A Tool for Nesting and Clustering Large Objects[*]

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

Abstract

In implementations of non-standard database systems, *large objects* are often embedded within an aggregate of different types, i.e. a *tuple*. For a given *size* and *access probability* of a large object, query performance depends on its representation: either inlined within the aggregate or swapped out to a separate object. Furthermore, the implementation of complex data models often requires *nested* large objects, and access performance is highly influenced by the clustering strategy followed to store the resulting tree of large objects.

In this paper, we describe a large object extension which automatically clusters nested large objects. A *rank function* is developed which indicates the suitability of a large object being inserted into a given cluster. We present two clustering algorithms of different run-time complexity, both using the rank function, and a series of simulations is performed to compare them to each other as well as to two trivial ones. One of the algorithms proves to compute the most efficient clustering in all tests.

1 Introduction

For the support of scientific applications in many cases the relational model is too limited, and more complex data models as well as systems implementing them are needed. Two major research directions to provide these are *extensible* and *object-oriented* DBMSs.

Extensible systems first and foremost support the addition of new data types with corresponding operations ("abstract data types") in the role of attribute types for relations or objects. This includes the support of such types on all levels of the system, for example, in indexing, or in query optimization. For many applications sufficient "modeling power" is gained already by this relatively simple change to the DBMS data model. Note that by the ADT approach the internal structure and complexity of the data types is hidden to the "surrounding" DBMS data model, which may still be as simple as the relational model.

A first class of extensible systems offers a fixed data model which may be relational or object-relational and allows extensions by packages of related abstract data types. For example, Informix Universal Server [Inf98] allows one to add "data blades" and numerous such packages for various purposes (e.g. spatial types, time series, image, ...) are already available.

More ambitious levels of extensibility allow changes to the DBMS object model as such. One stream of research [CDRS86, CDF$^+$94, GM93] has tried to provide toolkits with components such as a storage manager, generic indexes, or a query optimizer generator. Other approaches like Predator [SLR97] or SECONDO [GDF$^+$99, DG99] still offer a kind of "system frame" with well-defined interfaces for extensions which may now be more general than just addition of attribute data types.

The object-oriented approach offers in particular *type constructors* such as *set*, *list*, or *tuple* that can be applied in an orthogonal way, to model *complex objects*. In this case the internal structure of objects is visible to the DBMS.

A common concern in the implementation of data type packages for extensible systems and of object-oriented systems is the need to handle *large objects* requiring many pages of storage. Usually the implementation environments (e.g., storage manager tools), offer a large

[*]This work was partially supported by the CHOROCHRONOS project, funded by the EU under the Training and Mobility of Researchers Programme, Contract No. ERB FMRX-CT96-0056.

object (LOB) abstraction. A common subset of operations provided by all the large object abstractions includes creation, deletion, reading and writing portions of the LOB, and resizing.

A problem is the implementation of data types whose representations may vary in size from very small to very large. Consider, for example, the implementation of a polygon data type represented by a list of vertices. A polygon value could represent the borders of the US state of Colorado (which are exactly a rectangle) or those of Norway (which have thousands of vertices because of the many fjords). Since representations can be large, it is necessary to use the LOB abstraction for implementing them.

On the other hand, these data types are generally not used as stand-alone objects, but rather as components of a comprising structure, e.g. as attribute values in a tuple. Then, if representations are small, it is better to store them "inline" within a storage block representing a tuple. Of course, if representations are large, it is better to store them as separate LOBs, especially if these values are accessed only rarely.

We have presented in [DG98] a tool that allows database implementors to use efficiently large objects even if actual instances can be small. A new abstraction for large objects is offered which automatically switches the representation of large objects from an in-aggregation to a stand-alone one. The decision is based on a threshold size analytically computed from the access probability and the size of the object. Large objects in this abstraction are called *FLOBs* (*Faked Large OBjects*).

However, whereas the tool nicely supports the implementation of "atomic" data types (such as *polygon*), it is not sufficient for implementing *type constructors*. These occur both in the implementation of more complex systems of data types to be provided as extension packages, and in the implementation of object-oriented data models. Consider a data type that arises from the application of a type constructor to some other type (e.g. *list*(*polygon*)). The representation of such a data type needs to organize a set of representations of values of the argument types. If the argument types employ FLOBs themselves, we immediately arrive at a tree of storage blocks which may be small or large, that is, a *FLOB tree*. The generalized problem is then, given the sizes and access probabilities of the storage blocks, to determine an efficient storage layout or *clustering* of the tree, that is, a partitioning into components stored as LOBs.

In this paper, we extend the tool presented in [DG98] and describe a programming interface which offers *nested FLOBs* (or, more precisely "nestable" FLOBs), and so supports a direct and elegant implementation of type constructors. This is illustrated by an advanced application example involving spatio-temporal data types. We provide efficient algorithms for clustering FLOB trees, and simulation experiments to study the quality of the resulting clusterings.

The paper is structured as follows: Section 2 presents the basic concepts of our approach, together with an example application. Two clustering algorithms are presented and analyzed in Section 3. Section 4 demonstrates the efficiency of using nested FLOBs by means of some experiments. Related work is discussed in Section 5. Conclusions are given in Section 6.

2 Nested Faked Large Objects

In this section, we first present an example that allows us to show in more detail the purpose of the tool described in this paper. We then review the concept of FLOBs (*faked large objects*) from [DG98]. This is extended to Nested FLOBs in the last subsection.

2.1 A Motivating Application Example

The main purpose of the tool is the support of the implementation of complex application-specific data type extension packages. Such packages of types and operations (sometimes called "data blades") play an important role in the implementation of database systems for scientific applications.

Our example deals with *spatio-temporal data types* [EGSV99, GBE$^+$98], a collection of types and operations that allow one to represent the temporal development of spatial and other values by data types, and to formulate queries over them by means of the operations. For example, there are data types *moving point* and *moving region.* A moving point describes the time-dependent position of an object in the 2D plane, and a moving region describes the time-dependent position and extent (i.e., regions cannot only move, but also grow or shrink). Conceptually, a value of the *moving point* data type is a function from time into *point* values, and a value of type *moving region* a function from time into *region* values.

The paper [GBE$^+$98] presents a design of types and operations with an emphasis on closure and consistency. This design is at an *abstract level*, which means that temporal data types are indeed viewed as (arbitrary) functions from time into their respective domains; it means more generally that the domains of data types are defined in terms of infinite sets, without considering finite representations. The paper [FGNS99] proceeds to develop a *discrete model* where finite representations for all data types of the abstract model of [GBE$^+$98] are given, and corresponding data structures are defined. We now show a part[1] of the type system of [FGNS99] (see Table 1).

	→ BASE	*int*, *real*, *string*, *bool*
	→ SPATIAL	*point*, *region*
BASE ∪ SPATIAL	→ UNIT	*const*
	→ UNIT	*upoint*, *uregion*
UNIT	→ MAPPING	*mapping*

Table 1: Signature describing the discrete type system

The type system is given in the form of a *signature.* A signature in general has sorts and operations and defines a set of terms. Sorts control the applicability of operations. Here the role of sorts is taken by *kinds*[2] and that of operations by *type constructors.* The terms generated by this signature are the *types* available in this type system. In Table 1, for example BASE and SPATIAL are kinds, *int*, *region*, and *mapping* are type constructors. Some types generated by this signature are *int*, *const*(*region*), *mapping*(*uregion*), or *mapping*(*const*(*region*)).

The types have the following meaning: *int*, *real*, *string*, *bool* are simple basic types, *point* is a point in the plane. Type *region* is a set of polygons with polygonal holes.

Type constructor *mapping* serves to describe temporal values. The idea is to represent the temporal development as a set of disjoint time intervals with associated values; such a pair is called a *unit.* The associated value can either be a constant, or some description of a "simple" function of time.

The *upoint* data type describes linear movement of a point during the unit's time interval. Hence type *mapping*(*upoint*) can represent a moving point. Type *uregion* describes a *region* value whose vertices move linearly during the unit's time interval; hence *mapping*(*uregion*) can represent a moving region.

The *const* type constructor transforms a given argument type (from the kinds BASE or SPATIAL) into a unit type by adding a time interval to it. This allows us to represent temporal values describing stepwise constant functions of time. For example, the type *mapping*(*const*(*int*)) describes the development of an integer value (e.g. a salary) over time,

[1]To simplify, this is deliberately incomplete; so don't worry about missing types.

[2]The term "kind" is used in programming language theory to describe "the type of types"; here it suffices to view a kind as a set of types.

and *mapping*(*const*(*region*)) can represent a region changing only in discrete steps (e.g. the boundary of a parcel).

Since time intervals within a *mapping* are disjoint, they can be totally ordered, and the data structure for *mapping* should basically be an array of units ordered by time intervals. Note, however, that *mapping* has to accomodate units of fixed size (e.g. *upoint*) or widely varying size (e.g. *uregion* or *const*(*region*)). Note also that it should be able to accomodate data types that can be used independently as well (e.g. *region*). In the following subsections we show how varying size data types such as *region* can be stored efficiently using FLOBs, and how type constructors like *mapping* can be implemented in terms of Nested FLOBs.

2.2 Review of the FLOB Concept

As already mentioned in the introduction, a common problem in the implementation of non-standard data types is that the representations of their values may be very large and of possibly unpredictable size, so that values of the same type may also have very small representations. The *region* type is an example for this. Furthermore, such values occur in aggregations such as *objects* or *tuples* in which case they are usually called *attributes*. If attribute values are small, they are better represented inline within the storage block of a tuple representation.

In [DG98] a new large object abstraction called a FLOB is described. It offers an interface similar to those usually offered by large object abstractions (LOBs) provided by storage managers, and it is in fact implemented on top of the large object concept provided by SHORE. However, the FLOB implementation cooperates with a module in charge of constructing and accessing aggregations, called the *tuple manager*. Depending on the size and access probability of a FLOB, the value is stored within the tuple storage block (in memory as well as on disk), or as a separate large object.

In a bit more detail, the "agreement" between the tuple manager and an attribute data type implementation is that the attribute type is represented in a block of storage (called the *root block*) which may contain logical pointers to a fixed number of other storage blocks managed as FLOBs.[3] Pointers to FLOBs are called *FLOB handles*. Figure 1 shows on the left the general storage layout for attribute types, one with zero FLOBs, and another type with three FLOBs. On the right a tuple is shown. Besides some administrative information for managing the tuple, it contains a part for attribute root blocks and then a part for inlined FLOBs.

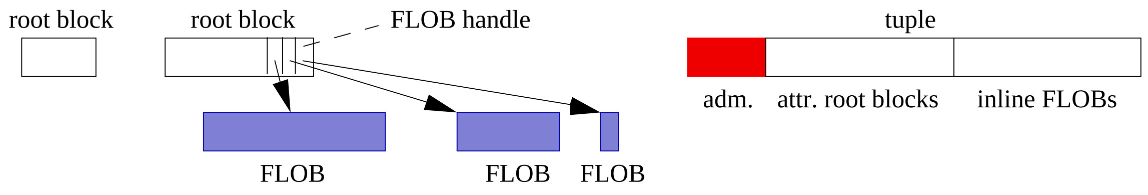


Figure 1: General storage layout of attributes and tuple

Figure 2 shows how this is used for the representation of data types *int* and *region*. The *region* representation contains three FLOBs to represent arrays for vertices, polygons, and holes, respectively. On the right a tuple with an *int* and a *region* attribute is shown; for this particular tuple instance, two of its FLOBs are stored inlined and the third as a separate LOB.

[3]In fact, there exists another abstraction on top of FLOBs called a *database array* which is an extensible array of components of a fixed size. Hence an attribute type can be implemented as a fixed size root block plus a fixed number of variable sized arrays. However, for the discussion in this paper the database array abstraction is not so relevant.

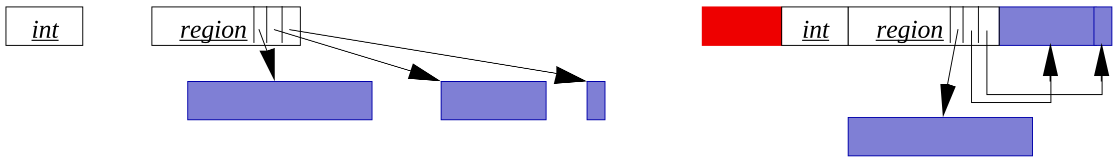


Figure 2: Storage layout for *int* and *region* types and a tuple containing an *int* and a *region* attribute

The implementation of this concept uses three abstractions (i.e. classes, interfaces) called *Tuple*, *Attribute*, and *FLOB* in addition to the data type being implemented. These interact as shown in Figure 3 where a line between classes indicates a "uses" relationship.

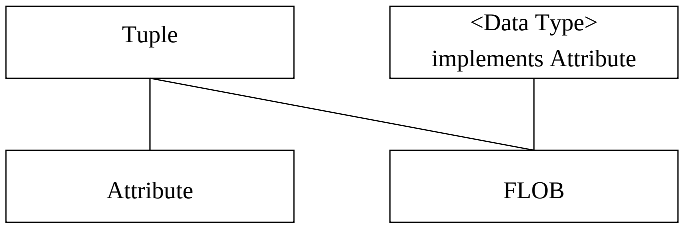


Figure 3: Interaction between classes

Their operations are shown in Table 2. We cannot discuss all operations in detail here, but

Tuple		*Attribute*	*FLOB*	
Create	Save	NumOfFLOBs	Create	Read
Destroy	Get	GetFLOB	Destroy	Write
Open	Put		Open	GetSize
Close	GetID		OpenMemory	Resize
			Close	SetProb
				GetProb

Table 2: Abstractions *Tuple*, *Attribute*, and *FLOB*

only give a brief overview. For a detailed description see [DG98].

The FLOB abstraction manages a large object that is possibly small and in memory. A FLOB can be created and destroyed, opened (i.e., made accessible in memory) and closed, and portions of it can be read and written. It can also be resized and the current size be determined. Finally one can set an access probability which can later be retrieved from it.

A data type implementation may contain a fixed number of FLOB handles. Usually a data type will be implemented as a record with FLOB handles as components. A data type implementation has to implement the *Attribute* interface.

The *Attribute* interface is used by the tuple manager to access FLOBs contained in attribute values. Its operation (method) *NumOfFLOBs (): integer* returns the number of FLOBs used in this data type. Operation *GetFLOB (n: integer): FLOB* returns the FLOB handle of the n-th FLOB in the attribute value.

The *Tuple* abstraction manages the tuple representation. Besides the usual operations for creating and destroying, opening and closing, the essential operations are *Get (n: integer): Attribute* and *Put (n: integer; a: Attribute)* which allow one to get and set the n-th attribute value.

Of particular interest here is the *Save ()* operation. Its general purpose is to commit changes to the in-memory representation of the tuple to the underlying storage manager. It is this operation which computes the storage layout of a tuple. When a tuple is created, and possibly when it is opened (depending on the implementation), a list of attribute type descriptors is passed as an argument to *Create* and *Open*, respectively. From this, *Save* knows the sizes of attribute root blocks and can lay out the first part of the tuple. Through the *Attribute* interface, it can access the FLOB handles contained in attribute values and determine for each FLOB its size and access probability which are the basis for the layout decisions.

2.3 Nested FLOBs

The FLOB concept suffices to implement efficiently storage management for tuples with embedded "atomic" attributes of varying size. In our example, we can efficiently represent the *region* type. But it does not offer enough support for implementing type constructors such as *mapping* which need to embed in their representation values of other data types possibly containing FLOBs.

Consider the two type constructors *const* and *mapping* in our example. Both are applicable to simple fixed size data types as well as to varying size data types containing FLOBs. The only direct and clean implementation one can think of has a storage layout as shown in Figure 4.

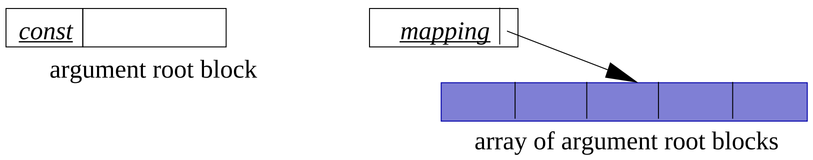


Figure 4: Storage layout for *const* and *mapping* type constructors

If we instantiate the *const* constructor with data types *int* and *region*, we get the storage layouts shown in Figure 5.

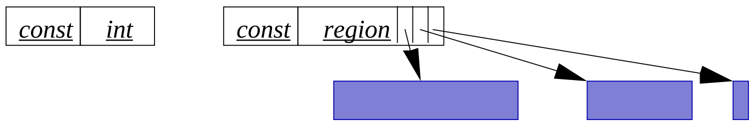


Figure 5: Storage layout for *const*(*int*) and *const*(*region*) types

If we then apply the *mapping* constructor to the types *const*(*int*) and *const*(*region*), we obtain the storage layout shown in Figure 6.

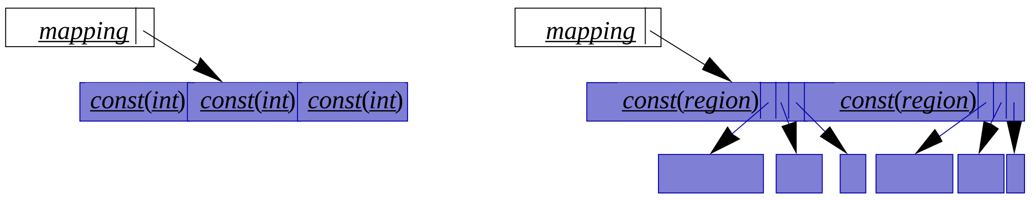


Figure 6: Storage layout for *mapping*(*const*(*int*)) and *mapping*(*const*(*region*)) types

It turns out that the FLOB handle in *mapping* points to a *FLOB tree*, that is, FLOBs contain embedded FLOB handles pointing to other FLOBs. Clearly the generalized storage concept needed for the implementation of non-standard data types including type constructors is that a data type is implemented as an attribute root block containing a fixed number of FLOB handles pointing to FLOB trees.

To implement this, one needs to slightly change the implementation concept of Section 2.2. The first addition is that we must also be able to ask a FLOB for the FLOB handles contained in it. This means that a FLOB also has to implement the *Attribute* interface, which is now renamed more appropriately into *Component*. Hence we obtain relationships between abstractions as shown in Figure 7.

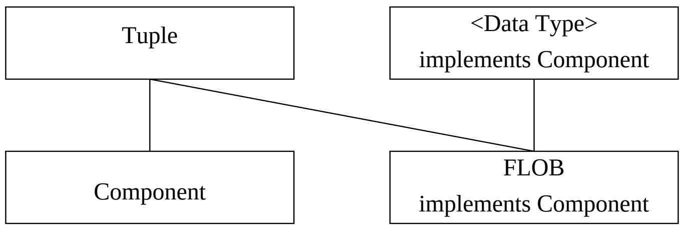


Figure 7: Revised interaction between classes

Second, in contrast to a data type implementation, the number of FLOB handles embedded into a FLOB may vary, and their positions are not fixed. Hence one must be able to explicitly inform a FLOB about a FLOB handle that has been written into it, and the FLOB must store this information somewhere (outside the area that can be read and written by a user, i.e., a client of the FLOB interface). For registering FLOB handles we provide the following interface methods for a FLOB:

RegisterHandlePos (offset: integer);
RemoveHandlePos (offset: integer);

The FLOB uses this information to answer the questions that can be asked via the *Component* interface. The tuple manager can in this way obtain all information that it needs to cluster FLOBs contained in subtrees into storage areas. Such algorithms are developed and analyzed in Section 3. When the tuple is saved, each cluster computed by the algorithms is stored as a LOB of the underlying storage manager. Whenever a tree element, either the tuple or a FLOB, is read, the entire cluster containing this element is restored in main memory.

User-Defined FLOB Groups

The interface for nested FLOBs developed so far allows a database implementor to set access probabilities for FLOBs and so to express how likely it is that a FLOB is accessed given its parent in the tree is read. In this way one can establish a strong or a weak connection between a FLOB node and its father in the FLOB tree.

We introduce one additional feature that appears very useful in the implementation of non-standard data types, and which allows one to express relationships between *siblings* in the FLOB tree. Namely, a database implementor may *know* that several FLOBs, which are children of the same component in the tree, are always (or often) accessed together. Hence one would like to put them together into a *FLOB group*, and so enforce that the clustering algorithm puts them together into a single storage area. Conceptually, in the FLOB tree these sibling nodes are merged into a single node.

To make this possible, we extend the FLOB interface by operations to create and destroy FLOB groups, to register FLOB handle positions with respect to FLOB groups, and to manage probabilities for FLOB groups. The interface is:

CreateGroup (numOfFLOBs: integer; offsets: array of integer; probability: real): integer;
RegisterHandlePos (group, offset: integer);
RemoveHandlePos (group, offset: integer);
SetProb (group: integer; probability: real);
GetProb (group: integer): real;
DestroyGroup (group: integer);

The first operation *CreateGroup* creates a FLOB group and initializes it by registering *numOfFLOBs* FLOBs whose positions within *this* FLOB are passed in the array *offsets*. Also, the *probability* for accessing the FLOB group is passed. The operation returns an identifier for the newly created group. The second and third operation overload those given above by using an additional *group* parameter and so manage these FLOB handles within that group. The fourth and fifth operation set and get probability values for a group (also overloading those for separate FLOBs). The last operation removes a group and "unregisters" all the FLOB handles in it.

The *Component* interface is also changed a bit to make the tuple manager aware of FLOB groups:

NumOfFLOBGroups (): integer
NumOfFLOBsInGroup (group: integer): integer;
GetFLOB (FLOBNumber, group: integer): FLOB;

So one can ask for the number of FLOB groups in this component and then for the number of FLOBs within each FLOB group. Finally, one can get FLOB handle *FLOBNumber* within group *group*.

To simplify this interface, all the "non-grouped" FLOBs in this component (that have been registered by *RegisterHandlePos* with only a single argument), are viewed as belonging to a FLOB group 0. Hence the tuple manager will consider FLOBs in group 0 not as a user-defined FLOB group, but as separate FLOBs.

Note that it is technically still possible to get a FLOB handle belonging to a group and to set its access probability individually; however, this will be ignored in clustering by the tuple manager and hence has no effect.

In summary, the interface of abstractions *Component* and *FLOB* now looks as shown Table 3. The operations for *Tuple* are the same as before.

Component	*FLOB*	
NumOfFLOBGroups	Create	SetProb
NumOfFLOBsInGroup	Destroy	GetProb
GetFLOB	Open	RegisterHandlePos
	OpenMemory	RemoveHandlePos
	Close	CreateGroup
	Read	DestroyGroup
	Write	
	GetSize	
	Resize	

Table 3: Revised abstractions *Component*, and *FLOB*

3 Analysis and Algorithms

The aim of this section is to develop algorithms which determine a good clustering for a nested FLOB structure by means of the given information: the size of each FLOB, the structure of references, basically the *parent* of each FLOB, and for each FLOB the conditional probability that it is read given its parent is read. Section 3.1 formally defines nested FLOB structures and analyzes the access probabilities of blocks consisting of several FLOBs. In Section 3.2 we introduce an appropriate cost model for read access to large objects, and Section 3.3 presents two resulting algorithms able to find good clusterings.

3.1 Basic Data Structures

3.1.1 FLOB Trees

A straightforward representation of nested FLOBs is a *FLOB tree*. The nodes of a FLOB tree correspond to FLOBs. FLOB f' is a son of FLOB f iff the value of f contains a handle to f'. The edges in a FLOB tree are labeled with the conditional probability p that a son is read given its father is read. Throughout the paper we assume $0 < p \leq 1$. Excluding $p = 0$ does not compromise the generality of our analysis since storing an object which is never read is not sensible.

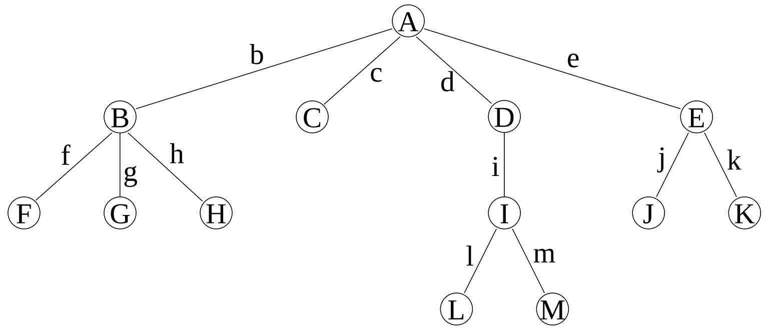


Figure 8: Sample FLOB tree

We will use the following operations on FLOB trees:

Operation	Semantics
Sons(f)	The set of all sons of node f
Subtree(f)	The set of all nodes in the subtree rooted by f, excluding f
$S(f)$	The size of the FLOB value of f in disk pages
$P(e)$	The label of edge e
$P(f)$	1, if node f is the root of the tree, otherwise the label of the edge from f's father to f
Nodes(f, f')	The set of all nodes in the path from f to f', excluding f
$P(f', f)$	The conditional probability of reading f' given f is read: $\prod_{\overline{f} \in Nodes(f,f')} P\left(\overline{f}\right)$

3.1.2 FLOB Tree Partitions

A *FLOB tree partition* is a 4-tuple $(V, E, r, \mathcal{P})$ with V the set of nodes in the tree, $E \subseteq V \times V$ the set of edges in the tree, $r \in V$ the root, and $\mathcal{P}$ a *partition* of V, i.e. the elements of $\mathcal{P}$, called *blocks*, are disjoint subsets of V, and $\bigcup_{B \in \mathcal{P}} B = V$.

Figure 9 shows one partitioned version of the sample FLOB tree in Figure 8. Each of the blocks α to ϵ corresponds to a cluster of FLOBs.

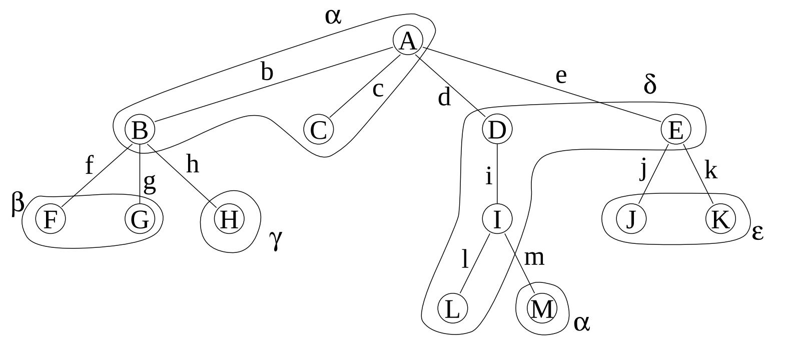


Figure 9: Sample FLOB tree partition

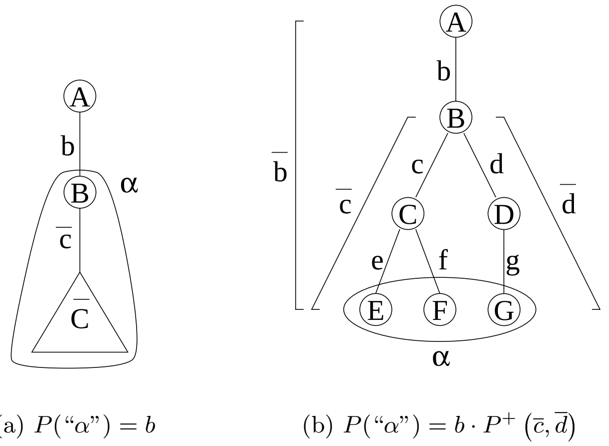
With FLOB tree partitions, additional operations will prove useful later on:

Operation	Semantics
$Roots(B)$	The set of all roots of subtrees in B: $\{f \in B \mid \nexists f' \in B : f \in Subtree(f')\}$
$S(B)$	The sum of the FLOB value sizes in B: $\sum_{f \in B} S(f)$
$Nodes^*(f, B)$	The set of all nodes in the paths from f to any FLOB in B, excluding f
$P(B, f)$	The probability that block B is read given FLOB f is read.

$P(B, f)$ can be computed as follows. Let "f" and "B" denote the *events* that "FLOB f is read" and "block B is read", respectively. Block B is read if at least one of the FLOBs in B is read:

$$P(B, f) = P(\text{“}B\text{”} | \text{“}f\text{”}) = P\left(\left(\bigcup_{f' \in B} \text{“}f'\text{”}\right) \mid \text{“}f\text{”}\right)$$

The computation of this probability exploits two basic observations:



(a) $P(\text{“}\alpha\text{”}) = b$ (b) $P(\text{“}\alpha\text{”}) = b \cdot P^+\left(\overline{c}, \overline{d}\right)$

Figure 10: Clippings from FLOB tree partitions

1. Only the FLOBs in $Roots(B)$ contribute to $P\left(\bigcup_{f' \in B} \text{“}f'\text{”}\right)$. We show this for a block containing one root, depicted in Figure 10(a). Label $\overline{c}$ is the probability that any FLOB

in subtree $\overline{C}$ is accessed. Given FLOB A is read, the probability that block α is read is the probability $P(\text{“}B\text{”} \cup \text{“}\overline{C}\text{”})$ that at least one FLOB in α is read. From probability theory we know that

$$P(E_1 \cup E_2) = P(E_1) + P(E_2) - P(E_1 \cap E_2) \tag{1}$$

for any two events E_1 and E_2. Thus the probability to access α is

$$P(\text{“}\alpha\text{”}) = P(\text{“}B\text{”} \cup \text{“}\overline{C}\text{”}) = P(\text{“}B\text{”}) + P(\text{“}\overline{C}\text{”}) - P(\text{“}B\text{”} \cap \text{“}\overline{C}\text{”}) = b + b\overline{c} - b\overline{c} = b \quad \square$$

The extension to blocks rooted by multiple roots is easy, considering a multi-root block as a union of single-root blocks.

2. From probability theory we know that the function P^+ computes the probability $P(E_1 \cup \cdots \cup E_m)$ from the probabilities $e_1, \ldots, e_m$ for the *independent* events E_1 to E_m:

$$\begin{aligned} P^+(e_1, \ldots, e_m) = &\sum_{i=1}^{m} e_i \; - \sum_{i=1}^{m-1} \sum_{j>i}^{m} e_i e_j \; + \sum_{i=1}^{m-2} \sum_{j>i}^{m-1} \sum_{k>j}^{m} e_i e_j e_k \\ &- \; \cdots \; + \; \cdots \; - \; (-1)^m \, e_1 e_2 \cdots e_m \end{aligned} \tag{2}$$

For events E_1 to E_m which are not independent, but rather have a common base event $E \notin \{E_1, \ldots E_m\}$ such that, once E happens, probabilities $P(E_1|E), \ldots, P(E_m|E)$ are independent, we use (2) to compute

$$\begin{aligned} P(E_1 \cup \cdots \cup E_m) &= P(E) \cdot P(E_1 \cup \cdots \cup E_m|E) \\ &= P(E) \cdot P^+(P(E_1|E), \ldots, P(E_m|E)). \end{aligned} \tag{3}$$

Figure 10(b) illustrates how (3) contributes to determining the probability $P(B, f)$ for reading a block given some higher-level FLOB is read. Let $\overline{b}$ be the probability that at least one of those nodes in α, which can be reached through edge b, is read given FLOB A is read. Similarly, $\overline{c}$ and $\overline{d}$ are the probabilities that at least one of those nodes in α are read which can be reached through edge c and d, respectively, given FLOB B is read. Then due to (3) we know that

$$P(\alpha, A) = \overline{b} = P((\text{“}E\text{”} \cup \text{“}F\text{”}) \cup \text{“}G\text{”}) = b \cdot P^+(\overline{c} \cdot \overline{d}).$$

Resuming, we give the following recursive definition of $P(B, f)$:

$$P(B, f) = \begin{cases} 1 & \text{if } f \in B, \\ P^+(P(f_1) \cdot P(B, f_1), \ldots, P(f_k) \cdot P(B, f_k)) & \text{otherwise,} \end{cases} \tag{4}$$

with $\{f_1, \ldots, f_k\} = \mathit{Sons}(f) \cap \mathit{Nodes}^*(f, B)$.

3.2 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. The solid line in Figure 11 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

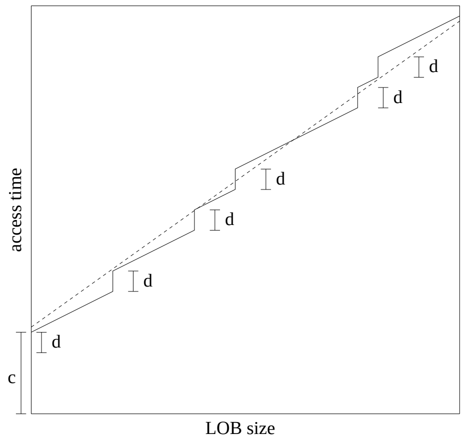


Figure 11: LOB access time, qualitative

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 we use a linear cost model, assuming the costs for reading a LOB is sufficiently well computed by

$$C_S(n) = c + g \cdot n \tag{5}$$

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.

By means of (4) and (5) we can now define the expected costs C_T for reading a FLOB tree partition as the costs for reading the cluster containing the root of the tree plus the expected costs for reading the other clusters in the tree:

$$C_T(V, E, r, \mathcal{P}) = \sum_{B \in \mathcal{P}} P(B, r) \cdot C_S(S(B)). \tag{6}$$

Applying this formula to the FLOB tree partition in Figure 9, C_T computes to

$$1 \cdot C_S(S(\alpha)) \; + \; b \cdot P^+(f, g) \cdot C_S(S(\beta)) \; + \; b \cdot h \cdot C_S(S(\gamma)) \; + \; b \cdot P^+(d, e) \cdot C_S(S(\delta)) \\ + \, e \cdot P^+(j, k) \cdot C_S(S(\epsilon)).$$

Here we already have taken into account that calling P^+ with a single argument returns the value of the argument: $P^+(x) = x$.

It is the task of the tool described in this paper to find a good partition for a given FLOB tree. Exhaustive search finds the optimal partition, but cannot be used due to the high number of partitions of a given set of n elements, known to be the n-th *Bell number* [GKP94].[4] Instead, in Section 3.3 we present algorithms running in $O(n)$ and $O(n \cdot h)$ (n: number of nodes, h: height of the tree) finding reasonably good solutions.

[4]Already the 12th Bell number is beyond 4 million.

3.3 Algorithms

3.3.1 Rank Functions

In [DG98] we determined the expected costs to read a tuple with embedded FLOBs as follows. A tuple is a pair (t, F), consisting of a core tuple t and a set $F = \{f_1, f_2, \ldots, f_n\}$ 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:

$$C_T(t, F) = C_S(S(t)) + \sum_{f \in F} \begin{cases} r_f \cdot C_S(S(f)) + (1 - r_f) \cdot 0 & \text{if } f \text{ is LOB,} \\ r_f \cdot g \cdot S(f) \quad + (1 - r_f) \cdot g \cdot S(f) & \text{otherwise.} \end{cases} \tag{7}$$

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

$$C_T(t, F) = C_S(1) + \sum_{f \in F} \begin{cases} r_f \cdot C_S(S(f)) & \text{if } f \text{ is LOB,} \\ g \cdot S(f) & \text{otherwise.} \end{cases} \tag{8}$$

Obviously, Equation (8) 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, which is the case if

$$r_f \cdot C_S(S(f)) < g \cdot S(f) \tag{9}$$

holds for those FLOBs $f \in F$ that are swapped out to a LOB and does not hold for those FLOBs that are stored within the tuple byte string.

In the more complex situations arising from the possibility to nest FLOBs addressed in this paper, not only a tuple access, but also a FLOB access can be the base event upon which other FLOBs are read. Furthermore, not only those FLOBs referenced directly by the base objects, but all FLOBs in the subtree rooted by the base object should be considered for possible inlining with the base object. So the version of (9) regarding our framework for nested FLOBs is given by

$$P(f', f) \cdot C_S(S(f')) < g \cdot S(f') \quad \forall \quad f' \in \mathit{Subtree}(f). \tag{10}$$

The left-hand side computes the expected costs for reading a FLOB f' which is stored in an external LOB given f has been read. The right-hand side shows the costs for reading f' if it is stored together with its base object f.

Another representation of (10) is

$$\left(\frac{1}{P(f', f)} - 1\right) \cdot S(f') > \frac{c}{g} \quad \forall \quad f' \in \mathit{Subtree}(f). \tag{11}$$

The ratio $\frac{c}{g}$ is a system-dependent constant which can be determined easily by running a test program reading large objects of different sizes, as described in Section 4.

Based on (11) we define the *rank* function R as follows:

$$R(f, f') = \frac{\left(\frac{1}{P(f', f)} - 1\right) \cdot S(f')}{\frac{c}{g}} \tag{12}$$

The greater the value of $R(f, f')$, the higher is the rank of inserting f' in a block different from the one rooted by f. If $R(f, f') < 1$, it is more efficient to store f' together with f in a common cluster.

For convenience, we will also use another version of R, taking as parameter the *conditional probability* that FLOB node f' is accessed, rather than the source node:

$$\tilde{R}(f', p) = \frac{\left(\frac{1}{p} - 1\right) \cdot S(f')}{\frac{c}{g}} \tag{13}$$

3.3.2 An $O(n)$ Algorithm

Algorithm 1 Find a FLOB tree partition in $O(n)$ time

algorithm Partition1(f, p, *isroot*)

input: f is the root of a FLOB subtree, p is the conditional probability that f is accessed given the root of f's block is accessed, and *isroot* is a flag indicating whether f is the root of a block and thus the block can be output after all other block members have been found. To find a partition for a given FLOB tree rooted by r we call Partition1(r, 1, true).

output: A block of FLOB nodes containing f is returned, possibly to be merged with the father's block. Complete blocks are output. After termination, the set of all output blocks is a partition of the input tree.

method:

```
  result ← {f}
  for all $f' \in Sons(f)$ do
    if $\tilde{R}(f', p \cdot P(f')) \geq 1$ then
      Partition1($f'$, 1, true) {ignore return values}
    else
      result ← result ∪ Partition1($f'$, $p \cdot P(f')$, false)
    end if
  end for
  if isroot then output result end if
  return result
end Partition1
```

Algorithm 1 finds a partition of the set of FLOBs in a FLOB tree such that it is efficient to store each block in a single FLOB cluster, using the rank function R in a top-down traversal. Essentially, it checks for each son of a node whether it should be included in the node's block. If this is not the case, the son becomes the root of a new block.

Algorithm 1 visits each node exactly once, thus running in $O(n)$. We pay for such a good run-time complexity by considering only a small subset of all possible partitions. To be specific, only partitions will be found whose blocks contain single, complete subtrees.

Due to the limited search space, Algorithm 1 cannot find blocks like α in Figure 9, because the path connecting node A and M leaves α. Furthermore, blocks like β, δ, and ϵ are not found; they consist of more than a single subtree.

3.3.3 An $O(n \cdot h)$ Algorithm

Algorithm 2 searches a larger search space. It processes all FLOBs in a tree in a preorder traversal by computing for all nodes f and all nodes $f' \in Subtree(f)$ whether f' should be

Algorithm 2 Find a FLOB tree partition in $O(n \cdot h)$ time

```
algorithm Partition2($f$)
input: $f$ is the root of a FLOB tree all of whose nodes are unmarked.
output: Complete blocks are output. After termination, the set of all output blocks is a
        partition of the input tree.
method:
  if $f$ is not yet marked then
    $result \leftarrow \{f\}$
    mark $f$
  else
    $result \leftarrow \emptyset$
  end if
  for all $f' \in Subtree(f)$ do
    if $f'$ is not yet marked and $R(f, f') < 1$ then
      $result \leftarrow result \cup f'$
      mark $f'$
    end if
  end for
  if $result \neq \emptyset$ then output $result$ end if
  for all $f' \in Sons(f)$ do Partition2($f'$) end for
end Partition2
```

inserted in the block rooted by f.

The run-time costs of this algorithm applied to the root of a FLOB tree are given by

$$C_{\text{Partition2}}(f) = a + b \cdot |Subtree(f)| + \sum_{f' \in Sons(f)} C_{\text{Partition2}}(f'). \tag{14}$$

Here a is some initial cost for a single call of Partition2, and b is the cost for checking for a single node in the subtree whether it should be included in the root block or not. To rewrite (14) as a function of n, the number of nodes in the FLOB tree, and h, the height of the tree, we consider the arising costs for processing a FLOB tree level-wise. For every node in the FLOB tree cost a has to be added to the entire costs, since each node of the tree is visited exactly once. Furthermore, for each level l containing nodes f_1 to f_k, the costs $b \cdot (|Subtree(f_1)| + \cdots + |Subtree(f_k)|)$ must be taken into account. Since all subtrees of nodes of the same level are disjoint we know that $|Subtree(f_1)| + \cdots + |Subtree(f_k)| < n$, thus giving rise to:

$$C_{\text{Partition2}}(f) < n \cdot a + \sum_{l=0}^{h} b \cdot n = n \cdot (a + (h+1) \cdot b). \tag{15}$$

So the run-time complexity of Partition2 is in $O(n \cdot h)$.

Notice that block δ in Figure 9 cannot be found by Algorithm 2, either: Starting with FLOB A, block α is built. In the following steps FLOBs D and E become *roots* of distinct blocks and thus cannot belong to the same block.

3.3.4 Extensions

The run-time complexity of Algorithm 2 can be decreased easily by setting a *minimal size* parameter, indicating the expected minimal size of FLOB values. This parameter can be used to limit the length of paths to be considered. With increasing path length, the probability of accessing a FLOB reached by the path decreases exponentially. Figure 12 shows for different values of the system-dependent constant $\frac{c}{g}$ the relationship between access probability and threshold size. For instance, for a $\frac{c}{g}$-value of 0.5 and a minimal FLOB value size of 16 words of storage (64 bytes), we need not consider paths with an edge label product $> \left(\frac{1}{2}\right)^6 = \left(\frac{1}{4}\right)^3 = \left(\frac{1}{8}\right)^2$, because the FLOBs reached by such paths are not small enough to be included in the root block.

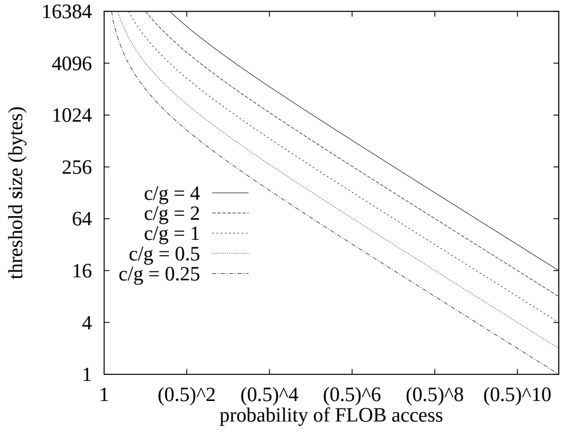


Figure 12: Threshold size

4 Simulations

In this section we present some of the simulations we performed to evaluate the quality of the algorithms presented in Section 3.3. As a first step, we determined the system-dependent constants required to compute the rank function as well as the cost for reading a LOB on a real system; this is described in Section 4.1. The actual simulation results are then presented in Section 4.2.

4.1 System-Dependent Constants

To determine reasonable values for the system-dependent constants c and g, we performed access-time measurements on a SUN Ultra-1 workstation with 128 MB main memory, Solaris 2.6, and SHORE 2.0. The SHORE buffer pool was set to 1 MB.

We subsequently created LOBs (SHORE *records*) of size $< 1, 2, \ldots, 10 >$ disk pages and repeated 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 13 illustrates the results. The solid line depicts the dependency between the size of objects to be read and 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.[5]

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 approximately

[5]This coarse time granularity was the reason to repeat all object retrievals 20 times; otherwise the access of all objects with size ≤ 18 would have been reported to be 1.

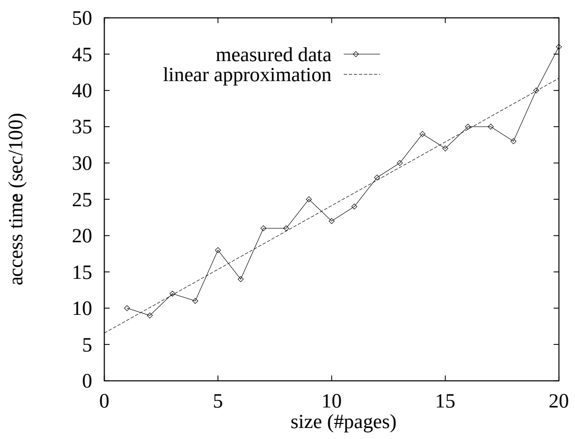


Figure 13: LOB access time

defined by $y = 6.57 + 1.75x$. Thus we conclude the system-dependent constants c, g, and $\frac{c}{g}$ to be $\frac{6.57}{20} \approx 0.329$, $\frac{1.75}{20} \approx 0.088$, and $\frac{6.57}{1.75} \approx 3.75$, respectively.

4.2 Results

Then, we compared four different algorithms as follows. Alg. 1 is the $O(n)$ algorithm proposed in Section 3.3, clustering entire subtrees. Its $O(n \cdot h)$ extension is referred to as Alg. 2. In addition to that, we also used two trivial algorithms: Alg. 3 stores each FLOB in a LOB on its own, and Alg. 4 stores a complete tree in a common cluster in depth-first order. In a single simulation run, we create a FLOB tree, apply each of the four algorithms to the tree, and compute the expected costs for accessing the resulting tree partitions according to Equation 6.

In a first series of simulation runs, we created complete trees with constant FLOB size, access probability, height, and degree. We used tall (degree = 2, height = 12) and wide (degree = 20, height = 3) trees, small (64 Bytes) and large (10 pages = 81200 Bytes) FLOBs, and low (0.05) and high (0.8) conditional access probabilities. Table 4 shows the expected costs for accessing the tree partitions produced by the different algorithms.

Tree Shape	FLOB Size	Access Probability	Access Costs (sec/100)			
			Alg. 1	Alg. 2	Alg. 3	Alg. 4
tall	small	low	0.34	0.34	0.37	5.99
tall	small	high	6.00	6.00	246.69	6.00
tall	large	low	0.47	0.47	0.47	6.98
tall	large	high	870.99	935.11	902.44	1291.96
wide	small	low	0.95	0.91	1.32	6.16
wide	small	high	6.16	6.16	1439.05	6.49
wide	large	low	3.95	3.95	3.95	28.66
wide	large	high	4819.16	4880.58	5268.70	5341.73

Table 4: Access costs for complete trees

In general, the partitions produced by Algs. 1 and 2 are of almost the same high quality. In each simulation, either Alg. 3 computes a partition of similar quality, while Alg. 4 performs much worse, or vice versa. Neither trivial algorithm produces a good result in all simulations. Thus, Alg. 1 or Alg. 2 are the best choice here.

There is no qualitative difference in the results obtained by partitioning tall or wide trees. To further explore the impact of the tree structure on algorithm performance, in a second simulation series we repeated the first one, now creating sparse trees instead of complete ones. We omit the results here, since they do not show any qualitative difference to Table 4. Apparently the tree structure does not influence the choice of the best algorithm.

To see the advantage of Alg. 2, we again created complete trees (degree: 4, height: 6). For each node, however, access probability and size were set randomly. The access probabilities were uniformly distributed in $]0, 1]$. For $\frac{1}{10}$ of the node we set a large size, for $\frac{9}{10}$ a small one. Basically, small sizes were normally $(64, \frac{64}{2})$ distributed, large sizes were normally $(81200, \frac{81200}{2})$ distributed. In the rare case of a size < 1, a new random size was generated instead. The average costs after ten experiments were as follows:

Alg. 1	Alg. 2	Alg. 3	Alg. 4
16.95	17.53	52.36	117.90

While for each algorithm the result values showed a notable variance in this ten experiments — the highest expected cost differed by almost factor 4 from the lowest one — the ratios of expected costs for different algorithms were very stable. In particular, in all experiments Alg. 2 produced a partition slightly *worse* than the one returned by Alg. 1! For an explanation, let us consider the main difference between Alg. 1 and Alg. 2. Following a path towards a leaf, whenever Alg. 1 encounters a node which should not be included into the block B of its father, no other node in that path will be inserted into B later. In contrast to that, Alg. 2 is able to "jump over" nodes in a path. Apparently, in many cases this presumed advantage turns out to be a disadvantage, since deeper nodes are assigned to the blocks of higher nodes in a too greedy manner. Consider a little example tree with root a, b a son of a, and c a son of b. Let a and b be assigned to different blocks. Then Alg. 2 essentially decides whether to insert c into the block rooted by a or not by comparing the expected access cost of a block $\{a, c\}$ and the cost of storing c in a block on its own; the possibly optimal alternative with a block $\{b, c\}$ is ignored.

Finally, we simulated reading tuples containing a $\underline{mapping}(\underline{const}(\underline{region}))$ value, as described in Section 2. Remember that a $\underline{mapping}$ value is represented using a FLOB, implementing a variable-sized array of *units*. A unit might be represented as a time interval (8 Bytes), a set of flags (8 Bytes), and a FLOB handle (20 Bytes) referencing a $\underline{region}$ value. In our simulation, a $\underline{region}$ value consists of segments, each of them represented as a pair of half-segments. Thus we assume the representation of a segment to require $2 \cdot (4 \cdot 4) = 32$ Bytes.

We created four different trees representing a tuple with an embedded value of type $\underline{mapping}(\underline{const}(\underline{region}))$. We varied the number of units (10 or 100) and the number of segments in a region (30 or 300). The access probabilities of the $\underline{region}$ values were set to those corresponding to a binary search on the mapping: $< \ldots, \frac{1}{2}, \ldots, 1, \ldots, \frac{1}{2}, \ldots >$. The resulting access costs are the left-hand values in Table 5. Again Alg. 1 proves its superiority.

To test the impact of probability settings, we also computed the expected costs for binary search on partitions resulting from setting all region access probabilities to 0.5. They are printed behind the original costs for each tree and Algs. 1 and 2 in Table 5. For small numbers of units with only small regions, the expected costs are the same, because a single cluster is created anyway. Obviously, with increasing number of units and growing size of regions, setting probabilities correctly becomes more and more important, because with constant probabilities Algs. 1 and 2 cluster too many regions with the FLOB representing the array of units.

After these simulations, Alg. 1 is clearly the algorithm of choice, since it produces the best partitions for all the tested cases, while there is no faster alternative. Another result is that

Units	Segments / region	Access Costs (sec/100)			
		Alg. 1	Alg. 2	Alg. 3	Alg. 4
10	30	0.44 / 0.44	0.44 / 0.44	1.07	0.44
10	300	0.95 / 1.38	0.98 / 1.38	1.23	1.38
100	30	0.96 / 1.41	1.00 / 1.41	1.63	1.41
100	300	1.67 / 10.75	1.71 / 10.75	1.94	10.75

Table 5: Access costs for *mapping* trees with correct/incorrect probability settings

access performance is sensitive to strongly incorrect settings of a FLOB's access probability. It might be necessary to analyze the access patterns of the algorithms invoked by user queries, maintain access statistics, initially run the system in training mode, etc., to find appropriate probability settings.

5 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 throughput for sequential access, etc.

Within all those large object abstractions, there is not paid any attention to the storage environment of large objects. In toolkits like SHORE and object-relational systems like Informix Universal Server, the DBI has direct access to the large object interface and is able (and obliged) to determine the way large objects are used for value representation. Thus, the responsibility for *efficient use* of large objects is simply passed to the DBI. 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 [PYK$^+$97]. 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* [Inf98]. 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 work described in this paper started with [DG98]. Here the concepts and a sample C^{++} implementation of a tool managing tuples containing FLOBs is presented, automatically switching the representation of embedded large objects according to the object's size and a threshold size determined analytically. It turned out, however, that the single-level FLOB hierarchy provided by [DG98] is not sufficient to support the implementation of data models allowing nested or even recursive definitions of variable-sized types. These can be found in application-specific non-standard data models as well as in more general models like the object-oriented model.

As a consequence, clustering of nested structures has been a research issue since the first variants of object-oriented systems have been implemented. For implementations of the hierarchical model, Schkolnick [Sch77] presents an $O(n)$ algorithm computing a clustering for a given *type tree*. This tree is essentially constructed using the schema information given by the user. The algorithm returns the optimal clustering with respect to a predefined access pattern and the underlying cost model based on page faults.

Chang and Katz [CK89] also use "knowledge of structural relationships and inheritance", i.e., schema information, for clustering objects. Furthermore, statistics about access frequencies in a real-world CAD application and additional user hints for clustering objects are considered. Their clustering algorithm essentially chooses a placement for each newly created instance based on the static information about the instance and its access frequencies. With changing information, reclustering is performed at run-time.

In ORION, usually all instances of a class are stored in a common segment [KBC$^+$87, KBG89]. A set of objects related by ORION's IS-PART-OF relationship is called a *composite object*; the user may issue "Cluster" messages to determine a good clustering for composite objects. Furthermore, the clustering strategy takes into account versioning information.

Cheng and Hurson [CH91] propose to store entire complex objects in *chunks* of contiguous pages. Creation of new objects possibly triggers reclustering actions. The object order in a chunk is computed by a *level clustering* algorithm, using access frequency statistics or user hints, similar to Kruskal's algorithm for constructing a minimum cost spanning tree.

The clustering approach in O_2 uses *placement trees*, indicating related objects, and formal descriptions of access methods' characteristics given by the database administrator [BDH92]. A cost model essentially considering page-faults and main-memory overload is the basis for dynamically adapting the initial clustering.

Other research on clustering in object-oriented database systems explores more precise cost models for specific environments, e.g., Markov chains to analyze caching in a client-server environment with objects smaller than a disk page [TN91, TN92], alternative search strategies like simulated annealing [HLL94], or specific applications, for instance knowledge-based systems [RG96].

All the work on clustering mentioned above either exploits schema information, or directly controls characteristics of the underlying storage manager like buffer allocation and replacement policies, or both. Firstly, however, with database toolkits like Exodus [CDRS86] or SHORE [CDF$^+$94], or in generic database development environments like Volcano [GM93], Predator [SLR97], or SECONDO [GDF$^+$99, DG99], there is no prescribed data model; thus we cannot assume any schema information. Secondly, from a software engineering point of view it was preferable to provide a general interface enabling us to use different storage managers rather than to implement one from scratch, since there is already a great variety of up-to-date storage managers which have proven to be reliable and efficient.

The generality of our approach is mainly founded by the cost model. Most other clustering strategies either do not set up a cost model at all, heuristically exploiting schema information or statistics, or use cost models which restrict the maximum size of objects to values less than a disk page. The cost metric usually is the number of page faults, ignoring the benefits of accessing data located on contiguous disk pages. In contrast to that, our cost model takes into account that reading a segment of contiguous pages is superior to reading scattered pages. As a consequence, our cost model does not only consider page faults, but rather different costs for accessing the first page of a segment and its subsequent pages, and is able to process arbitrary object sizes.

Finally, we set off the lack of impact on the buffer management strategy of the underlying storage manager at least partially by reading entire clusters, as described in Section 2.3, thereby enforcing preemptive caching in the application's memory space.

6 Conclusions

We have presented in this paper a large object extension that automatically clusters trees of nested objects. A sound and general interface was described, and two different clustering techniques using a cost model based on access probabilities and object sizes were given. In a series of simulations, we analyzed how the algorithms compare to each other and to two trivial ones. The results showed that they improve efficiency and that one outperforms the other in all situations.

The main contribution of our work is that we assume only few basic features on the underlying storage manager, resulting in a generic tool that can be used with many different storage managers and in many different environments. In addition to that, the behaviour of the clustering algorithm can be influenced by means of hints and access probabilities, but even without this information, using our tool increases performance considerably. An important difference to other approaches is that no schema information is used in the algorithms, hence they do not depend on a particular data model.

We expect this work to be very useful in scientific applications allowing for the definition of complex data types. So far, large objects are considered untyped byte strings. We plan to provide higher abstraction levels like arrays, trees, lists, or graphs on top, which will benefit from the characteristics of the tool.

The implementation of the advanced spatio-temporal data model sketched in Section 2 using our tool is ongoing work.

References

[BDH92] V. Benzaken, C. Delobel, and G. Harrus. Clustering Strategies in O_2: An Overview. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System. The Story of O_2*, chapter 17, pp. 385–410. Morgan Kaufmann, 1992.

[CDF$^+$94] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, and M.J. Zwilling. Shoring Up Persistent Applications. In *Proc. ACM SIGMOD Conference*, pp. 383–394, Minneapolis, 1994.

[CDRS86] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and File Management in the Exodus Extensible Database System. In *Proc. of the 12th VLDB Conference*, pp. 91–100, Kyoto, 1986.

[CH91] J.R. Cheng and A.R. Hurson. Effective Clustering of Complex Objects in Object-Oriented Databases. In *Proc. ACM SIGMOD Conference*, pp. 22–31, Denver, 1991.

[CK89] E.E. Chang and R.H. Katz. Exploiting Inheritance and Structural Semantics for Effective Clustering and Nuffering in an Object-Oriented DBMS. In *Proc. ACM SIGMOD Conference*, pp. 348–357, Portland, 1989.

[DG98] S. Dieker and R.H. Güting. Efficient Handling of Tuples with Embedded Large Objects. Technical Report Informatik 236 (http://www.fernuni-hagen.de/inf/pi4/papers/FLOBs.pdf), FernUniversität Hagen, 1998. To appear in *Data & Knowledge Engineering*, 32(3):247–269, March 2000.

[DG99] S. Dieker and R.H. Güting. Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. Technical Report Informatik 249, FernUniversität Hagen, Praktische Informatik IV, 1999.

[EGSV99] M. Erwig, R.H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3):265–291, 1999.

[FGNS99] L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. Technical Report Informatik 260, FernUniversität Hagen, 1999. To appear in *Proc. ACM SIGMOD Conference 2000.*

[GBE$^+$98] R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. Technical Report Informatik 238, FernUniversität Hagen, 1998. To appear in *ACM Transactions on Database Systems.*

[GDF$^+$99] R.H. Güting, S. Dieker, C. Freundorfer, L. Becker, and H. Schenk. SECONDO/QP: Implementation of a Generic Query Processor. In *Proc. of the 10th Intl. Conf. on Database and Expert Systems Applications (DEXA'99)*, pp. 66–87, Florence, 1999.

[GKP94] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd ed.* Addison-Wesley, 1994.

[GM93] G. Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. of the 9th IEEE Intl. Conf. on Data Engineering*, pp. 209–218, Vienna, 1993.

[HLL94] K.A. Hua, S.D. Lang, and W.K. Lee. A Decomposition-Based Simulated Annealing Technique for Data Clustering. In *Proc. of the Thirteenth ACM Symposium on Principles of Database Systems (PODS '94)*, pp. 117–128, Minneapolis, 1994.

[Inf98] Informix Software, Inc. *Extending INFORMIX-Universal Server: Data Types, version* 9.1, 1998. Online version (http://www.informix.com).

[KBC$^+$87] W. Kim, J. Banerjee, H.T. Chou, J.F. Garza, and D. Woelk. Composite Object Support in an Object-Oriented Database System. *ACM SIGPLAN Notices*, 22(12):118–125, 1987.

[KBG89] W. Kim, E. Bertino, and J.F. Garza. Composite Objects Revisited. In *Proc. ACM SIGMOD Conference*, pp. 337–347, Portland, 1989.

[PYK$^+$97] J. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellmann, J. Kupsch, S. Guo, J. Larson, D. DeWitt, and J. Naughton. Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. In *Proc. ACM SIGMOD Conference*, pp. 336–347, Tucson, 1997.

[RG96] A. Ramanujapuram and J.E. Greer. A Hybrid Object Clustering Strategy for Large Knowledge-Based Systems. In *Proc. of the 12th IEEE Intl. Conf. on Data Engineering*, pp. 247–257, New Orleans, 1996.

[Sch77] M. Schkolnick. A Clustering Algorithm for Hierarchical Structures. *ACM Transactions on Database Systems*, 2(1):27–44, 1977.

[SLR97] P. Seshadri, M. Livny, and R. Ramakrishnan. The Case for Enhanced Abstract Data Types. In *Proc. of the 23rd VLDB Conference*, pp. 66–75, Athens, 1997.

[TN91] M.M. Tsangaris and J.F. Naughton. A Stochastic Approach for Clustering in Object Bases. In *Proc. ACM SIGMOD Conference*, pp. 12–21, Denver, 1991.

[TN92] M.M. Tsangaris and J.F. Naughton. On the Performance of Object Clustering Techniques. In *Proc. ACM SIGMOD Conference*, pp. 144–153, San Diego, 1992.