Structured Large Objects in Databases

Mark McKenney, Alejandro Pauly, Reasey Praing & Markus Schneider *
University of Florida
Department of Computer Science & Engineering
Gainesville, FL 32611, USA
{mm7,apauly,rpraing,mschneid}@cise.ufl.edu

March 30, 2006

Abstract

Emerging database applications like genomic, biological, and multimedia databases process application objects that are complex, highly structured, and of variable representation length. At the DBMS implementation level, large objects (LOBs), originally designed for unstructured data and equipped with a simple, low-level interface, are so far the only available method for persistently storing such application objects. However, the LOB interface is inappropriate for performing operations that take into account structural properties of application objects. This paper proposes a new concept and SQL data type named SLOB for implementing and managing *structured large objects* in databases. SLOBs are designed as an implementation basis for complex type systems, or *algebras*, integrated into databases. Instead of granting access to byte sequences like LOBs, SLOBs leverage a new conceptual framework designed to provide random access to the components of a complex application object without knowing their meaning. A strength of this approach is the handling of *random updates* of complex objects. A book algebra implementation based on a SLOB implementation demonstrates and a performance analysis evaluates the concepts.

1 Introduction

Emerging database applications like biological, genomic, biomedical, moving object, image, spatial, and multimedia databases permanently confront database technology with new, advanced data management requirements at all levels of a DBMS architecture. In contrast to traditional database applications with their simply structured, alphanumerical data, an essential feature of these applications is that their objects are complex, highly structured, and of variable representation length. A largely accepted approach to handling complex data is to model and implement them as *abstract data types* 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 to the DBMS.

Extensibility in databases has been an active and widely studied research topic in the database community. Well known DBMS prototypes like Exodus, Genesis, Starburst, Postgres, and many more witness the intensive efforts and significant advances in research to make DBMSs more extensible and customizable at all levels of their architecture. Commercial DBMSs (CDBMSs) have adopted and adapted a number of these extensibility mechanisms into their own architectures to be able to support advanced applications.

At the type system level, DBMSs enable the specification of new types and operations. Fixed-length types can be defined by the type constructors and built-in types of the DBMS. Large, structured values of

*This work was partially supported by the National Science Foundation under grant number NSF-CAREER-IIS-0347574.

variable-length types, as they appear in emerging applications (e.g., spatial objects like regions and volumes, genomic objects like genes and DNA, multimedia objects like videos and images), can currently only be persistently stored as *large objects* (LOBs). But for LOBs they reveal a number of shortcomings for this purpose. First, they have originally been designed for storing unstructured data. They provide and process byte sequences and offer a low-level interface for simple read/write access to byte ranges. Unfortunately, they do not include methods to access internal components of complex application objects stored in them (*abstraction problem*). An algebra implementor would highly benefit from this feature since she can then access an application object and all its components at a higher abstraction level and not only at the low byte level. Second, LOBs 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*). For instance, a replacement of a two minute segment in the middle of a video by a three minute segment requires a shift of all data after the old segment to make room for the new segment. Third, LOB implementations have different interfaces with different functionalities in different DBMSs (*generality problem*). The non-uniformity of the LOB interfaces makes it difficult for third-party developers to create database applications that are DBMS independent; each DBMS platform requires an own, separate implementation of a database extension. Fourth, a general problem is that many useful concepts of DBMS research prototypes, which are tailor-made for a certain problem area and utilize a specialized infrastructure, cannot be easily transferred to CDBMSs. Therefore, these achievements and the prototypes offering them do not find an adequate appreciation since the users' question of whether they can use them in their own CDBMS has to be negated (*acceptance problem*).

The first main goal of this paper is to propose a new conceptual framework that enables current, extensible DBMSs to adequately support algebra (type system) implementations for emerging applications. This framework is based on two orthogonal concepts called *structure index* and *sequence index*. A *structure index* facilitates the preservation of the structural composition of application objects in unstructured LOB storage. A *sequence index* is a mechanism that permits full support of *random updates* in a LOB environment. Both concepts only require minimal functionality from a LOB implementation.

Based on this framework, the second main goal is to propose a new SQL data type named *SLOB* for managing and implementing *structured large objects* in databases. This new database data type is designed as an abstract data type and solves the four aforementioned problems of unstructured LOBs. SLOBs address the abstraction problem by providing a component-based (but not byte-based) access to structured application objects by utilizing the unstructured storage capabilities of DBMSs. This component-based view is internally provided by the structure index. The user obtains efficient access to all components of application objects; only those of interest have to be retrieved into main memory for querying and modification. In this sense, SLOBs represent a communication bridge between the high-level algebra and the low-level DBMS extensibility mechanism. The important update problem is alleviated by the sequence index concept which allows random updates of structured data without the overhead of shifting data. The SLOB data type addresses the generality problem by providing the user with a single, general, and DBMS independent interface to implement arbitrary, complex algebras requiring structured, persistent, and efficient data access. Each extensible DBMS implementing SLOBs will be equipped with the *same* SLOB interface. This enables the seamless integration and deployment of the *same* algebra in *different* DBMSs. Furthermore, each SLOB implementation can be tuned according to the particular characteristics of each DBMS. Finally, our SLOB implementation makes a contribution to the acceptance problem. Since users employ CDBMSs in practice, we propagate new implementation concepts on the basis of CDBMS extensibility mechanisms. IF possible, extensions should be designed CDBMS independent so that they can be integrated into many of them. Our SLOB implementation satisfies these criteria. The benefit is that the impact of new research developments can be expected to be much higher, and the hope is that CDBMS vendors can be much better convinced to include new innovations into their systems since they have been validated as prototypes in their own systems.

Section 2 describes relevant research related to the SLOB concept. Section 3 presents the conceptual framework (structure index and sequence index) as it is deployed in the SLOB implementation. In Section 4 we present the concept of structured large objects and the SLOB data type with its interface and implementation. In Section 5 we detail an implementation of a sample book algebra and examine the performance of the SLOB concept. Finally, in Section 6 we discuss conclusions and future work.

2 Related Work

Our particular motivation for dealing with the topic of this paper is our interest in designing and implementing algebras (type systems) for new emerging applications in a database context. Examples are the *ROSE Algebra* for spatial databases [14], the *Moving Objects Algebra* for spatio-temporal databases [13], and the *Genomics Algebra* for genomic databases [16]. Publications related to the SLOB concept can be roughly classified into three categories. We consider extensibility mechanisms for embedding new type systems into DBMSs, large object management as the currently only mechanism to store large application objects in a DBMS, and DBMS standardization efforts.

The need for *extensibility in databases* [7] in general and the need for new (*abstract*) *data types* in databases [20, 24, 26] in particular has been an extensive research topic in the eighties and early nineties. Several extensible DBMS prototypes like Exodus [9], Genesis [4], Starburst [15], DASDBS [22], and Postgres [21] have implemented extensibility mechanisms at all levels of their architecture. Proposals for designing and incorporating data types in extensible DBMSs include *externally defined types* (*EDTs*) [27], *enhanced abstract data types* (*E-ADTs*) [23], and *user-defined data types* (*UDTs*) [19]. All these type systems form many-sorted algebras. In CDBMSs user-defined algebras are specified and integrated as cartridges (Oracle) [2], extenders (DB2) [10], or data blades (Informix) [1].

For the implementation of algebras and especially large complex application objects in a DBMS, several storage systems have been proposed in the literature [6]. For example, Exodus offers the concept of dynamic *large storage objects* which are represented on disk as B^+-tree like indexes on the byte position within the object plus a collection of leaf data blocks [8]. BSSS [17] aims at improving the performance of the Exodus approach. Starburst provides *long fields* [18] implemented on the basis of the buddy system. Genesis represents objects in Jupiter files [3]. EOS stores large objects in a sequence of variable-sized segments and employs a tree structure for indexing byte positions within an application object [5]. Postgres stores large objects as a collection of *variable-length segments* (*v-segments*) [25]. CDBMSs like Oracle, DB2, Informix, and Sybase offer uninterpreted byte sequences in *large objects* (*LOBs*) that are intended for storing unstructured large application objects. The work in [11] deals with the embedding of LOBs in aggregates like tuples and determines thresholds when a LOB should be inlined within the aggregate or swapped out to a separate object. All these storage concepts have in common that they operate on variable-length, uninterpreted byte sequences, offer low-level byte range operations for insertion, deletion, and modifications, and have been designed for *unstructured* large application objects. That is, they do not maintain information about the structure of application objects in order to enable the algebra built on top to become aware of it. Therefore, they are inappropriate for representing and managing large *structured* application objects. In this paper, we regard LOBs as given and available and design and implement our proposed SLOB data type on top of them.

The work in [12] comes nearest to our concept. The authors present a method to store structured objects by nesting LOB-like structures within each other. This allows a component view of structured objects to be stored in a LOB much like the SLOB concept. In essence, their approach allows an object represented in a LOB to contain handles to other LOBs representing sub-objects. However, this approach does not address the update problem. In order to insert a sub-object into an object, the LOB handles in the parent object must be shifted to make room for the new handle. Furthermore, this concept requires features that are not available in CDBMSs; thus, it does not address the acceptance problem.

A number of standardization efforts in the database field has been performed in the past. They especially focus on establishing an SQL standard and in unifying DBMS interfaces by standardized APIs such as JDBC and ODBC. However, there are no accepted standards for integrating new algebras into DBMSs, no unique interfaces for LOB manipulation, and no support for the implementation of structured large application objects, as the rather different concepts of cartridges, data blades, and extenders show. Our SLOB concept is a step towards such a standardization. It solves the generality problem and permits code reusability.

3 Structured Large Objects: The Conceptual Framework

The concepts we present in this section detail a framework of ideas and mechanisms that are necessary to address the acceptance, abstraction, and update problems described in Section 1. The concepts presented are general and can in principle be applied to any unstructured storage medium. However, in order to address the acceptance problem, we define them with respect to the general capabilities offered by CDBMS extension mechanisms and the underlying LOB data type. To relieve the abstraction problem, support for representing and accessing structural information in unstructured storage is provided by the *structure index* described in Section 3.1. The update problem is resolved by enhancing the basic LOB data types through the use of a *sequence index* shown in Section 3.2.

3.1 Structure Index for 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: the first is the representation of the data's structure, the second is the actual data. 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* (an object with no substructure). Within each structured object, its sub-objects reside in sequentially numbered slots. The leaves of the structure hierarchy contain base objects. For the remainder of this paper, we will discuss the structure index as it pertains to CDBMS LOBs in order to address the acceptance problem more directly.

For the sake of simplicity, we use an example to describe the structure index. Assume we have a structured object representing a book, which, for this example, contains two levels of structure. A book is made up of word objects (the first structural level) which in turn are described by simple strings of characters (the base object). Consider an editor who wants to randomly access the 3000^{th} word of a book. Given a book that is stored in a simple unstructured LOB, the DBMS must sequentially traverse the book until the desired word is found. Because words are of variable length, the location of the 3000^{th} word cannot be easily computed without extra support built in to the LOB. In order to avoid an undesirable sequential traversal of the LOB, we introduce the notion of *offsets* to describe structure. Each hierarchical level of structure in a structure index stored in a LOB 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} word 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 word. Figure 1 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 book 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 raw data. For example, we can extend the structure of the book so that it is made up of chapters each of which

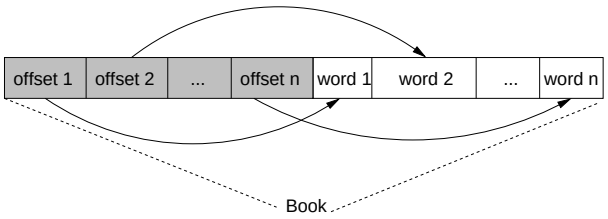

Figure 1: A structured object consisting of *n* sub-objects and *n* internal offsets

contains authors and sentences, which in turn are made up of words. A list of authors can be implemented as a string of characters. This example is illustrated in Figure 2 where the top level object shown represents a chapter with the details of its authors and one of its sentence sub-objects.

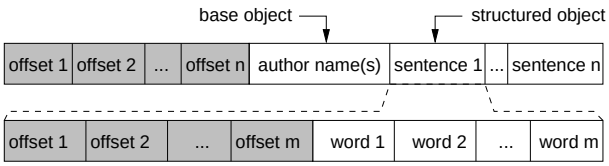

Figure 2: A structured object consisting of a base object and structured sub-objects

In general, a specific structure index implementation must be defined with respect to the underlying unstructured storage medium. Because we are using a structural index in a LOB, we are forced to embed the structure index into a LOB along with the data. 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.

3.2 Sequence Index for Updating Data

Current implementations of LOB types in databases assume that their primary use is for handling unstructured data. As a result, early LOB interfaces were rather restrictive. Recent DBMSs have bettered their LOB handling capabilities to provide more functionality. We classify the LOB handling capabilities that a DBMS can offer in three levels of increasing functionality. At the lowest and most primitive level (*level 1 LOB*), the DBMS has the ability to read from any location within the LOB (*random read*) but write only at the end of the LOB (*append*). At the middle level (*level 2 LOB*), the capability of truncating the LOB data is provided in addition to the capabilities provided by the lower level. Finally, at the top level (*level 3 LOB*), in addition to the capabilities provided in the lower two levels, the DBMS provides the ability to overwrite data.

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. For example, consider a structured book object that is composed of chapters, each chapter is composed of paragraphs, each paragraph is composed of sentences and each sentence is composed of words. During the editing process, it might be necessary to update a single word or sentence, or delete a sentence, or introduce a new one. Given a large book, updating the whole book for each word change would be very costly due to the update problem; thus, it is desirable to update only the part of the structure that needs to be updated.

In order to provide a framework capable of resolving the update problem for any currently available CDBMS, we present a novel sequence index concept based on level 1 LOBs (random read and data append). Extra capabilities provided by higher level LOBs are considered a luxury and are useful for optimization purposes.

The sequence index concept is based on the idea of physically storing new data at the end of a LOB and providing a sequence index that is able to logically maintain the correct order of the data. Consequently, data will have internal fragmentation and will be physically stored out-of-order, as illustrated in Figure 3.

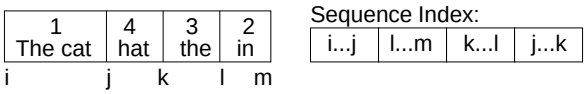

Figure 3: An out-of-order set of data blocks and their corresponding sequence index

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

Figure 4: The initial in-order and defragmented data and sequence index.

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 LOB, as shown in Figure 4. Suppose 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 LOB to make room for the new data, we append it to the LOB as block $[j \ldots l]$. By modifying the sequence index to reflect the insertion, we are able to locate the new data at its logical position in the object.

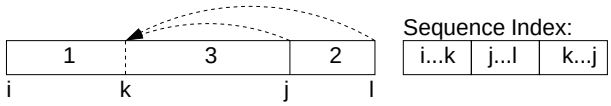

Figure 5: A sequence index after inserting block $[j \ldots l]$ at position *k*.

Figure 6 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 LOB, the sequence index must be updated to reflect the new sequence that must be read. Because the LOB does not actually allow for deletion of data, the sequence index is modified to eliminate access to the deleted block $[m \ldots n]$ of data. Thus, level 1 LOBs (and in level 2 and 3 LOBs), deletions result in internal fragmentation.

Finally, Figure 7 illustrates the case of an update where the values of a block of data $[o \ldots p]$ as a portion of block $[j \ldots l]$ are replaced with values from a new block $[l \ldots q]$. In this type 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.

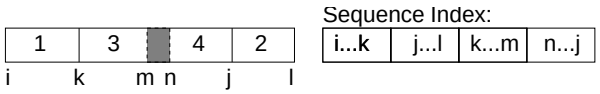

Figure 6: A sequence index after deleting block $[m \ldots n]$.

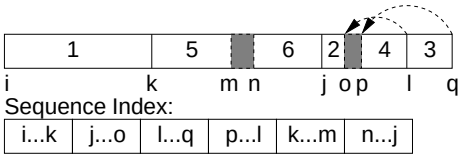

Figure 7: A sequence index after replacing block $[o \ldots p]$ by block $[l \ldots q]$.

Having examined the behavior of the sequence index given a level 1 LOB implementation, we now consider optimizations that may be taken depending on more advanced features offered by the underlying LOB mechanism:

- **Truncate**: Level 2 and 3 LOB mechanisms allow the deletion of blocks at the end of a LOB object (truncate). Such a feature is advantageous for deletion or replacement of blocks that are located at the end of the LOB.
- **Overwrite**: Given the ability to overwrite data at random locations within the LOB (as in level 3 LOBs), it is possible to perform same sized block replacements without any modification in the sequence index. If the size of the new block is different from the size of the old block, then sequence index adjustments must be made.

4 Structured Large Objects: The Implementation

At the implementation level, we propose a new SQL data type named SLOB that integrates the concepts of structure index and sequence index. This data type is intended as the implementation basis for constructing algebras with complex, structured application data types. A SLOB object is general enough to store any structured object irrespective of its depth and the complexity of its hierarchical structure. If the SLOB data type is made available in CDBMSs, new emerging database applications can store their specialized objects in SLOBs and at the same time take advantage of the advanced features these highly developed systems provide (i.e., concurrency and transaction control, multi-user access, recoverability, robustness, etc.). In Section 4.1, we provide a general definition of the SLOB data type and the standardized SLOB interface. A prototype level 1 LOB implementation is discussed in Section 4.2.

4.1 SLOB: The Data Type

In general, a SLOB data type allows the construction of algebras or application specific data types and provides support for the creation, retrieval, and manipulation of structured objects through a generic and DBMS independent interface. Recall that the structure index mechanism is used to impose and maintain the object's structure while the sequence index mechanism is used to preserve the integrity of the object after an arbitrary manipulation to the object's data. Both mechanisms are orthogonal, i.e., they independently

provide specific functionality over unstructured data. By integrating both concepts into a SLOB data type, we can alleviate the acceptance, abstraction, generality and update problems introduced in Section 1. As mentioned previously, the acceptance problem is addressed by the use of the DBMS LOBs as a storage medium. The abstraction and update problems are solved through the use of the structure index and sequence index, respectively. Finally, the generality problem is covered in this section through the definition of a standardized SLOB interface.

The SLOB data type consists of two parts. The first part contains the structure index, and the second part contains the sequence index, as shown in Figure 8.

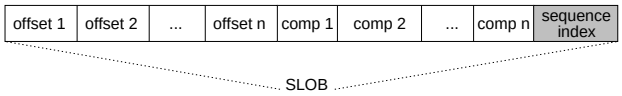

Figure 8: A representation of a SLOB object

We now present the standardized interface of the SLOB data type. This interface consists of operations for constructing, retrieving, and manipulating a SLOB object. From a DBMS standpoint, the SLOB interface is available internally as a registered SQL data type or externally as part of a software library. Within this 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 a SLOB or any of its sub-objects, *Stream* as an output channel for reading byte blocks of arbitrary size from a SLOB object or any of its sub-objects, and *data* as a representation of a base object. Figure 9 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**: A SLOB object can be constructed in three different ways. The first constructor *create*() (1) creates an empty SLOB object. The second constructor *create*(*sh*) (2) constructs a SLOB object from a specific storage structure handle *sh* such as a LOB object handle or a file descriptor. The third constructor *create*(s) (3) is a copy constructor and builds a new SLOB object from an existing SLOB object *s*. Similarly, a SLOB object s_2 can also be copied into another SLOB 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 a SLOB 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 SLOB object *s* whose Locator *l* is provided by the operator *locateSlob*(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).

- **Read and Write**: Since SLOBs 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 SLOB 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 SLOB 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*.

$$create: \rightarrow SLOB \quad (1)$$
$$create: Storage \rightarrow SLOB \quad (2)$$
$$create: SLOB \rightarrow SLOB \quad (3)$$
$$copy: SLOB \times SLOB \rightarrow SLOB \quad (4)$$
$$locateSlob: SLOB \rightarrow Locator \quad (5)$$
$$locate: SLOB \times Locator \times Int \rightarrow Locator \quad (6)$$
$$getStream: SLOB \times Locator \rightarrow Stream \quad (7)$$
$$insert: SLOB \times data \times Int \times Locator \times Int \rightarrow SLOB \quad (8)$$
$$insert: SLOB \times SLOB \times Locator \times Int \rightarrow SLOB \quad (9)$$
$$remove: SLOB \times Locator \times Int \rightarrow SLOB \quad (10)$$
$$append: SLOB \times data \times Int \times Locator \rightarrow SLOB \quad (11)$$
$$append: SLOB \times SLOB \times Locator \rightarrow SLOB \quad (12)$$
$$length: SLOB \times Locator \rightarrow Int \quad (13)$$
$$count: SLOB \times Locator \rightarrow Int \quad (14)$$
$$resequence: SLOB \rightarrow SLOB \quad (15)$$

Figure 9: The standardized SLOB interface

- **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 SLOB object *s* collapsing its sequence index such that it contains a single range. This operation effectively synchronizes the physical and logical representations of the SLOB object and minimizes the storage space required.

4.2 Implementing SLOBs

In this section, we describe our SLOB implementation based on a level 1 LOB. As mentioned in Section 3.2, by demonstrating the SLOB implementation at the lowest capability level, we ensure that all SLOB functionality can be provided under any CDBMS. The implementation of SLOBs at higher levels of LOB capabilities allows for increased performance and reduced storage overhead.

The SLOB interface is implemented in C++ to take advantage of its capabilities as a system programming language as well as the fact that most DBMSs provide C++ accessability libraries. Our SLOB implementation is provided for the Oracle DBMS version 10g. Oracle provides level 3 LOBs; however, we ignore the Oracle LOB operations available at the top two levels. In Section 4.2.1, we describe our sequence index implementation. Then, in Section 4.2.2, we present our structure index implementation and the details of its integration into the SLOB data type implementation.

4.2.1 Sequence Index Implementation

The purpose of the sequence index is to provide a logically sequential view of data physically stored in arbitrary order in the LOB. To allow this, we represent the sequence index as an array of address ranges. For our implementation, the start of each range is represented by the address of the first byte in the range, and the end of each range is represented by the address of the last byte in the range. The sequence index is designed as a main memory array stored on disk. In order to facilitate efficient access and manipulation of the sequence index and the underlying SLOB, the sequence index is loaded into main memory upon access to a particular SLOB.

Locating data in the SLOB is done by using the sequence index to map logical byte positions to physical addresses. For example, consider Figure 3. Assume we have read block 1 and are now at position j. To retrieve the next logical block (in this case block 2), we must find the physical address of the logical byte position immediately following block 1 (position $j + (1\ \textit{logical byte position})$). Note that in this particular example, this physical address is l. The sequence index provides this mapping through the location operations described in the interface below.

When an update operation is performed on a SLOB object, the sequence index is modified to represent a logical shift in the data sequence. To delete a block of data, we remove any references within the sequence index to physical addresses in the block. Note that removing data from the sequence index does not necessarily imply a physical deletion of the referenced data. To reflect the insertion of new data in the sequence index, references to the new block of data must be inserted into the sequence index. Either a new range is inserted into the sequence index, or an existing range is extended to include the new block. Replacing a data block with a new data block amounts to a deletion followed by an insertion. Figure 10 presents the interface offered by the sequence index. Let *SeqIdx* be the type of all sequence indexes.

$$\textit{create}: \ \rightarrow \textit{SeqIdx} \tag{16}$$
$$\textit{create}: \ \textit{range}[\,] \rightarrow \textit{SeqIdx} \tag{17}$$
$$\textit{fromBack}: \ \textit{SeqIdx} \times \textit{Int} \rightarrow \textit{Int} \tag{18}$$
$$\textit{fromFront}: \ \textit{SeqIdx} \times \textit{Int} \rightarrow \textit{Int} \tag{19}$$
$$\textit{fromPos}: \ \textit{SeqIdx} \times \textit{Int} \times \textit{Int} \rightarrow \textit{Int} \tag{20}$$
$$\textit{insert}: \ \textit{SeqIdx} \times \textit{Int} \times \textit{Int} \times \textit{Int} \rightarrow \textit{SeqIdx} \tag{21}$$
$$\textit{append}: \ \textit{SeqIdx} \times \textit{Int} \times \textit{Int} \rightarrow \textit{SeqIdx} \tag{22}$$
$$\textit{rmBySize}: \ \textit{SeqIdx} \times \textit{Int} \times \textit{Int} \rightarrow \textit{SeqIdx} \tag{23}$$
$$\textit{rmByEnd}: \ \textit{SeqIdx} \times \textit{Int} \times \textit{Int} \rightarrow \textit{SeqIdx} \tag{24}$$
$$\textit{replace}: \ \textit{SeqIdx} \times \textit{Int} \times \textit{Int} \times \textit{Int} \rightarrow \textit{SeqIdx} \tag{25}$$
$$\textit{byteCount}: \ \textit{SeqIdx} \times \textit{Int} \times \textit{Int} \rightarrow \textit{Int} \tag{26}$$

Figure 10: Sequence index interface

The sequence index operations span the following categories:

- **Construction:** A sequence index can be constructed with $\textit{create}()$ (16) which generates an empty sequence index with no ranges. When constructed with $\textit{create}(d)$ (17), creates a sequence index loaded with the address ranges provided in the array d.
- **Location:** Data is located by traversing the sequence index in three ways. A call to $\textit{fromBack}(si, b)$ (18) is able to find the physical address that is located b (logical) bytes from the end of the byte

sequence given by the sequence index *si*. Similarly, *fromFront*(si, f) (19) finds the address located *f* (logical) bytes from the beginning of the sequence index *si*. Function *fromPos*(si, p) (20) works the same way as function (19) but instead of starting from the beginning, it starts from a given location *p*.

- **Insertion:** Operator *insert*(si, l, p, s) (21) modifies the logical sequence by inserting *s* bytes of data stored at physical location *p* at the position where address *l* is found in the sequence index *si*. The operation *append*(si, p, s) (22) appends *s* bytes of data stored at physical location *p* to the end of the logical sequence.

- **Removal:** Data is logically removed by using the sequence index in one of two ways. The operator *rmBySize*(si, s, l) (23) allows the removal of *l* bytes from start address *s* in sequence index *si*. Notice how this removal is logical and can span several ranges of the sequence index. The operator *rmByEnd*(si, s, e) (24) removes all bytes between the start address *s* and end address *e* in sequence index *si*. Note that such a removal can span several logical address ranges.

- **Replacement:** Operator *replace*(si, l, p, s) (25) allows the replacement of *s* bytes at location *l* with *s* bytes stored at location *p*.

- **Information:** Operator *byteCount*(si, s, e) (26) provides the number of bytes accessible by the sequence index between *s* and *e* where *s* and *e* can span multiple byte ranges.

4.2.2 Structure Index Implementation

The implementation of our structure index works by maintaining, for each object, a set of fixed size offsets to the address where each of its sub-objects ends. We purposely define offsets to point to end positions of sub-objects because the beginning of the first sub-object can be calculated by locating the end of the last offset. On the other hand, the end position of the last sub-object cannot be calculated in general; thus, it must be stored. Each offset value is determined by the physical location where the offset is written, and the physical location where the sub-object it represents ends. So if at location x we write the offset for a sub-object that ends at location y, the value of that offset will be $y - x$. Conceptually, an offset is a *pointer* to the end of an object. For this reason, we can locate the beginning of that sub-object by computing the end position of its (logically) previous sub-object (or in the case of the first sub-object, the end position of the last offset).

4.2.3 SLOB implementation

The SLOB data type implementation unifies the functionality of the sequence index and the structure index. In addition to the data required for the structure index (offsets) and sequence index (address ranges), the SLOB requires each object to include its number of contained sub-objects, which we denote the *count*, and a *logical object boundary*, implemented as a *dummy* byte. The count is located immediately following the dummy byte at the end of an object. Figure 11 illustrates the location of the count and the dummy bytes at different levels of the hierarchy.

A basic implementation requirement for SLOBs is the ability to find and refer to specific sub-objects within a SLOB object. We define the type *Locator* as an object handle type for a SLOB and its sub-objects. An instance of the Locator type (which we denote a *locator*) represents an object but does not contain any of the object's data. In our implementation, a locator is made up of the physical address where an object starts, the physical address where an object ends, and an object's count. If the object is at the lowest level, the number of sub-objects is set to 0.

Recall that offsets refer to physical byte positions instead of logical byte positions. Offsets are implemented in this way because if offsets were to store logical addresses, every data update in the SLOB would

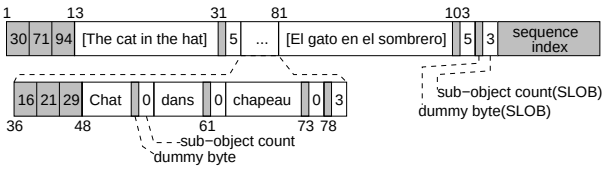

Figure 11: A sample instance of a SLOB. Note that the inner offset structure of the sentences in between brackets is not shown in order to improve clarity. The bytes immediately after these sentences symbolize their dummy bytes and their sub-object counts.

cause all offsets logically forward of the update to become *obsolete* due to the logical shift of data. In the case of physical offsets, this problem only appears if the last sub-object of an object is updated causing the physical end of the object to change which would render that object's offset obsolete. This problem is solved by the existence of the dummy byte marking the logical end of an object, which is pointed to by the object's offset. The dummy byte is only removed if the object it corresponds to is removed. As a consequence, the offset always points to a physical location that exists and can be found through the sequence index.

Based on the previous ideas, the internal reference functions listed in Section 4.1 are implemented. We provide the details of the implementation of each of the SLOB interface functions as follows:

- **Construction**: To create a SLOB based on a LOB handle with *create* (2), we load the sequence index which is located at the end of the LOB. This is possible because we store the size of the sequence index as the last piece of data of the whole LOB.

- **Internal Reference**: To generate a locator for the SLOB in *locateSlob* (5), we first locate its end address by moving q bytes from the end of the SLOB, where q is equal to the sum of the size of the SLOB's count and the size of the dummy byte. This is done with the sequence index function *fromBack* (18). Similarly, its start address is retrieved with a call to the sequence index operation $\mathit{fromFront}(0)$ (19).

  The locator of the sub-object at slot k, given a locator l to its parent object, is generated using the operation *locate* (6). To locate the end of the object we must find its offset which is located $k * s_{off}$ from the start of the parent (where s_{off} represents the number of bytes used by each offset). To find the start of the sub-object when $k = 0$ (i.e., it is the first sub-object) we can simply calculate $\mathit{count} * s_{off}$ where count is the number of sub-objects of the parent. For all other sub-objects, their start can be calculated by finding the offset for the previous sub-object $(k-1)$ and logically moving (from the end of the previous sub-object) q bytes.

- **Write**: First, the data corresponding to the new sub-object is appended to the end of the LOB. These data include the count, the dummy byte and the offset of the new object, in addition to the incremented count of its parent. Once this is done, the new data is logically relocated by modifying the sequence index and then writing the new sequence index into the LOB. Similarly, when a sub-object is removed, the only data to be written into the LOB is the decremented count of the parent object. The removed sub-object is only removed through the sequence index but it still remains physically stored in the LOB.

- **Properties and Maintenance**: Resequencing a SLOB with operation *resequence* (15) entails modifying the physical locations of the objects. As a consequence, the resequencing process must generate new offsets. By recursively resequencing each hierarchical level, the resequence operation is able to rewrite the SLOB with all data in-order and without internal fragmentation.

5 SLOB Performance

In this section, we demonstrate the effectiveness of the SLOB data type by presenting an algebra implementation based on SLOBs. A book algebra implementation is first presented. Then, we consider performance aspects of the SLOB data type.

5.1 A Test Case: Book Algebra

Our book algebra implementation allows a user to store a structured book object in a SLOB object, and therefore, manipulate any level of the book's structural hierarchy by using high level programming constructs. The highest level object in the algebra's hierarchy is a book which contains a sentence representing the book's title, and one or more chapters. Each chapter in the book contains a sentence representing the chapter's title, and one or more paragraphs. Each paragraph contains one or more sentences. Each sentence contains a sequence of one or more word objects and punctuation objects, which are base objects. Punctuation objects are fixed length and contain a single punctuation mark. Words are variable length and contain a sequence of characters. We choose to implement the book algebra in C++ because the SLOB data type implementation is in C++; therefore, words and punctuation are implemented as the C++ *char* type.

Because only base objects contain low level data, the higher level book algebra classes do not actually need to manipulate low level data in memory. Instead, they contain information to indicate where their corresponding sub-objects are located in the SLOB. For instance, a chapter must contain information as to which sentence in the SLOB is its title and which paragraphs belong to it; however, the actual character data representing the title is stored in word and punctuation objects within the sentence object that represents the title. Therefore, higher level objects can be implemented as Locator objects to their corresponding objects in the SLOB. A paragraph, for example, contains a locator to its corresponding paragraph object in the SLOB. The i^{th} sentence belonging to the paragraph can then be retrieved by using the *locate* operation provided by the SLOB interface to create a locator to that sentence. The user can then use the SLOB interface to manipulate that sentence through the locator.

The chapter and book classes are slightly more complex than the other high level classes in the book algebra since they both contain two different types of sub-objects (a book contains a sentence and chapters, and a chapter contains a sentence and paragraphs). We have implemented these classes such that the title (the sentence object) is always the first sub-object of a book and a chapter. Because the structural mechanisms of the SLOB have no knowledge of the semantic meaning of the data it stores, this constraint must be enforced by the book algebra itself. Thus, when a book or chapter object is instantiated, an empty title is inserted as its first sub-object, unless a title already exists.

In general, the base objects of any algebra must be able to manipulate data in memory. Thus, when a word or punctuation object is instantiated, the data corresponding to that object is read from the database into memory. In order to write any changes that have been made to the memory copies of words and punctuation objects, the book algebra provides the operations *writeToSlob* and *readFromSlob* to save the memory version of the object to persistant storage in the SLOB and re-read the data from the SLOB into memory, respectively. These two operations leverage SLOB interface operations such as *insert*, *remove*, and *getStream*.

In addition to the functionality described previously, the interface of each class in the book algebra provides methods to create empty sub-objects, insert new sub-objects, and remove sub-objects from the current object. Based on the SLOB interface, creating the book algebra and using it to store books in a database was a straightforward process. The entire book algebra was built using our SLOB implementation and was written in approximately 1000 lines of C++ code. This is only about 100 lines of code more than the original implementation of the book algebra that was written using C++ vectors and could only handle objects in main memory. Figure 12 shows a sample program that creates a simple book that is stored in a

database. The book consists of a title and two chapters, each of which contain a single paragraph consisting of a single sentence.

```
char cs1[] = "The Cat In The Hat.";
char cs2[] = "In French.";
char cs3[] = "Chat dans chapeau.";
char cs4[] = "In Spanish.";
char cs5[] = "El gato en el sombrero.";

book b(&blobHandleObject);
b.setTitle(cs1, strlen(cs1) + 1);
chapter c1 = b.allocateChapter(0);
c1.setTitle(cs2, strlen(cs2) + 1);
paragraph p1 = c1.allocateParagraph(0);
p1.insertSentence(cs3, strlen(cs3) + 1, 0);
chapter c2 = b.allocateChapter(1);
c2.setTitle(cs4, strlen(cs4) + 1);
paragraph p2 = c2.allocateParagraph(0);
p2.insertSentence(cs5, strlen(cs5)+1, 0);
```

Figure 12: Example code for creating a simple book with the book algebra.

5.2 SLOB Performance Evaluation

The example of Section 5.1 effectively demonstrates the advantages provided by SLOBs with respect to code design and portability of DBMS algebra extensions. The concepts from Sections 3 and 4 detail the advantages that SLOBs provide as a mechanism for solving the problems introduced in Section 1. In this section, we investigate the performance of a SLOB in comparison to the performance of a LOB storing a structured object.

To make a valid comparison, we make the following assumptions about DBMS LOBs. First, in our analysis, we ignore the amount of bytes that must be read in order to locate objects within SLOBs and LOBs. CDBMSs are highly optimized for read operations. Therefore, reading large amounts of data is generally not a problem in comparison to updating data. To remain in compliance with the update problem, we assume the LOB does not include a mechanism to allow random updates; however, we assume that the DBMS in which the LOB resides provides the highest level of LOB handling capabilities (random data overwrite, truncate, append and random read). We do not assume that SLOBs are implemented at a particular level of LOB functionality, rather, we test SLOBs at each LOB functionality level. In summary, we are testing the highest possible functionality LOB against SLOBs based on each level of LOB functionality. Note that these assumptions do not indicate that a particular DBMS be used. Instead, the comparison is valid for all LOB implementations that provide the highest level of LOB handling capabilities.

We identify two critical measures that gauge the performance of SLOBs versus LOBs. The first is the number of bytes written through the life of a structured large object. This measure takes into consideration the number of bytes required to create an initial object, plus any bytes that must be written as a result of updates to the object. This measure quantifies the required work load for performing updates on structured large object. The second measure quantifies the size of bytes actually stored in the SLOB or LOB after a series of updates have been executed. This measure is used to understand the storage overhead introduced when using SLOBs.

Table 1: Formulas used to quantify SLOB measures of performance

Data Type	Bytes Stored	Bytes Written	Symbols
SLOB level 1 SLOB level 2 SLOB level 3	$T(N_T) = (b+x)v + by + S_\alpha(N_T) + (N_B y) + (N_T)(v+w)$ $T(N_T) = (b+x)v + by + S_\beta(N_T) + (N_B y) + (N_T)(v+w)$ $T(N_T) = (b+x)v + by + S_\beta(N_T) + (N_B y) + (N_T)(v+w) - (N_B a)$	$Q(N_T) = (b+x)v + by + S_\alpha(N_T) + (N_B y) + (N_T)(w+v)$	b = num of base objects x = num of nonbase objects y = avg num of bytes per base object v = overhead per object w = overhead for updating an object a = byte reuse rate N_T = total num of updates N_B = num of base object updates N_X = num of non − base object updates S_α, S_β = num of bytes to write sequence index
LOB	$T(N_T) = y(b+N_B)$	$Q(N_T) = Q_B(N_B)$ $Q_B(i) = Q_B(i-1) + y + (Q_B(i-1)/2)$ $Q_B(0) = by$	

The two measures we test are quantified in the formulas shown in Table 1. We begin by summarizing the notation used in these formulas. Each formula calculates a number of bytes that must be written or stored; Q is a function that yields the total sum of bytes written to a LOB or SLOB during N_T updates, and T is a function that yields the total number of bytes stored in a LOB or SLOB after N_T updates. Q_B is a recursive function that calculates the number of bytes that must be written in a LOB to insert a new object. N_B is the total number of updates involving base objects (objects at the lowest levels of the structural hierarchy), and N_X is the total number of updates involving non-base objects. For example, adding an empty chapter to a book requires the insertion of a non-base object, while adding a word to a book requires adding a base object. We calculate these values by introducing a constant d which represents the percentage of the total updates that are base object updates. N_B and N_X are then calculated as follows:

$$N_B = \lfloor dN_T \rfloor$$
$$N_X = N_T - N_B$$

We assume that each LOB and SLOB that we test will already contain an initial object. The value of b represents the original number of base objects in the LOB or SLOB, and x represents the original number of non-base objects. We define y to be the average size of a base object in units of bytes. v represents the number of bytes of overhead required to represent structural information for each object in a SLOB. In our implementation, the overhead for an object consists of the offset for that object (4 bytes), the boundary byte for that object (1 byte), and the integer indicating the number of sub-objects within that object (4 bytes); thus, $v = 9$. We consider the bytes of the original object to be physically in-order; therefore, the size of the original object in a SLOB is determined by calculating the overhead required for all objects plus the number of base object bytes plus the size of a minimal (single range) sequence index: $v(b+x) + by + 8$. The original object size for an object in a LOB is equal to the number of base objects multiplied by the average base object size: by.

In order to calculate the number of bytes that must be written to a SLOB for a given number of updates, we must also consider the overhead of updating an object. For instance, when we add a chapter to a book, in addition to adding the new chapter and its structural overhead, the count of the book object must be modified to indicate that it now has another chapter. We introduce w to indicate this overhead, and for our

experiments, set $w = 4$ (size of count). Thus, we can calculate the number of bytes required to perform N_T updates as the number of updates of base objects multiplied by the average size of a base object, plus the total number of updates multiplied by the sum of the overhead for an object and the overhead for updating an object: $yN_B + N_T(v + w)$. If the SLOB is implemented on a level 3 LOB, then updates replacing one object with a new object can reuse the bytes in which the old object was stored. Thus, the resulting SLOB will have a length of the size of the object minus the amount of bytes that we can be overwritten. To model this, we introduce the byte reuse rate constant a.

Finally, we provide two formulas to calculate the size of the sequence index over a series of updates. The first formula, S_α, calculates the number of sequence index bytes that must be written throughout a series of N_T updates. After each update, a new sequence index is written, and this second formula calculates the sum of all of these sequence indexes. For any given update, the sequence index can grow by 32 bytes in the worst case. This case occurs when a new object is inserted such that its offset and data each break a sequence index range into three ranges, effectively generating four extra ranges in the sequence index. Not every update results in worst case growth; however, we will assume a worst case growth pattern for SLOB updates. A sequence index representing an object with no updates has a size of 8 bytes. The second sequence index, S_β calculates the size of a final sequence index once N_T updates have been completed. The formulas for both are as follows:

$$S_\alpha(N_T) = (32\sum_{i=0}^{N_T} i) + 8(N_T + 1)$$
$$S_\beta(N_T) = N_T 32 + 8$$

Using the above notation and the formulas in Table 1, we calculate the performance behavior of SLOBs and LOBs regarding storing and updating structured large objects. In Figure 13, we first examine the number of bytes that must be written to a SLOB or LOB over the course of a series of updates. Note that the graph does not represent the final amount of bytes stored after the updates have occurred, but the number of bytes written to make all the updates. Recall that after each update, the sequence index is written to the end of the SLOB. Because we are measuring bytes written, the fact that some LOB capability levels allow the overwriting of data has no effect. If 5 bytes are written to a slob, and 3 bytes overwrite previously used space, 5 bytes have still been written. Thus, in Table 1, only a single formula is given for all three SLOB levels for this measure. The formula calculates the size of the original SLOB, then adds the amount of bytes required to write the objects involved in the update, and finally calculates the number of bytes required to write the sequence index to the SLOB after each update. Calculating this metric for a LOB is more difficult. Recall the update problem associated with LOBs. In order to insert a new object within the LOB, we must physically shift the data that is positioned after the new object to make room for the new object. Similarly, to delete objects or replace objects with different sized objects, we must shift data as well. For this measure, we assume that objects are updated in the middle of the LOB on average. Thus, to insert a new object, we must write the new object, plus its overhead, plus half of the current size of the LOB in order to shift data to make room for the new object. Rewriting half of the LOB for every update requires the amount of data written to the LOB to increase greatly as the number of updates increase (note the logarithmic scale of the Y axis in Figure 13). This measure emphasizes the impact of the update problem associated with LOBs. Because SLOBs overcome the update problem, making updates requires that the new object be appended and the sequence index rewritten. Therefore, dramatically fewer bytes are written when updating SLOBs versus LOBs.

While using SLOBs instead of LOBs for storing structured objects alleviates the update problem, SLOBs do have a greater storage overhead than LOBs. In Figure 14, we graph the amount of data stored in a SLOB or LOB after user executes a series of updates. Because LOBs must store data in-order, the resulting LOB will always be smaller than the resulting SLOB. This is due to the storage overhead required to store the sequence index in SLOBs. In Table 1, we provide a separate formula for each level of SLOB. Recall that

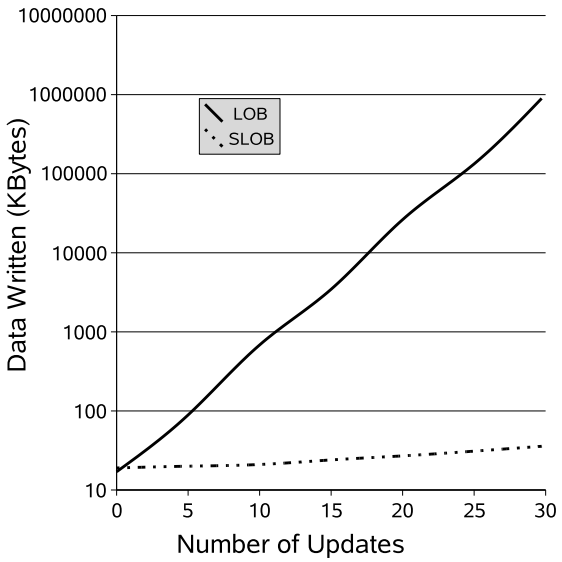

Figure 13: Number of bytes written on updates. Note the logarithmic scale of the Y axis

level 1 SLOBs do not allow any data to be truncated. Thus, after each update, the sequence index for the old SLOB is not physically removed from storage; it is removed from the new sequence index so that it logically does not exist any more. We denote this the *worst case SLOB*, and the large overhead for storing the sequence indexes is evident in its graph. Level 2 SLOBs allow the physical truncation of data. Thus, when an update occurs, the old sequence index can be physically removed from the SLOB and the new sequence index written in its place. By simply allowing truncation, the storage overhead of SLOBs drops dramatically. The level 3 SLOB allows some data to be overwritten when an object update is made. In general, we assume that the amount of bytes that can be reused is relatively low. For this graph, we assume that 20 percent of the updated bytes will overwrite existing bytes. As expected, the level 3 SLOB has even lower storage overhead than the level 2 slob. In the case where the size of base objects is fixed, a level 3 SLOB would have the same storage overhead as the LOB implementation, plus 8 bytes for the single range sequence index. Although SLOBs of all levels incur some storage overhead when storing structured objects as compared to LOBs, the performance they gain by alleviating the update problem shows that they are a viable alternative to storing structured objects in LOBs.

6 Conclusions and Future Work

It is generally acknowledged that emerging applications require more functionality than what is offered by unstructured storage types for managing complex, structured data. Even though it is possible to use unstructured storage types to support structured data, this results in many problems as identified in Section 1 (the *acceptance*, *abstraction*, *generality* and *update* problems). The main contribution of this work is that we propose a novel SLOB concept and SQL data type that alleviates these problems through the introduction of two important mechanisms: the structure index and sequence index. An implementation of this concept together with a performance evaluation effectively validates the proposed concepts and shows the superiority of our SLOB concept over existing unstructured LOB mechanisms for storing and manipulating structured large objects. As a tradeoff, we pay only a small storage overhead when using DBMSs providing level 2 or level 3 LOBs, which are available in most CDBMSs today.

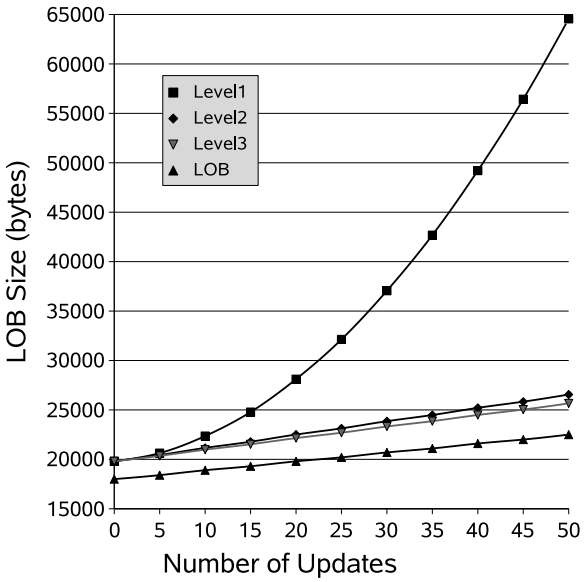

Figure 14: Size of the underlying LOB object

There are a number of topics that we consider for future research study and development. Since we have so far demonstrated a SLOB implementation for level 1 LOBs, we will continue with the SLOB implementation for level 2 and level 3 LOBs in the near future. Having this complete package, we can tailor a specialized and optimized SLOB implementation for each of the popular CDBMSs. This allows us to implement a highly complex algebra, such as the two-dimensional Spatial Algebra (SPAL2D), using SLOBs to seamlessly integrate them with all popular CDBMSs. Furthermore, we plan to develop concepts for *SLOB-based indexes* for SLOBs as well as an algebra specification language for designing algebras and constructing complex structured objects.

References

[1] P. M. Aoki. How to Avoid Building Datablades that Know the Value of Everything and the Cost of Nothing. In *Int. Conf. on Scientific and Statistical Database Management*, pages 122–133, 1999.

[2] S. Banerjee, V. Krishnamurthy, and R. Murthy. All Your Data: The Oracle Extensibility Architecture. In K. R. Dittrich, editor, *Component Database Systems*, Morgan Kaufmann Series in Data Management Systems, chapter 3, pages 71–104. Morgan Kaufmann Publisher, 2001.

[3] D. S. Batory. Modeling the Storage Architectures of Commercial Database Systems. *ACM Trans. on Database Systems*, 10(4):463–528, 1985.

[4] 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(11):1711–1730, 1988.

[5] A. Biliris. An Efficient Database Storage Structure for Large Dynamic Objects. In *Int. Conf. on Data Engineering*, pages 301–308, 1992.

[6] A. Biliris. The Performance of Three Database Storage Structures for Managing Large Objects. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 276–285, 1992.

[7] M. Carey and L. Haas. Extensible Database Management Systems. *ACM SIGMOD Record*, 19(4):54–60, 1990.

[8] M. J. Carey, D. J. DeWitt, J. F. Richardson, and E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Int. Conf. on Very Large Data Bases*, pages 91–100, 1986.

[9] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A Data Model and Query Language for EXODUS. In *ACM SIGMOD Int. Conf. on Management of Data*, page 413423, 1988.

[10] J. Davis. IBM's DB2 Spatial Extender: Managing Geo-Spatial Information within the DBMS. Technical report, 1998.

[11] S. Dieker and R. H. Güting. Efficient Handling of Tuples with Embedded Large Objects. *Data & Knowledge Engineering*, 32(3):247–269, 2000.

[12] S. Dieker, R. H. Güting, and M. R. Luaces. A Tool for Nesting and Clustering Large Objects. In *Int. Conf. on Scientific and Statistical Database Management*, pages 169–181, 2000.

[13] R. Güting, M. Böhlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M.Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Trans. on Database Systems*, 25(1):881–901, 2000.

[14] R. H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4:100–143, 1995.

[15] 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*, 2(1):143–160, 1990.

[16] J. Hammer and M. Schneider. Genomics Algebra: A New, Integrating Data Model, Language, and Tool for Processing and Querying Genomic Information. In *Conference on Innovative Data Systems Research*, pages 176–187, 2003.

[17] B. Hwang, I. Jung, and S. Moon. Efficient Storage Management for Large Dynamic Objects. In *Euromicro Conf. on System Architecture and Integration*, pages 37–44, 1994.

[18] T. J. Lehman and B.-G. Lindsay. The Starburst Long Field Manager. In *15th Int. Conf. on Very Large Data Bases*, pages 375–383, 1989.

[19] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *Int. Conf. on Very Large Data Bases*, pages 294–305, 1988.

[20] S. L. Osborn and T. E. Heaven. The Design of a Relational Database System with Abstract Data Types for Domains. *ACM Trans. on Database Systems*, 11(3):357–373, 1986.

[21] L. A. Rowe and M. Stonebraker. The POSTGRES Data Model. In *Int. Conf. on Very Large Data Bases*, pages 83–96, 1987.

[22] 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*, 2(1):25–43, 1990.

[23] P. Seshadri. Enhanced Abstract Data Types in Object-Relational Databases. *VLDB Journal*, 7(3):130–140, 1998.

[24] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *2nd Int. Conf. on Data Engineering*, pages 262–269, 1986.

[25] M. Stonebraker and M. A. Olson. Large Object Support in POSTGRES. In *Int. Conf. on Data Engineering*, pages 355–362, 1993.

[26] M. Stonebraker, B. Rubenstein, and A. Guttman. Application of Abstract Data Types and Abstract Indices to CAD Databases. In *ACM/IEEE Conf. on Engineering Design Applications*, pages 107–113, 1983.

[27] P. F. Wilms, P. M. Schwarz, H.-J. Schek, and L. M. Haas. Incorporating Data Types in an Extensible Database Architecture. In *Int. Conf. on Data & Knowledge Bases*, pages 180–192, 1988.