrow iBLOB \tag{15}$$

Fig. 12. The standardized iBLOB interface

- **Read and Write**: Since iBLOBs support large objects which may not fit into main memory, we provide a stream based mechanism through the operator $getStream(s,l)$ (7) to consecutively read arbitrary size data from any l-referenced object. The stream obtained from this operator behaves similarly to a common file output stream. Other than reading data, the interface allows insertion of either a base object d of specified size z through the operator $insert(s,d,z,l,i)$ (8) or an entire iBLOB object s_1 through the operator $insert(s,s_1,l,i)$ (9) into any l-referenced object at a specified slot i. A base object d such as in the operator $append(s,d,z,l)$ (11) or a iBLOB object s_1 such as in operator $append(s,s_1,l)$ (12) can be appended to an l-referenced object. This is effectively the same as inserting the input as the last sub-object of the referenced object. The operator $remove(s,l,i)$ (10) removes the sub-object at slot i from the parent component with Locator l.
- **Properties and Maintenance**: The actual size of an l-referenced object is obtained by using the operator $length(s,l)$ (13) while the number of sub-objects of the object is provided by the operator $count(s,l)$ (14). Finally, the operator $resequence(s)$ (15) reorganizes and defragments the iBLOB object s collapsing its sequence index such that it contains a single range. This operation effectively synchronizes the physical and logical representations of the iBLOB object and minimizes the storage space.

To test the functionality we have implemented the iBLOB data type in Oracle, Informix and PostgreSQL using object oriented extensions and programming API of the DBMS. Due to space constraints, we have omitted the iBLOB implementation details in this paper. Each operator in the TSS interface can be implemented using the corresponding iBLOB interface operator. For e.g., to implement $get(region.face[1].outerCycle.segment[1])$, we first use $locateiBLOB$ (5) to get a Locator to the iBLOB, then use *locate*() (5) repeatedly to move across levels and navigate to the required component (i.e., first segment), and finally, *getStream*() (7) to retrieve the first segment of the outerCycle in fifth face. Other TSS interface functions like *set*, *baseObjectCount* and *subObjectCount* can also be implemented in a similar manner.

6 Conclusions

In this paper, we provide a novel solution to store and manage complex application objects (i.e., variable length, structured, hierarchical data) by introducing a new mechanism for handling structured objects inside DBMSs. This includes two major concepts. First, we present a *type structure specification* (*TSS*) that helps to describe the structure of complex application objects. Then we introduce a special SQL data type called *Intelligent Binary Large Object* or *iBLOB* that enables the database to handle structured objects. The combination of type structure specification and iBLOBs provides the necessary tools to easily implement type systems in databases. However, the focus of this paper is to extend database functionality to natively support complex objects. As future work, we plan to optimize iBLOBs for performance.

References

1. HDF-Hierarchical Data Format. *http://www.hdfgroup.org/*.
2. D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. Genesis: an Extensible Database Management System. *IEEE Trans. on Software Engineering*, 14:1711–1730, 1988.
3. A. Biliris. The Performance of Three Database Storage Structures for Managing Large Objects. *ACM SIGMOD Int. Conf. on Management of Data*, pp. 276–285, 1992.
4. T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. *W3C recommendation*, 6, 2000.
5. M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A Data Model and Query Language for Exodus. *ACM SIGMOD Record*, 17:413 – 423, 1988.
6. L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita. Starburst Mid-flight: As the Dust Clears. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 2:143–160, 1990.
7. B. Hwang, I. Jung, and S. Moon. Efficient storage management for large dynamic objects. *EUROMICRO 94. System Architecture and Integration 20th EUROMICRO Conference.*, pp. 37–44, Sep 1994.
8. R.E. McGrath. XML and Scientific File Formats. *The Geological Society of America*, 2003.
9. R.K. Rew, B. UCAR, and EJ Hartnett. Merging netCDF and HDF5. *20th Int. Conf. on Interactive Information and Processing Systems*, 2004.
10. H.-J. Schek, H.-B. Paul, M. H. Scholl, and G. Weikum. The DASDBS Project: Objectives, Experiences, and Future Prospects. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 2(1):25–43, 1990.
11. M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. *Int. Conf. on Data Engineering Conference (ICDE)*, pp. 262–269, 1986.