Efficient Handling of Tuples with Embedded Large Objects

Stefan Dieker and Ralf Hartmut Güting
Praktische Informatik IV, FernUniversität Hagen
D-58084 Hagen, Germany
{stefan.dieker, gueting}@fernuni-hagen.de

Abstract

Modern database systems and storage manager toolkits usually provide a *large object* abstraction. Very often large objects are not used as standalone entities, but rather embedded within an aggregate of different types, i.e. a *tuple*. Depending on the large object's size and access probability, query performance is determined by the representation of the large object: either inlined within the aggregate or swapped out to a separate object. This paper describes a sound and general large object interface extension which automatically switches the representation of large objects according to their actual size. The optimum threshold size for switching the large object's representation is determined, based upon a linear cost model. Furthermore, a SHORE-based implementation and its performance are presented. It turns out that switching the representation of large objects yields great performance improvements for objects whose size is varying from quite small to large.

Key words Extensible database systems; Data type programming interface; Large objects; Tuple representation; SHORE

1 Introduction

Conventional relational database systems cannot meet the requirements from modern database application domains like GIS, CAD, spatial and spatio-temporal information systems, or the large and still growing field of multimedia processing. As a result of the great amount of research effort that has been done in order to overcome the limitations of relational database systems, new data models and systems implementing these models have arised, e.g. object-relational and object-oriented models and systems.

A common challenge for almost all new systems is the need to handle *large* objects, whose representations on secondary storage are exceeding a single disk page. Fortunately, building systems for advanced applications is supported by the storage manager components of toolkits like EXODUS [6] or SHORE [5], which provide all the typical DBMS features like transaction management, multiuser access control, and logging and recovery. These storage managers have proven to be reliable and efficient in handling standard data types as well as large objects.

Another approach to building systems for new application domains is extending an existing database system frame — usually tightly bound to a specific data model — by new abstract data types which can be used in the same way as standard data types. Access methods for new types have to be provided by the data type implementor using a well-defined application programming interface (*API*) to handle persistent data. This interface usually includes methods for access to large objects. Examples of such extensible systems are the commercially available Informix Universal Server or the SHORE-based PREDATOR system [14].

Work on large objects started with System R [1], supporting long fields of up to 32 Kilobytes. Partial access to long fields was not supported. Later on, Haskin and Lorie [8] presented an advanced mechanism to handle long fields of a maximum size of about 2 Megabytes.

The Wisconsin Storage System (WiSS) [7] provided partial access to large objects which could grow up to 1.6 Megabytes. Hence there is still a size limit predefined by the storage manager rather than just depending on the underlying hardware and operating system configuration. A common drawback of either system is the loss of sequentiality at the physical level, thereby giving rise to high access costs for objects stored on several pages.

EXODUS [6] not only overcame these limitations, but also introduced efficient handling of "large" objects which actually may be small, being represented by only a fractional amount of a disk page. The same holds, e.g., for the storage manager component of SHORE [5], the successor of EXODUS, and large objects (LOBs) in DB2 [10].

Several other approches to management of large objects have been explored, each of them emphasizing different properties. For instance, Starburst provides efficient read and append operations [11], EOS enables efficient partial insertion and deletion even in the middle of an object [2, 3], and BeSS [4] as well as Fellini [12] support the special requirements rising from multimedia applications.

In this paper we address the *usage* of large objects as they are offered by those systems. We do not propose any direct improvement of large object implementations, but rather introduce a new software layer on top of large object abstractions. Often large objects are not stand-alone entities, but components of a comprising structure that aggregates the large objects with other objects. Our approach enhances the efficiency of reading such aggregates without affecting any property of the large object implementation of the underlying storage manager. Thus we can still use its features concerning transaction management, concurrency control, logging and recovery, and handling of large objects.

The interface to a large object typically provides a minimum set of methods, including creation, deletion, resizing, reading, and updating large objects. The possibility of resizing an object gives rise to not only use the large object abstraction for objects that are really *large*, but rather for all objects that are *variable in size.* Moreover, some data type representations are potentially large, but not necessarily: An instance of a polygon data type, essentially represented by a set of vertex coordinates, may be a triangle as well as a region defined by thousands of vertices. The data type implementor should be allowed to use the same large object abstraction for all possible instantiations of polygons without loss of efficiency in case of actually small objects.

In case an aggregate contains embedded variable-sized objects, an important decision is how to store the aggregates and their contents:

Alternative 1 Use a single object to store the aggregation entirely. If the size of the complete aggregate is small, this is the right choice, since the object can be read via a single disk access. But if the size of any embedded large object is larger than some threshold size, e.g. a single disk page, the large object should be swapped out, only leaving a small access handle to the large object within the aggregate, since not all components of an aggregate are necessarily read when it is loaded into main memory.

Alternative 2 Store each large object within a storage space dedicated solely to the large object, leaving a reference handle within the aggregate that logically contains the large object. In case of really large objects this has the advantage of not always transferring lots of bytes from disk to main memory, even if the large object is not going to be read at all. In case of small objects, however, this strategy might force multiple disk access operations to read a single aggregate completely.

Consider a relational GIS containing a relation `cities (name: STRING[20], population: INTEGER, area: INTEGER, shape: POLYGON)`. Many queries involving `cities` will not examine the `shape` attribute. This should be taken into account for the physical database design: if the size of a `shape` is large (because information about the shape of the actual city is detailed), it will be more efficient to vertically partition the relation in such a way that the `shape` value is stored externally, only loading it to main memory on demand. On the other hand, if `shape` is just a rectangle it does not hurt to store the `shape` value within the byte string representing the tuple, thereby avoiding additional disk access whenever the shape of the cities is to be read.

Such situations arise not only in relational systems, but whenever single data types may be logically combined to an aggregate. Hence the mechanisms presented in this paper are not restricted to implementations of relational systems, although, for convenience, in the sequel we will use the terms "tuple" and "attribute" in place of "aggregation" and "component", respectively.

In this paper we present a mechanism to handle large objects via a simple and clean interface while automatically switching from an in-aggregation representation to a stand-alone representation and vice versa, based on the size of a large object, whenever its comprising aggregation is to be stored on disk.

The remainder of this document is structured as follows. Section 2 presents the basic concepts of our approach. Section 3 deals with the analysis of tuple access costs, based upon a linear cost model, in order to find a well performing threshold size indicating whether a large object should be swapped out or stored as part of the tuple string. In Section 4 we describe a C++ implementation of our approach, using the SHORE storage manager. Its performance is presented in Section 5. Section 6 compares our approach to other related work, and Section 7 concludes the paper.

2 Basic Concepts

As demonstrated by the example in Section 1, an application implementor might wish to use the large object abstraction of the underlying storage manager, while actual instances of large objects may indeed be small. In case of large objects being a component of a tuple, i.e. an attribute value or part of an attribute value, the large object implementation should not be used for writing the object to disk, but rather be replaced by a more efficient implementation which exploits the fact that the "large" object is embedded within an environment that has to be saved anyway. The data type implementor, however, should as little as possible be bothered with such efficiency problems in order to leave his focus of concentration on data type functionality issues.

Our approach to solving the conflict between efficiency and soundness of the API adds another interface level between application programs and storage manager interface, essentially providing a top level variable-sized object abstraction and a tuple abstraction which efficiently handles attribute values possibly consisting of variable-sized objects.

We assume the existence of a storage manager or extensible database API providing a large object abstraction with the following properties:

Identity A unique object identifier corresponds to each large object. This identifier is persistent, i.e. the object may be retrieved by means of the object identifier at any point of time after creation, regardless of whether the object actually has an in-memory representation or not, until the object is deleted explicitly.

Arbitrary size The size of large objects may range from one word of storage to unlimited size (within the bounds of the underlying operating system and hardware).

Random access Read and write operations may concern the whole object as well as a specific number of bytes within the object.

Resizability An existing large object may be made smaller or larger. In the former case, the object's rear is cut off with respect to the required new size. In the latter case, an appropriate amount of storage is appended at the end of the original object.

These specifications are, for instance, met by *records* of the SHORE Storage Manager or by *SLOBs (Smart Large Objects)* of Informix Universal Server. Actually, existing storage managers provide many more access methods for their respective large object abstraction. For illustration of our approch, regarding just the mentioned properties is sufficient. We will refer to the large object abstraction of whatever storage manager is used to implement them by the term *LOB (large object)*.

On top of the storage manager or API, we define a new large object abstraction that offers most access methods of the original one. We call this abstraction *FLOB (Faked Large Object)*. The substantial difference between LOBs and FLOBs lies in the fact that a FLOB is not a persistent object on its own, but can only be used as part of an *attribute* which in turn is materialized in the context of a comprising *tuple*, as described below.

Creating a FLOB is implemented by either allocating some intermediate main memory structure (essentially a contiguous byte string of appropriate size) or using an underlying LOB. Whenever a tuple is restored which contains FLOBs, these FLOBs are restored subsequently by means of either a persistent LOB identifier or the main memory byte string — as part of the comprising tuple byte string — representing the LOB value.

Since an implementation of an attribute data type may contain FLOBs, we separate an attribute's representation into a fixed-length and a variable-length part. The former one we call *attribute core*, the latter one *attribute extension*. An attribute type implementation which employs not only fixed length components, but also FLOBs, leaves fixed-length FLOB handles within its core. The attribute extension is the set of all byte strings used to represent the values of an attribute's FLOBs.

While for each attribute there is an attribute core, the extension of an attribute data type value may be empty for two reasons:

- The attribute type does not use any FLOB, hence the attribute extension is never used.
- The attribute type does use FLOBs, but all of them are represented by LOBs of the underlying storage manager.

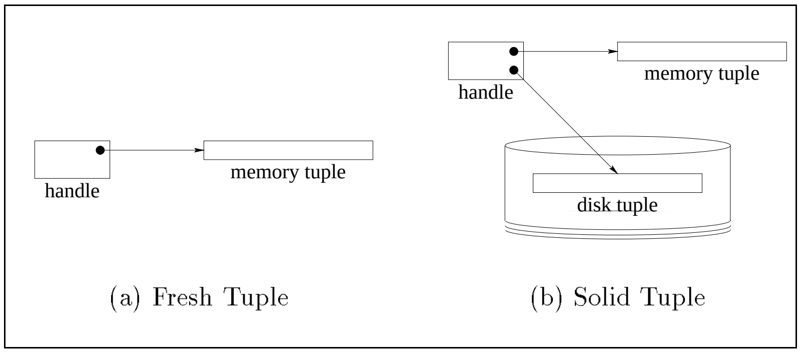


Figure 1: Tuple States

Attribute values are never standalone objects, but exist in the context of a comprising tuple only. We use the *tuple* abstraction to aggregate attribute values which might contain FLOBs. Tuple value representations may be stored on disk (*disk tuple*) or within main memory (*memory tuple*). Access to either representation is provided via a main memory *handle*. Figure 1 illustrates both possible forms of tuple appearance at runtime: a *fresh tuple*, consisting of a handle and a memory tuple referenced by the handle, and a *solid tuple* with a handle referencing both the memory tuple and the disk tuple.

When a tuple is saved, a contiguous tuple byte string is created as follows. First, all attribute core values are stored sequentially. After that, for each involved FLOB the decision is taken whether the string representing its value is appended to the tuple or swapped out. Finally, the tuple byte string is written to disk (controlled by transaction management of the underlying storage manager). When opening a tuple, each involved FLOB handle is assigned either the underlying large object or the corresponding byte string within the tuple representation.

So far we have described a set of general abstractions which support efficient and comfortable handling of attribute data types whose representation may employ large objects. However, as we will see below, providing an efficient implementation of the general concept is a non-trivial task. In Section 4 we present a sample implementation on top of the SHORE storage manager, using the C++ programming language. While some details are specific to SHORE and C++, many aspects of our solution are still transferable to other environments.

3 Threshold Size Analysis

When saving a tuple, for each FLOB we have to decide whether it should be saved as part of the tuple byte string or by a LOB on its own. For this purpose we use a *threshold size* parameter. If the FLOB to be saved is larger than the threshold size, it is swapped out to a separate LOB; otherwise it is appended to the tuple string. In this section, we analyse the impact of the threshold size on tuple access costs in order to identify good threshold sizes.

3.1 Cost Model

Storage managers typically arrange the pages representing a LOB value in *segments* of adjacent disk pages. For each LOB there is an index, for instance a B-tree, which keeps track of all the segments used to store a LOB value. It depends on the specific implementation and its applications whether LOBs of a given size tend to consist of some large segments or many small segments.

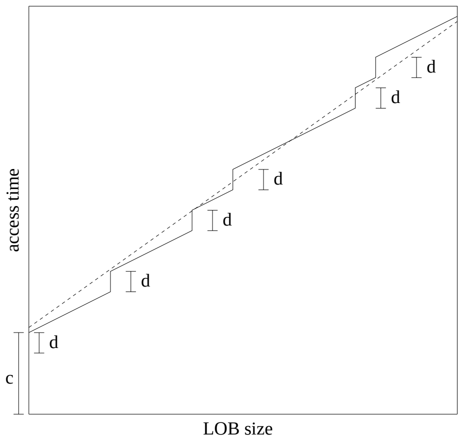


Figure 2: LOB Access Time, Qualitative

The solid line in Figure 2 qualitatively illustrates how the read access time for a LOB might depend on its size. The linear line segments depict reading a single segment of pages. Each segment is preceded by a *segment offset* d corresponding to the mean time for searching the LOB index and positioning the disk head. Finally, the start offset c is the sum of the time needed by the storage manager to prepare read access, for instance acquiring locks and loading the LOB index, and the segment offset.

The dotted line shows a linear approximation of the solid line. Due to the constant gradient of costs for reading a single segment, the linear approximation is very good if LOBs are stored in only a few, large segments. In practice, however, also for LOBs stored in many small segments the linear approximation is of sufficient quality due to the small deviation of segment offsets from the constant value d and the small variance in the length of segments.

Thus, to determine a threshold size indicating which FLOB values should be swapped out of the comprising tuple, we use a linear cost model, assuming that the time needed to read n disk pages is computed sufficiently well by

$$C_P(n) = c + g \cdot n \tag{1}$$

with n the number of disk pages to be read, c the time needed for some initial actions taking place exactly once, and gradient g the time needed to read a single disk page.

Within the context of this paper, our view of a tuple is that of a pair $T = (t, \boldsymbol{F})$, consisting of a core tuple t and a set $\boldsymbol{F} = \{f_1, f_2, \ldots, f_n\}$ of n FLOBs, $n \geq 0$. Due to the variable size of FLOBs we cannot calculate the costs of reading a tuple exactly, but we can approximate the expected costs if we know the probability r_f of a FLOB f to be read whenever the comprising tuple is read:

$$C_T(t, \boldsymbol{F}) = C_P(s_t) + \sum_{f \in \boldsymbol{F}} \begin{cases} r_f \cdot C_P(s_f) + (1 - r_f) \cdot 0 & \text{if } f \text{ is LOB} \\ r_f \cdot g \cdot s_f \quad + (1 - r_f) \cdot g \cdot s_f & \text{otherwise} \end{cases} \tag{2}$$

with s_x being the size of object x in number of pages, rounded up to entire pages.

Notice that for every FLOB being read Equation 2 adds the costs of reading the FLOB. This might yield too high cost values since several small FLOBs may be read by accessing a single page once. On the other hand, FLOBs are intended to be used for representing objects which might be small as well as very large without knowing much about the FLOB size distribution. Hence for our purposes the mean fault of applying Equation 2 is expected to be small enough to be ignored.

In most cases the core tuple will fit onto a single disk page. Applying this assumption yields the following simplified cost formula:

$$C_T(t, \boldsymbol{F}) = C_P(1) + \sum_{f \in \boldsymbol{F}} \begin{cases} r_f \cdot C_P(s_f) & \text{if } f \text{ is LOB} \\ g \cdot s_f & \text{otherwise} \end{cases} \tag{3}$$

3.2 Threshold Size

Obviously, Equation 3 returns minimal costs if each of the terms of the sum returns minimal costs. This in turn holds if for each of the FLOBs f_1 to f_n the better of either representation alternative — inlined or swapped out — has been chosen. In the following we determine the FLOB threshold size, a simple means to decide on the representation of a FLOB.

According to Equation 3, reading a single FLOB f represented as a LOB yields minimum costs if the inequation

$$r_f \cdot C_P(s_f) < g \cdot s_f$$

holds, which evaluates to

$$\frac{c}{g} \cdot \frac{r_f}{1 - r_f} < s_f$$

The ratio $\frac{c}{g}$ is a system dependent constant. Thus, the optimal threshold size for a FLOB f is a function of $\frac{c}{g}$ and r_f:

$$S(\frac{c}{g}, r_f) = \frac{c}{g} \cdot \frac{r_f}{1 - r_f} \tag{4}$$

Figure 3 illustrates the relationship between r_f and the optimal threshold size for some selected values of $\frac{c}{g}$. Notice the log scaled y-axis. This graph can be used to determine the optimum threshold size of a single FLOB f for a specific system as follows.

1. Determine the system constant $\frac{c}{g}$.

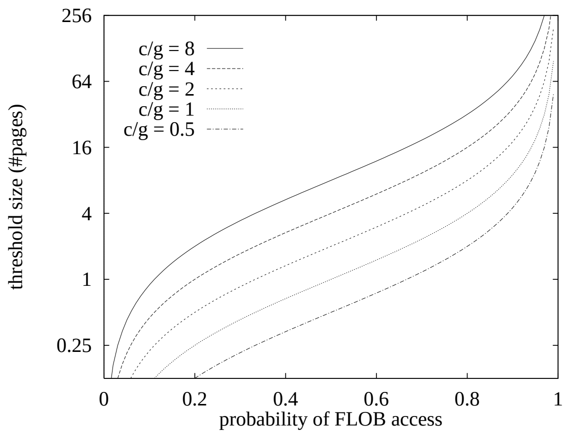


Figure 3: FLOB Threshold Size

2. Determine the read probability r_f of f.

3. Find the corresponding threshold size s within Figure 3. Read access of f being represented by an underlying LOB performs better than read access of f being part of the corresponding tuple string iff $s_f > s$. Hence choosing s as threshold size for FLOB f yields minimal read costs.

So far, we have developed a method to determine the threshold size for single FLOBs. A straightforward way to implement this insight would be to maintain an access ratio variable for each FLOB, initially set to, for instance, $\frac{1}{2}$, and readjust it during runtime by means of actual access measurements.

If, however, maintaining FLOB access ratio variables is considered to be too expensive, we propose to use a single threshold variable for *all* FLOBs, set to a threshold size s (in pages) according to a read probability $\frac{1}{2}$ by means of Equation 4:

$$s = S(\frac{c}{g}, \frac{1}{2}) = \frac{c}{g} \cdot \frac{\frac{1}{2}}{1 - \frac{1}{2}} = \frac{c}{g} \tag{5}$$

The experiments presented in Section 5 confirm that using the threshold size $\frac{c}{g}$ yields very good FLOB access performance for a broad range of access ratios.

3.3 Caching

So far, we have determined an optimal FLOB threshold size by cost analysis based upon a clean and simple model for tuple access costs. We chose this model because it allows for relatively easy mathematical analysis and, on the other hand, results in a threshold size formula as simple as Equation 5, using a single system parameter only. In Section 5.1 we see how it can be determined easily.

In practice, however, we find further parameters which have some impact on the recommendable threshold size. The probably most important one is *caching*. If a tuple containing no swapped-out FLOBs is read several times and is not flushed off the cache between two accesses, only the costs of the first access count, because the time needed to read from main memory can be ignored compared to reading from disk. Similarly, for each swapped-out FLOB only the first access counts as long as it remains in the cache.

To integrate these observations with the cost model presented in Section 3.1, we introduce a new parameter N. The value of N is the number of times a tuple is read without being flushed off the cache. Then the *amortized* costs of once reading n disk pages are given by

$$\overline{C_P}(n) = \frac{C_P(n)}{N} = \frac{c + g \cdot n}{N} \tag{6}$$

In computing $\overline{C_P}$, we distribute the costs of once loading the pages from disk into main memory among all N accesses.

The amortized costs of reading a tuple with known read probability r_f for its FLOBs can be calculated by a modified version of Equation 2. Again we add the costs of reading the core tuple t and the expected costs for each involved FLOB $f \in \boldsymbol{F}$. In addition to distributing read costs among all N accesses, we regard cache effects by adjusting the FLOB read probability used to weight the costs for a single FLOB access to

$$\overline{r}_f = 1 - (1 - r_f)^N \tag{7}$$

The value $\overline{r}_f$ is the probability that in N tuple accesses the comprised FLOB f is read *at least once*. Then the amortized costs for reading a tuple compute to

$$\overline{C_T}(t, \boldsymbol{F}) = \overline{C_P}(s_t) + \sum_{f \in \boldsymbol{F}} \begin{cases} \frac{\overline{r}_f \cdot C_P(s_f) + (1 - \overline{r}_f) \cdot 0}{N} & \text{if } f \text{ is LOB} \\ \frac{\overline{r}_f \cdot g \cdot s_f + (1 - \overline{r}_f) \cdot g \cdot s_f}{N} & \text{otherwise} \end{cases}$$

On the assumption that the core tuple fits into a single disk page, the above formula is simplified to

$$\overline{C_T}(t, \boldsymbol{F}) = \overline{C_P}(1) + \sum_{f \in \boldsymbol{F}} \begin{cases} \overline{r}_f \cdot \overline{C_P}(s_f) & \text{if } f \text{ is LOB} \\ \frac{g \cdot s_f}{N} & \text{otherwise} \end{cases} \tag{8}$$

Computing the optimal FLOB threshold size $\overline{S}$ following the strategy from Section 3.2, but now taking into account cache effects by means of Equation 8, yields

$$\overline{S}(\frac{c}{g}, r_f) = \frac{c}{g} \cdot \frac{\overline{r}_f}{1 - \overline{r}_f} \tag{9}$$

In practice Equation 9 cannot be used directly, because it is not possible to specify the parameter N exactly. The value of N does not only depend on the cache size, but also on other influences like, for instance, the size of the tuples and FLOBs actually being read, the access patterns of the application, i.e. how often is the same tuple or FLOB read more than once without being pushed away by other tuples or FLOBs, or, in a multi-user environment, the activity of concurrent processes.

Nevertheless we can make qualitative use of Equation 9. It defines $\overline{S}$ in the same way as S is defined in Equation 4, just replacing r_f by $\overline{r}_f$. Thus, if FLOBs are used in an environment where effective caching is done ($N > 1$), experiments should be performed to increase the threshold size returned by S, since

$$\overline{S}(\frac{c}{g}, r_f) \; > \; S(\frac{c}{g}, r_f) \quad \forall \quad N > 1$$

This result can be understood intuitively as follows. Provided that some effective caching takes place, it might be a good idea to load a FLOB together with its comprising tuple from disk into main memory even if at first the FLOB is not read, because there is still a chance that the FLOB will be read from cache later on. Thus, with caching the probabiblity that it is beneficial to store a FLOB inlined is higher than without caching. Increasing the number of FLOBs stored inlined is just the effect of increasing the threshold size.

3.4 Further Parameters

In Section 3.1 we already mentioned that a large object is typically represented by segments. It partially depends on the implementation strategy of the underlying storage manager whether the representation tends to consist of many small segments or only a few large ones.

Unfortunately, also the *application* which uses large objects influences their representation. For instance, if large objects usually are created in a single step, their representation will typically employ a few large segments. In contrast to that, stepwise creation of large objects, in each step appending a small portion to the existing object, will probably result in a representation scattering a lot of small segments around the disk. In addition to that, updates, in particular insertions, might force the storage manager to break large segments into smaller ones. Thus, apart from the storage manager, the general representation of large objects is highly application dependent and might change over time with changing applications. As a consequence, also the system parameter $\frac{c}{g}$ can change with different applications and should be determined regularly.

Furthermore, the application's behaviour with respect to creation and deletion influences the choice of a good threshold size. If the application tends to create and delete tuples very often, while read access is a relatively rare event, it is advisable to increase the threshold size, because creation and deletion of a tuple represented as a single, even large, byte string clearly outperforms creation and deletion of a tuple consisting of several distinct persistent objects.

4 Implementation

Within this section, we at first present general interfaces for the abstractions used in our approach. Thereafter, we describe the C++ implementation of these interfaces on top of the SHORE storage manager. The contribution of this section is twofold: We demonstrate that the FLOB concept is implementable by mapping the abstractions introduced in Section 2 to C++ classes in a straightforward manner. On the other hand, we show by example the variations and extensions to the abstract concept necessary to exploit a specific implementation environment.

4.1 General Interfaces

```
SORTS integer, Address, LOB, ID
OPS      Create:                                     integer → ref(LOB)
           Open:                                          ID → ref(LOB)
          Close:                                    ref(LOB) → []
        Destroy:                                    ref(LOB) → []
          GetId:                                    ref(LOB) → ID
        GetSize:                                    ref(LOB) → integer
         Resize:                          ref(LOB) × integer → ref(LOB)
           Read:                ref(LOB) × integer × integer → Address
          Write: ref(LOB) × Address × integer × integer → ref(LOB)
```

Signature 1: Large Object

Signature 1 depicts the operations that we assume to be applicable for LOBs in order to be able to implement the FLOB concept. The sort `ref(LOB)` reflects the fact that a LOB actually is accessed via a main memory *handle* uniquely referencing the underlying LOB.

The semantics of the operations of Signature 1 are as follows.

`Create(size)` A LOB of `size` bytes and a corresponding handle are created.

`Open(id)` The LOB denoted by the unique persistent object identifier `id` is opened by creating a handle.

`Close(lob)` The LOB referenced by `lob` is closed by destroying the handle.

`Destroy(lob)` `lob`'s LOB is destroyed. Its persistent id becomes invalid, as does the main memory handle.

`GetId(lob)` The persistent id of `lob`'s LOB is returned.

`GetSize(lob)` The size of the `lob`'s LOB in bytes is returned.

`Resize(lob, size)` The size of `lob`'s LOB is changed to `size` bytes by either appending new memory space at the end or cutting off the rear part of appropriate size. The contents of the untouched front part of the LOB remains unchanged.

`Read(lob, offset, length)` This function returns the main memory address of the portion of `lob`'s LOB denoted by `offset` and `length`.

`Write(lob, source, offset, length)` The portion of `lob`'s LOB denoted by `offset` and `length` is updated to the value pointed to by `source`.

Signature 2 summarizes the complete FLOB signature. Following the general concept, the FLOB interface is very similar to the LOB interface. Since a FLOB is not a persistent object on its own, however, there is no `GetId` method. Furthermore, in addition to the `Open` method of the LOB interface taking a persistent LOB id as argument, the FLOB interface offers another constructor:

```
SORTS integer, Address, FLOB, ID
OPS        Create:                           integer → FLOB
             Open:                                ID → FLOB
       OpenMemory:                 Address × integer → FLOB
            Close:                              FLOB → []
          Destroy:                              FLOB → []
          GetSize:                              FLOB → integer
           Resize:                    FLOB × integer → FLOB
             Read:          FLOB × integer × integer → Address
            Write: FLOB × Address × integer × integer → FLOB
```

Signature 2: Faked Large Object

`OpenMemory(start, length)` A FLOB is restored by means of the byte string containing `length` bytes, beginning at main memory position `start`.

The `OpenMemory` FLOB constructor is called when a tuple is opened which contains FLOBs whose value is represented within the tuple byte string.

Read/Write access methods of FLOBs check whether the FLOB is represented by means of a main memory structure or a LOB. Either the respective access calls are passed to the underlying large object or the main memory structure is changed appropriately. Efficient handling of FLOBs employed by attribute data types is performed by

```
SORTS Attribute, AttributeType, Tuple, ID, integer
OPS     Create:                AttributeType+ → Tuple
          Save:                         Tuple → Tuple
          Open:                            ID → Tuple
         Close:                         Tuple → []
       Destroy:                         Tuple → []
         GetID:                         Tuple → ID
           Get:               Tuple × integer → Attribute
           Put: Tuple × integer × Attribute → Tuple
```

Signature 3: Tuple

the implementation of the tuple interface presented in Signature 3. The semantics of operations are as follows.

`Create(typelist)` A fresh tuple instance is created. The `typelist` is a list of unique persistent type descriptors, identifying the type of each attribute of the actual tuple.

`Save(tuple)` Transform a given fresh tuple into a solid one.

`Open(id)` Open an existing tuple by means of the unique persistent tuple identifier `id`, resulting in a solid tuple.

`Close(tuple)` The main memory handle is deleted.

`Destroy(tuple)` The `tuple` instance is destroyed completely.

`GetId(tuple)` Return the unique persistent identifier of `tuple`.

`Get(tuple, n)` Return the `nth` attribute of `tuple`.

`Put(tuple, n, attr)` (Re-)Set the `nth` attribute of `tuple` to the value of `attr`.

In a given programming environment, the underlying storage manager provides the interface for its large object abstraction. On top of that, the specific constructs of the employed programming language are used to define data structures implementing the FLOB and tuple interfaces, taking into account the characteristics of the storage manager and the programming language. In the following subsection, we illustrate this process by a C++ implementation using the SHORE storage manager.

4.2 Sample Implementation

4.2.1 Overview

We decided to use the storage manager component of SHORE and the C++ programming language [15]. SHORE is a modern system, paying special regard to efficient object management, easy to install and use, equipped with a very good documentation, and, last not least, available for free. Once SHORE was chosen, using C++ is straightforward since it is the native SHORE implementation and application programming language. In addition to that, C++ supports low-level operations on single bytes within main memory as well as high-level object-oriented modeling of data, hence C++ is most appropriate for our purposes, anyway.

SHORE's abstraction of variable sized objects is called *record*. In addition to the LOB functionality, a record offers several other access methods we are not going to bother with in the context of this paper. Hence we continue using the LOB abstraction, which is implemented as a C++ class. It essentially offers an easy-to-use subset of SHORE's record functionality. SHORE supports (and requires) clustering of an arbitrary number of records within a *file*. We will not use any file functionality beyond file creation and deletion.

For each of the signatures presented in Section 4.1 our implementation defines a corresponding C++ class. Instances of classes `LOB`, `FLOB`, and `Tuple` are main memory handles referencing the respective persistent abstraction. In addition to that, we define a class `Attribute`, serving as a base class for all user defined attribute classes, consisting of two virtual functions returning the number of employed FLOBs and their addresses, respectively.

To each operation defined in the signatures given above there is a corresponding method in the respective class. Notice that `Create` and `Open` operations are implemented by class constructors, `Close` operations by destructors.

Since SHORE records are not only efficient and user-friendly representations for *large* objects, but rather for all objects whose logical representation is a contiguous byte string, we exploit them — via the LOB abstraction — not only for implementing FLOBs, but also for the persistent representation of tuples. This simplifies tuple implementation because a distinction between fixed-length and variable-length tuples is not necessary.

4.2.2 Adapting the Interfaces

Using SHORE, a large record cannot be read within a single operation, since SHORE implements read-only record access in page-sized portions. This gives rise to an extension of the general LOB interface as follows. On the one hand, the SHORE approach is efficient because unnecessary copying from SHORE's system buffer to the application's memory space is avoided in cases where indeed not an entire large object is needed, but only a small portion of the object. On the other hand, this solution is uncomfortable if the user really wants to read a large part of the object. Thus our LOB implementation supports either read access method: read-only access without copying, restricted to page boundaries, as well as copying an arbitrary clipping of a large object representation to a memory space provided by the user. So class `LOB` offers two method for read access, namely `Pin` and `Read`, respectively.

Regarding the FLOB interface, there is no FLOB constructor corresponding to the `Open` method of Signature 2, which opens from a given LOB. Neither may the user call such a constructor, because the underlying representation of FLOB values is user transparent, nor is it needed for tuple reconstruction, because a FLOB instance stores the id of its underlying LOB in an appropriate member variable. Thus, when a FLOB instance located within an attribute value is reconstructed, as a positive side effect the respective member variable referencing the LOB is reconstructed, too.

```
class Tuple
{
public:
  Tuple(TupleType *type);                                        /* Create a fresh tuple */
  Tuple(ID lob, TupleType *type, OpenMode mode);                 /* Open a solid tuple */
  ~Tuple();                                                      /* Close the tuple */
  void Save();                                                   /* Commit changes */
  void SaveTo(File *tupleFile, File *lobFile);                   /* Make persistent */
  void ReInit();                                                 /* Destroy the disk tuple */
  ID GetId();                                                    /* Return persistent id */
  Attribute *Get(int attrno);                                    /* Read attribute value */
  void Put(int attrno, Attribute *value);                        /* Update attribute */
  void DelPut(int attrno, Attribute *value);                     /* Update attribute */
  void AttrPut(int attrno_to, Tuple *tup, int attrno_from);      /* Update attribute */
}
```

Figure 4: Public Methods of Class Tuple

Class `Tuple`, depicted in Figure 4, is subject to several interface modifications. The `Open` operation of Signature 3, accepting a single argument, is implemented by a constructor with two additional parameters of type `TupleType` and `OpenMode`. While in general it is sufficient to pass the tuple type once at creation time and store the type information within the tuple byte string, we decided not to do so in order to avoid wasting memory when lots of tuples are of the same type. For instance, in a relational system all tuples of a relation have the same type. As a consequence, the tuple type is passed as an additional argument of the opening constructor. The third constructor parameter specifies whether the tuple is to be opened in read-only or write access mode. Our implementation uses this information to avoid copying in case of read-only access; the details are beyond the scope of this paper.

The operation `Destroy` of Signature 3 is replaced by the method `ReInit`. Like `Destroy`, `ReInit` destroys the persistent representation of the tuple value, i.e. the disk tuple (cf. Figure 1), but in contrast to `Destroy` the method `ReInit` leaves the handle and the memory tuple intact, thereby transforming a solid tuple into a fresh one. This modification is a contribution to the fact that the C++ class concept does not support multiple destructors. Furthermore, we save superfluous deletions and creations of tuple handles and memory tuples if tuple destruction and creation are performed successively.

For committing changes of a solid tuple by propagating the contents of its memory tuple to its disk tuple the parameterless method `Save` is provided. A fresh tuple, however, does not yet reference an underlying disk tuple, because a disk tuple is not created until the tuple has been saved once. Thus the method `SaveTo(tupleFile, lobFile)` is added, copying the memory tuple to a new LOB located within `tupleFile` and potentially swapping out LOBs representing large FLOBs to `lobFile`.

The `Put` method does not copy the attribute value passed as argument, but rather stores its *address* only, thereby avoiding copying if the tuple is not saved eventually. Such situations arise whenever intermediate tuples are processed. Using class `Tuple`, it turns out to be very convenient in some cases to have the passed attribute value deallocated automatically at tuple destruction time rather than by the previous caller of `Put`. Thus we offer another method providing just that functionality, namely `DelPut`.

We can even go one step further: It should be possible to put not only references to objects created by the user as new attribute values, but also references to attribute values that are located within another tuple! Consequently, we come up with a third kind of `Put` method, called `AttrPut`, ensuring that destroying tuple instances is delayed until their attribute values are no longer referenced by other tuples.

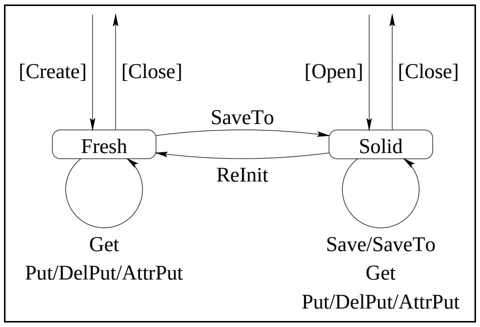


Figure 5: Tuple States and Operations

Following the interface description given above, Figure 5 summarizes states and operations of tuple instances. The operations enclosed by brackets actually are implemented by class constructors and destructors, respectively, while all other operations correspond to equally named methods.

4.2.3 Implementation Details

In order to illustrate the general process of tuple usage, Figure 6 shows a typical situation. Notice that the implementation of a memory tuple consists of a *tuple byte string* sufficiently large to contain all attribute core values as well as a *pointer tuple* referencing the actual attribute core values.

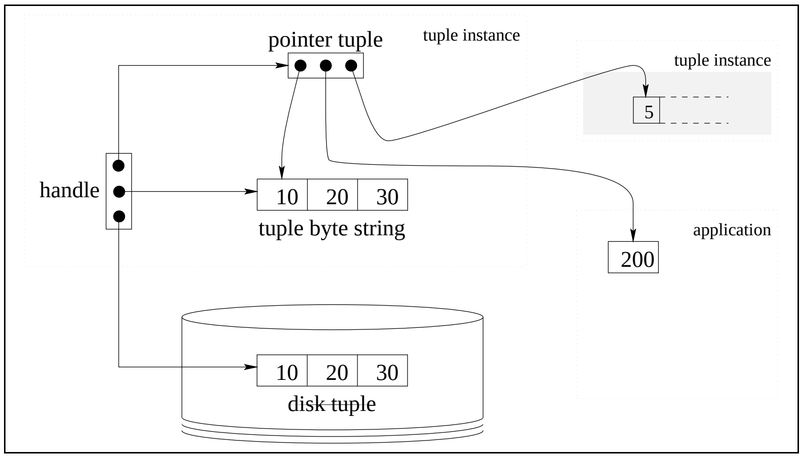


Figure 6: Snapshot on Tuple Instance

The depicted solid tuple instance has been constructed by means of a disk tuple containing three integer attributes: (10, 20, 30). A subsequent `Put` or `DelPut` has set the address of the second attribute to the address of another integer value outside the original tuple byte string. Furthermore, the third attribute value has been changed to a reference to the first attribute, value 5, of another tuple. Since no `Save` or `SaveTo` has taken place so far, pointer tuple and tuple byte string are not equivalent. As the logical value of a tuple is always defined by the pointer tuple, the tuple value is now (10, 200, 5).

For reasons of clarity, the above example does not consider the case that FLOBs are employed as attribute value members. Actually, the structure of a tuple instance remains the same, no matter whether FLOBs are involved or not. It's just the tuple byte string that might be extended by attribute extensions in case of small FLOBs. Figure 7 illustrates the effects of applying `SaveTo` to a fresh tuple's byte string consisting of attribute values which contain FLOBs.

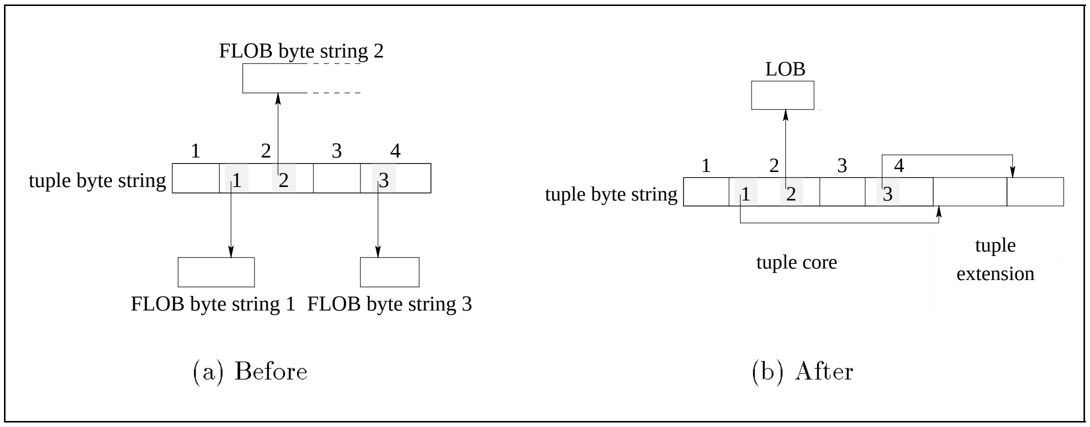


Figure 7: Tuple Byte String Before and After Applying `SaveTo`

Figure 7(a) shows the situation before calling `SaveTo`: The *tuple core*, i.e. the fixed-length part of the tuple byte string, contains the attribute cores. Attribute value 2 contains two FLOBs, attribute value 4 contains a single FLOB. All FLOBs are created from scratch, so their values are represented within byte strings scattered over main memory.

Figure 7(b) depicts the tuple byte string after `SaveTo` has been called: The tuple byte string still contains the tuple core. However, a *tuple extension* has been appended containing the value representation of FLOB 1 and 3, which are small. In contrast to that, FLOB 2 is a large one, hence it is represented by an underlying LOB.

5 Results

Within this section, we report on the results of some experiments we performed using the implementation of the FLOB concept presented in Section 4. Hardware platform is a SUN SPARCstation 20 with 64 MB of main memory and the Solaris 2.5.1 operating system running. We used the optimized version of SHORE 1.1.1 and the gcc 2.7.2.2 compiler with its 2.7.2 version of libraries. The SHORE buffer pool size is set to 1 MB.

5.1 Cost Parameters

In order to determine the system-dependent constants c and g of Equation 1 in Section 3.1, we subsequently create LOBs of size $< 1, 2, \ldots, 10 >$ disk pages and repeat this ordered creation 20 times, thereby avoiding many small lobs being placed on contiguous disk pages. This prevents us from misinterpretation of cache effects when reading LOBs ordered by size later on.

After LOB creation, the test data is read in LOB size order. The time necessary for reading all 20 LOBs of same size is logged once for each size. Figure 8 illustrates the results. The solid line depicts the dependency between the size of objects to be read and

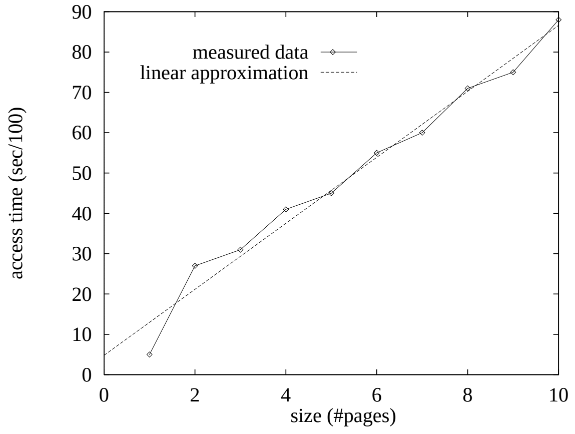


Figure 8: LOB Access Time

the elapsed time while reading 20 such objects. The time unit is $\frac{1}{100}$ second, which is the finest granularity for time measurements on the hardware we used.[1]

The dotted line is a linear approximation of the measured data, calculated by applying the method of least squares to the sample data. The resulting linear function is defined

[1]This coarse time granularity was the reason to repeat all object retrievals 20 times; otherwise the access of all objects with size ≤ 5 would have been reported to be 1.

by $y = 4.80 + 8.18x$. Thus we conclude the system dependent constant $\frac{c}{g}$ in Equation 3 to be $\frac{4.80}{8.18} \approx 0.6$. According to Equation 5, the recommended initial global threshold size is $\frac{c}{g}$ disk pages $\approx 0.6 \cdot 8120$ bytes $= 4872$ bytes.

5.2 Using FLOBs

We demonstrate the benefits of FLOB usage by another sequence of experiments, each of them employing a test database consisting of 400 tuples of type `(Integer, Polygon, Integer, Polygon)`. Each Polygon instance contains two FLOBs and some additional fixed-sized data members. The size of the core tuple computes to 168 bytes.[2] Within the experiment, we measure the time used for performing 1000 read access operations to the sample tuples in random order, thereby avoiding cache effects with respect to tuples. The FLOBs of a single tuple, on the other hand, have been created successively and therefore might be subject to cache effects in case of small FLOBs. We expect this situation to be characteristic for applications involving tuples with variable sized attributes.

For each experiment, we define

- a constant FLOB size for all FLOBs, ranging from 4 bytes to 32 disk pages.
- a constant FLOB threshold size s, with s being 0, 0.6, or 33 disk pages. With $s = 0$ each FLOB is a LOB, $s = 33$ is larger than the test FLOB, thus all FLOBs are located within the tuple extension, and $s = 0.6$ is expected to be a good value for automatically switching FLOBs.
- a constant FLOB access probability r, $r \in \left\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\right\}$.

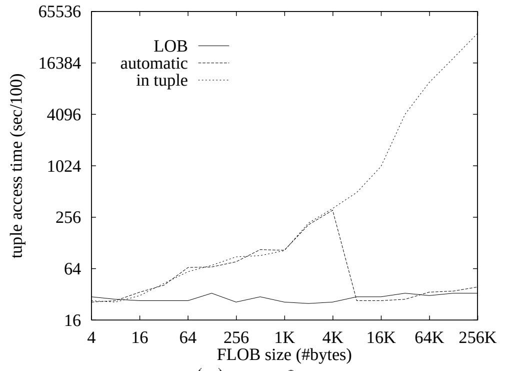
The results are shown in Figure 9. Notice the log scaled axes. These graphs depict the relationship between FLOB size and tuple access costs for the aforementioned three threshold sizes 0, 0.6, and 33 disk pages, labelled "LOB", "automatic", and "in Tuple", respectively. For each graph, FLOB access took place randomly with a given FLOB access probability.

As we expected, if FLOBs are never read (Figure 9(a)), for any FLOB size the best strategy is to store them as separate LOBs. On the other hand, if all of a tuple's FLOBs are read whenever the tuple is read (Figure 9(e)), storing FLOBs within tuples is always superior to swapping them out.

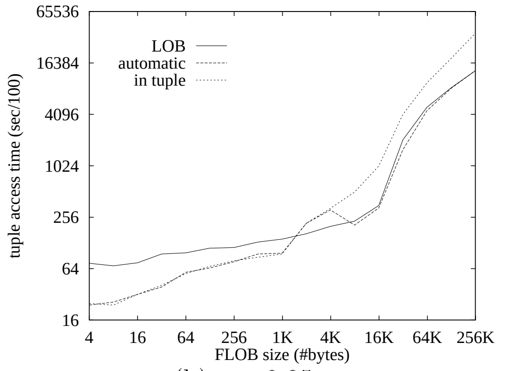
For more moderate access probabilities (Figure 9(b) to 9(d)) we find that in case of small FLOBs performance is better if they are located within the tuple, whereas large FLOBs perform better if being stored as LOBs. The point of intersection depicts the optimal FLOB threshold size; if FLOB representation is switching at exactly that size, we obtain minimal access costs for all possible FLOB sizes.

In Figure 9(c), we find that the threshold size of 0.6 disk pages performs quite well for $r = 0.5$. This is also true for $r = 0.75$, whereas for $r = 0.25$, with increasing size FLOB representation switches too late. However, the resulting cost overhead for FLOB sizes in the range of about 1.5 and 8 KB is still acceptable compared to the benefits for all sizes not within this range.

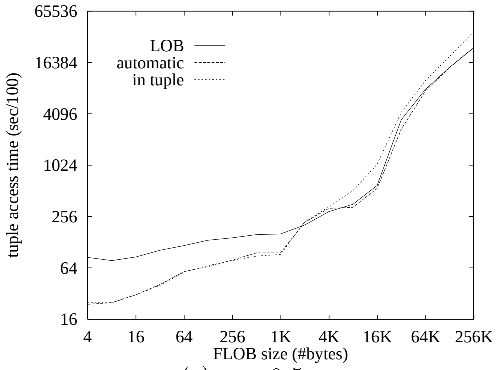
[2] sizeof(Integer) = 8, sizeof(Polygon) = 76.



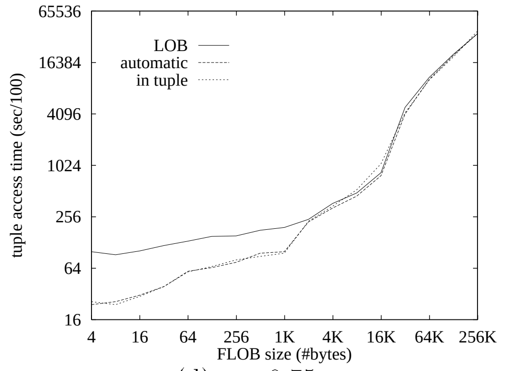
(a) $r = 0$



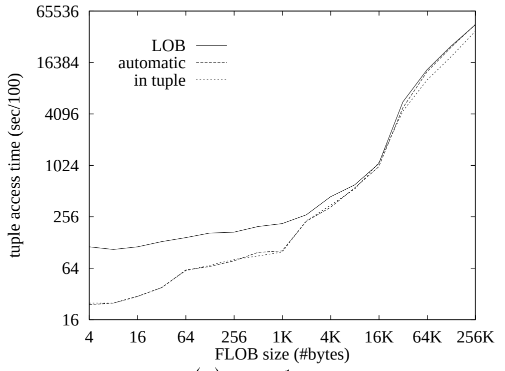
(b) $r = 0.25$



(c) $r = 0.5$



(d) $r = 0.75$



(e) $r = 1$

Figure 9: FLOB Size Versus Access Time

For $r = 0$ we cannot expect any benefit of an automatically switching FLOB representation. There is no cost curve intersection possible, hence we cannot find any sensible threshold value. Thus it is no surprise that in Figure 9(a) employing automatically switching FLOBs performs badly up to the point where they switch to LOB representation.

In principle, also for $r = 1$ the automatically switching FLOB representation cannot perform well. Nevertheless, Figure 9(e) illustrates that in this case performance of switching FLOBs is quite good. The reason is that for small sizes the better representation is chosen, while for large FLOBs the change for the worse is not significant. Remember that for large FLOBs there is not much access cost difference between representation strategies, since transfer costs are dominant to initial access costs if many bytes are read.

From the above observations we conclude the following findings:

- FLOB usage can reduce tuple access costs considerably.
- The decision, whether FLOB usage is beneficial or not, is quite insensitive to a large bandwidth of access probabilities even with a constant threshold size. In particular, even a very large access probability does not cause much harm compared to the advantage of FLOB usage for more moderate access probabilities.
- Only if FLOB access probability is very low there is a distinct negative effect of FLOB usage for FLOB sizes just smaller than the threshold size. An implementor should be aware of this fact, trying to identify such application data, and consequently avoiding FLOB usage for such extreme cases.

Hence we have demonstrated that in most cases FLOB usage is beneficial. In particular if an implementor is not able to identify expected size and size distribution of a data type whose representation he is going to implement, lower access costs will be achieved if FLOBs are always used as compared to the costs emerging from the strategy never to use FLOBs.

5.3 FLOBs versus LOBs

The last experiment is performed in order to find out the access cost overhead of FLOB usage compared to LOB usage when all FLOBs are represented as LOBs. Therefore, we again use the environment described in Section 5.2. In addition to that, we use an alternative implementation of the Polygon data type, replacing all FLOBs by LOBs. For both Polygon implementations, we measure the reading access time with access probability 1 in order to make access time differences appearing as distinct as possible.

We do not print the resulting graph due to the lack of information provided by the curves: There is *no* difference between reading LOBs directly or via the FLOB interface with respect to run time, even for very small LOBs. Actually, this result is not a surprise since the FLOB overhead for reading a LOB essentially is one additional function call.

6 Related Work

Lots of research efforts have been made to develop efficient implementations of persistent large objects. As a result, a variety of powerful large object representations is at the disposal of the implementor of a non-standard database system, all of them emphasizing different properties like efficient insertion in the middle of an object, guaranteed data troughput for sequential access, etc.

Within all those large object abstractions there is not paid any attention to the storage environment of large objects. As long as the user has direct access to the large object interface and is able to determine the way large objects are used for value representation — which is the case with toolkits like SHORE and object-relational systems like Informix Universal Server — the responsibility for efficient use of large objects is simply passed to the user. For instance, the Paradise GIS, using SHORE, does automatic switching from inlined to swapped out value representation with a threshold of about 0.7 disk pages for its array ADT [13]. However, in Paradise automatic switching is not a general concept but restricted to the array ADT.

With the Spatial Datablade of Informix Universal Server, a similar mechanism is implemented for its spatial data types. The API of Informix Universal Server allows one to implement representation switching by the notion of an *opaque type* [9]. The user has to define a set of support functions for each opaque type which will be called by the system frame for importing, exporting, casting, etc. of values of the respective type. Within the implementation of the `assign` support function the user has to code manually whether an instance of the respective opaque type should be inlined or swapped out.

The O_2 object manager, based upon WiSS, performs automatic representation switching for its tuple objects in order to enable tuples growing larger than a single disk page [16]. This is a contribution to the fact that the long data items of WiSS are suitable representations only for those objects actually exceeding one disk page. However, single attributes are not swapped out of the tuple, but the tuple object remains a contiguous object throughout its lifetime.

IBM's commercially available DB2 relational database system supports large objects, called LOBs, which are used to store data of actually *any* size [10]. But neither is representation switching performed, nor does the user have a chance to influence LOB representation manually, since user access to LOBs as for every other type is granted at the SQL top level interface, not at storage manager interface level.

7 Conclusions

The contribution of this paper is the definition of a sound and general interface providing access to a mechanism which automatically switches the representation of large objects embedded within tuples, thereby increasing tuple access performance. We deduced the optimum threshold size for single FLOBs analytically, and demonstrated the benefit of FLOB usage for a wide range of application scenarios even with one threshold size for all FLOBs, so we demonstrated empirically the benefit of using our design in practice.

From a software engineering point of view, one of the most important features of our design is the fact that the FLOB interface is an almost complete copy of the LOB interface provided by the underlying storage manager. Thus, updating legacy code is a very simple issue. A programmer being able to handle LOBs is able to deal with FLOBs as well.

As a positive side effect, FLOBs are not only useful for objects whose size is varying from very small to quite large, but also for those objects of varying size that are known to remain small enough to be inlined throughout lifetime. Consider a tuple type containing several such objects: Using FLOBs, inlining these objects is provided just by the way, whereas coding the inlined representation manually is a time consuming issue, potentially avoided by using LOBs instead, despite of the loss of performance.

Another case for FLOB usage are storage systems which offer different object abstractions for small and large objects. The FLOB approach enables the programmer to use a single interface for all objects of variable size by just adding some statements to the FLOB implementation in order to decide which object abstraction should be used. In addition to that, a storage manager might restrict the maximum size of tuple byte strings to some constant value, for instance a single disk page. Again, this restriction does not require any changes to the presented interfaces to be made, but is implemented by a small extension as follows. Since the only operation determining whether a FLOB is swapped out or not is the `Save` operation applied to a tuple, it is sufficient to extend the respective method implementation by not only taking into account the actual sizes of the involved FLOBs, but also the sum of these sizes. As a consequence, FLOBs might be swapped out whose size is less than the optimum threshold size.

Future work will focus on adding structure to large objects. While large object abstractions of todays storage managers provide access to *untyped* byte strings, support for more specific generic data types like arrays, lists, trees, or graphs, used within the implementation of attribute data types, is still an open issue. Here the main challenge is to provide insertion and deletion of elements at arbitrary positions without wasting space while maintaining specific clustering properties.

References

[1] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, K. P. Ewaran, Jim Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[2] Alexandros Biliris. An efficient database storage structure for large dynamic objects. In *Proceedings of the 8th International Conference on Data Engineering*, pages 301–309, Los Alamitos, February 1992. IEEE Computer Society Press.

[3] Alexandros Biliris. The performance of three database storage structures for managing large objects. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD Record, pages 276–285, New York, June 1992. ACM Press.

[4] Alexandros Biliris and Euthimios Panagos. The BeSS object storage manager: Architecture overview. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(3):53–58, September 1996.

[5] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan,

Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 383–394, Minneapolis, June 1994.

[6] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the Exodus extensible database system. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 91–100, Kyoto, August 1986.

[7] Hong-Tai Chou, David J. DeWitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the Wisconsin Storage System. *Software: Practice and Experience*, 15(10):943–962, October 1985.

[8] Roger L. Haskin and Raymond A. Lorie. On extending the functions of a relational database system. In M. Schkolnick, editor, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 207–212, Orlando, June 1982.

[9] Informix Software, Inc. *Extending INFORMIX-Universal Server: Data Types, version* 9.1, 1998. Online version (http://www.informix.com).

[10] Tobin J. Lehman and Patrick J. Gainer. DB2 LOBs: The teenage years. In *Proceedings of the 12th International Conference on Data Engineering*, pages 192–199, New Orleans, February 1996. IEEE Computer Society.

[11] Tobin J. Lehman and Bruce G. Lindsay. The Starburst long field manager. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 375–383, Amsterdam, August 1989. Morgan Kaufmann.

[12] Banu Özden, Rajeev Rastogi, and Abraham Silberschatz. A framework for the storage and retrieval of continuous media data. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. IEEE Computer Society, May 1995.

[13] Jignesh Patel, JieBing Yu, Navin Kabra, Kristin Tufte, Biswadeep Nag, Josef Burger, Nancy Hall, Karthikeyan Ramasamy, Roger Lueder, Curt Ellmann, Jim Kupsch, Shelly Guo, Johan Larson, David DeWitt, and Jeffrey Naughton. Building a scaleable geo-spatial DBMS: Technology, implementation, and evaluation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 336–347, Tucson, June 1997.

[14] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The case for enhanced abstract data types. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 66–75, Athens, 1997.

[15] Bjarne Stroustrup. *The C++ Programming Language, 2nd Ed.* Addison Wesley, Reading, 1991.

[16] Fernando Vélez, Guy Bernard, and Vineeta Darnis. The O2 object manager: an overview. In *Proceedings of the 15th Conference on Very Large Databases*, pages 357–366, Amsterdam, August 1989.