An Evaluation of Physical Disk I/Os for Complex Object Processing

Wouter B. Teeuw* Christian Rich§ Marc H. Scholl§ Henk M. Blanken*

*Department of Computer Science, University of Twente
P.O. Box 217, NL–7500 AE Enschede, The Netherlands

§Swiss Federal Institute of Technology (ETH)
ETH-Zentrum, CH–8092 Zürich, Switzerland

Abstract

In order to obtain the performance required for non-standard database environments, we need suitable storage structures for complex objects. In this paper we use an hierarchical complex object model with object references. We describe several storage models for these complex objects, as well as a benchmark to evaluate their performance. We develop a cost model for analytical performance evaluation, and validate the analytical results by means of measurements on the DASDBS complex object storage system. The results show which storage structures for complex objects are the most efficient under which circumstances.

1 Introduction

Data-intensive applications such as geographic information systems, robotics, and CAD/CAM, the so-called *non-standard* database applications, require a high performance in retrieving and processing *complex objects*. Complex objects are data objects that are both highly structured, and large in size. These large clusters of structured data are a unit of manipulation. Lately, the modelling aspects of complex objects got much attention in the so-called object-oriented data models. While such data models are well suited to capture the structural aspects of complex objects [1], achieving the necessary performance is still an open problem. A crucial issue is the design of the internal storage structures: The physical design of a complex object server needs to allow for efficient retrieval and manipulation of the complex objects as a whole and of parts thereof.

In this paper we will focus on performance aspects of manipulating complex objects. In particular, we examine the problem of how to fragment and how to store complex objects in order to achieve a high performance. We will present a number of complex object storage models, some of them are direct, others normalized in nature. We will evaluate the performance of these storage models, both analytically, as well as experimentally. A cost model is developed in order to analytically evaluate the performance, and we used the DASDBS storage system for complex objects [13] to validate our analytical results by performance measurements.

In general, a complex object is built by applying various constructors, such as tuple, set, and list, to other complex objects or basic values. Also, a complex object has some notion of object identity and complex objects have relationships to each other. In this paper, we restricted ourselves to tuples with relation-valued attributes, the so-called nested or NF^2 tuples [12], as examples of complex objects. Our concepts, however, hold for more general objects as well. Also, we mainly focus on disk I/O. In the database world there is a general consensus that the disk I/O seems to be the bottleneck for such shared-nothing systems with a local area network [7].

This paper is organized as follows. In Section 2 we describe the complex object benchmark we used in our performance evaluation. The benchmark has been based on the Altair Complex Object Benchmark [6] and involves object retrieval, navigation, and updates. In Section 3 we present several storage models for complex objects. Also, we present the analytical cost formulas we developed for estimating the disk I/O costs for each storage model. In Section 4 we show the results of our analytical performance evaluation. In Section 5 we show the experimental setup and the results of our validation of the analytical results by means of measurements on DASDBS. In Section 6, finally, we present our conclusions.

2 A complex object benchmark

In order to evaluate complex object storage models, we need a benchmark. In the literature, some benchmarks for complex objects can be found. The most well-known are the (revised) Sun benchmark [5], the Hypermodel benchmark [3], and the Altair complex object benchmark [6]. We primarily focus on the number of disk I/Os, and want a small number of basic retrieval and update operations to evaluate our storage models. Therefore we based our benchmark on the Altair benchmark, the design of which has been influenced by the data model of the O_2 object oriented database system [9]. We use a revised version of this benchmark. The main differences with the original benchmark are that we made the objects larger, variable in size, and variable in structure.

2.1 Structure of our benchmark objects

The structure of our benchmark object is presented in Figure 1. For convenience, we have based the at-

*The investigations were partly supported by the Foundation for Computer Science in the Netherlands SION under project 612-317-025 nicknamed Starfish.

§Current address: Department of Computer Science, University of Ulm, P.O. Box 4066, D–W–7900 Ulm, Germany

```
COMPLEX OBJECT Station = {(          % 1500 tuples
   Key:          INT,                %    4  bytes
   NoPlatform:   INT,                %    4  bytes
   NoSeeing:     INT,                %    4  bytes
   Name:         STR,                %  100  bytes
   Platform:     {(                  %  0-2 tuples
      PlatformNr:  INT,              %    4  bytes
      NoLine:      INT,              %    4  bytes
      TicketCode:  INT,              %    4  bytes
      Information: STR,              %  100  bytes
      Connection:        {(          %  0-4 tuples
         LineNr:         INT,        %    4  bytes
         KeyConnection:  INT,        %    4  bytes
         OidConnection:  LINK,       %    4  bytes
         DepartureTimes: STR )} )},  %  100  bytes
   Sightseeing: {(                   % 0-15 tuples
      SeeingNr:    INT,              %    4  bytes
      Description: STR,              %  100  bytes
      Location:    STR,              %  100  bytes
      History:     STR,              %  100  bytes
      Remarks:     STR )} )}         %  100  bytes
```

Figure 1: The benchmark complex object.

tribute names on a railway example. Our database extension consists of 1500 complex objects of the type `Station`. Each `Station` is uniquely determined by its `Key`, and characterized by some (dummy) atomic attributes, such as its `Name`. Also, each `Station` is characterized by two non-atomic, i.e., relation valued attributes. The tuples in the `Platform` sub-relation represent the platforms from which trains may depart. The tuples in the `Sightseeing` sub-relation represent the tourist attractions in the neighbourhood.

As opposed to the Altair benchmark, we do not have a fixed number of sub-objects per `Station`. We have up to two `Platform` sub-objects, but they are each *created* with an independent *probability* of 80%. So there are *at most*, rather than exactly, two `Platforms`. Similarly, there are at most 15 `Sightseeing` sub-objects. The exact number will be randomly generated between 0 and 15. The `Platforms` are hierarchically structured again, since they contain the relation valued attribute `Connection`. In conformance with the Altair benchmark, each `Platform` has two railroads (each generated with an independent probability of 80%), and each railroad establishes two `Connections` to a neighbouring `Station` (again each generated with an independent probability of 80%). So, each `Platform` has at most four `Connections`, which are each generated with a probability of $(0.80^2 =)$ 64%.

Each `Connection` contains a reference to a randomly chosen `Station` object. The physical reference in `OidConnection` is the address of the referred `Station`. The `Stations` to which a particular `Station` object refers are called its *children*. On the average, each `Station` has $(2 * 80\%)^3 = 4.10$ children.

2.2 The benchmark queries

As with the Altair benchmark we have three types of queries. At first a scan of the database, i.e., read all objects. We have three variants of this scan query.

1a: Retrieve a single `Station` object given its address (OID).
1b: Retrieve a single `Station` objects given its key value.
1c: Retrieve all `Station` objects and divide the measured or estimated performance values by number of objects.

Second, randomly select an object (given its OID), find the identifiers of the objects it refers to (0-8, on the average 4.10 objects), fetch these child-objects, find the identifiers of the objects they refer to (0-64, on the average 16.7), and retrieve the atomic attributes of these *grand-children*. While navigating through an object in order to find the references to its children, only the attributes/tuples that are needed will be projected/selected. We may execute this query either once, or 300 times consecutively. In the latter case, almost all objects are referred to at least once, and the probability of buffer hits or buffer overflow will increase.

2a: Input the root records of the grand-children of a random `Station` object.
2b: As 2a, but 300 times in a *loop* and normalizing the results to a value per loop.

Third, an update version of query 2. The root record of the 0-64 (on the average 16.7) grand-children is modified. We update atomic attributes, that is, the object structure is not changed.

3a: As 2a, followed by an update of the root record of the grand-children.
3b: As 2b, with at the end of each loop an update of the grand-children.

3 Storage models for complex objects

We will investigate four storage models for complex objects. Two models directly store the complex objects on disk, that is, complex objects are stored as a whole on as few disk pages as possible. In addition, structural information may be preserved, such that access to parts is possible without necessarily reading the whole object. Two other models use normalization techniques, that is, they split a complex object into smaller (normalized) pieces and store these (possibly) separately from each other.

In non-standard database application areas, the response time is one of the most important performance indicators. However, the distinguishing feature of the storage models is the number of disk I/Os raised per operation, which will therefore determine the response time. Therefore we rather measure and estimate logical values. We will mainly focus on counting the number of pages that are loaded from secondary storage devices (i.e., the disk) into the main memory buffer (i.e., the cache).

g	: number of tuples in a cluster of tuples
k	: nr. of (small) tuples stored on a single page
m	: nr. of pages for storing an entire relation
p	: nr. of pages to store a single (large) tuple
t	: total number of tuples to be retrieved
C_X	: cost related to the aspect X
S_X	: size in byte of a unit called X
λ_X^f	: number of events X under condition f

Table 1: Explanation of the (nested tuple) parameters.

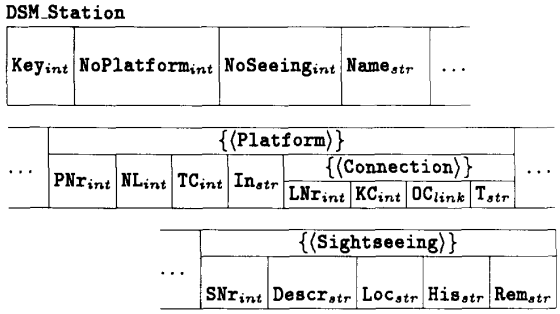

Figure 2: (DASDBS-)DSM representation of Station complex object: 1 wide NF^2-table

For each storage model we will show how to estimate the disk I/O costs. For the total disk I/O costs, two main contributions are important: The number of pages for I/O ($\lambda_{I/O\ pages}$) and the number of I/O calls needed to store or retrieve those pages ($\lambda_{I/O\ calls}$).

$$C_{disk\ I/O} = d_1 * \lambda_{I/O\ calls} + d_2 * \lambda_{I/O\ pages} \quad (1)$$

In the next subsections, we will present formulas for determining $\lambda_{I/O\ pages}$. The derivation of these formulas can be found in an additional report [14]. The parameters we use are summarized in table 1. In the formulas, the div operator gives the quotient, the mod operator the remainder of integer division.

3.1 DSM

With a Direct Storage Model (**DSM**) for complex objects there is no fragmentation. As far as possible, the nested tuples will be stored contiguously on disk [8,15], as shown in Figure 2 for our benchmark object. If the nested tuples (= objects) are larger in size than a page, the pages that store the tuple will not be shared by other tuples. If a tuple is S_{tuple} bytes in size, and the page size is S_{page} bytes, a single tuple spans p pages, with:

$$p = \left\lceil \frac{S_{tuple}}{S_{page}} \right\rceil \quad , \quad S_{tuple} > S_{page} \quad (2)$$

Consequently, if tuples are retrieved on their address or identifier, the number of disk I/Os for retrieving t tuples in their entirety is:

$$\lambda_{I/O\ pages}^{large,entire}(t, p) = t * p \quad (3)$$

If the tuples are smaller in size than a page, several tuples may share a single disk page. The tuples themselves do not span disk pages. If in a single disk access t tuples are retrieved, and these t tuples have been randomly distributed over the m pages that store the entire (nested) relation, the number of page accesses is given by the formula of Bernstein [2]:

$$\lambda_{I/O\ pages}^{small,random}(t, m) = \begin{cases} t & , \ t \leq m/2 \\ \frac{t+m}{3} & , \ m/2 < t \leq 2 * m \\ m & , \ t > 2 * m \end{cases} \quad (4)$$

3.2 DASDBS-DSM

DSM can be enhanced in such a way that, from the set of pages that stores the object, *only* those pages are retrieved that are actually used in a query. We will refer to this alternative direct storage model as **DASDBS-DSM**, since, although basically the same as DSM, it is based on a storage concept that was originally proposed in the DASDBS project [13]. Structural information is gathered in an "object header" that allows dedicated access to parts of a complex object. For DASDBS-DSM, Equation 3 has to be adapted. How many of the p pages per tuple will not be retrieved depends on both the percentage of tuple-data that is not used, and the clustering of these data within the object. Paul [11] showed that:

$$\lambda_{I/O\ pages}^{large,partial}(t, p) = t*(p - \sum_{\substack{\text{not used subobjects} \\ \geq \text{ in size than page}}} \frac{S_{subobject} - S_{page} + 1}{S_{page}}) \quad (5)$$

3.3 NSM

In contrast to DSM implementations, relational DBMSs store only flat tuples. We might map complex objects onto flat relations and store these flat relations in the database. The most obvious way to do so, is to *unnest* the complex object. Somehow the complex object structure has to be preserved, for example by storing primary to foreign key relationships in additional attributes within the relations [4,10]. We use a Normalized Storage Model (**NSM**) in which three attributes are added to each tuple in each relation in order to keep the object structure: a globally unique foreign key referring to the complex object the tuple belongs to (i.e., the root tuple), a foreign key referring to the parent tuple, and an own key which child tuples can use as a reference. The latter two numbers must be unique within their local context and make the normalization of the object and its inverse unambiguous. Figure 3 shows an example. Notice that superfluous key attributes have been omitted. That is, the parent key can be left out on the first level of nesting (equal to root key), the own key on the lowest level of nesting (not referred to), and on the root level we only need the own root key.

With NSM, we can again use Equation 4 to estimate the number of disk I/Os if retrieving t tuples that have

NSM_Station

Key_{int}	$NoPlatform_{int}$	$NoSeeing_{int}$	$Name_{str}$

NSM_Platform

RootKey	OwnKey	PNr_{int}	$Noline_{int}$	$TCode_{int}$	$Inform_{str}$

NSM_Connection

RootKey	ParentKey	$LineNr_{int}$	Key_{int}	Oid_{link}	$Times_{str}$

NSM_Sightseeing

RootKey	SNr_{int}	$Descr_{str}$	Loc_{str}	$History_{str}$	$Remarks_{str}$

Figure 3: NSM representation of Station complex object: 4 flat tables

been randomly distributed over m pages. However, tuples that belong to the same root or parent are likely to be stored clustered together. Suppose the t tuples have been stored one behind another. Notice that the tuples do not span pages (we have k tuples per page), but the cluster might! For this situation we use the next equation:

$$\lambda^{small,cluster}_{I/O\ pages}(t, m, k) = \begin{cases} 1 + \frac{t-1}{k}, & t \leq m * k - k + 1 \\ m, & t > m * k - k + 1 \end{cases} \quad (6)$$

If several objects, each represented by several tuples, are retrieved in a single I/O call, we get the situation that the tuples have been clustered, not in one large cluster of size t, but in $\frac{t}{g}$ smaller clusters (groups) of size g. If the clusters are randomly located on the m pages, we get:

$$\lambda^{small,groups}_{I/O\ pages}(t, m, k, g) =$$

$$\begin{cases} \lambda^{small,random}_{I/O\ pages}(\frac{t}{g} * (1 + \frac{g-1}{k}),\ m), & g \leq k - 1 \\ \lambda^{small,random}_{I/O\ pages}(2 * \frac{t}{g} * (\frac{(g-k)^2-1}{k^2} + \frac{2*k-g-1}{k}), & \\ \qquad m - \frac{g-k+1}{k} * \frac{t}{g}), & k - 1 < g \leq 2k - 2 \\ \lambda^{small,groups}_{I/O\ pages}(t - \frac{t * ((g-k+1)\ \underline{div}\ k)*k}{g}, & \\ \qquad m - \frac{t * ((g-k+1)\ \underline{div}\ k)}{g}, & \\ \qquad k,\ (g - k + 1)\ \underline{mod}\ k + k - 1) & \\ \qquad + \frac{t * ((g-k+1)\ \underline{div}\ k)}{g}, & g > 2k - 2 \end{cases} \quad (7)$$

For $g > 2k - 2$, Equation 7 calls itself. However, in this call g is always less or equal to $2k - 2$, so there is *at most* a single recursive call. The parameter k represents the number of tuples per page, and can be expressed in the page-size S_{page} and object-size S_{tuple}.

$$k = S_{page}\ \underline{div}\ S_{tuple} = \left\lfloor \frac{S_{page}}{S_{tuple}} \right\rfloor,\ S_{tuple} \leq S_{page} \quad (8)$$

3.4 DASDBS-NSM

With NSM many time and resource consuming joins will be needed to reassemble the object from its normalized representation. To speed up object reassembly with NSM we might cluster the flat tuples

DASDBS-NSM_Station

Key_{int}	$NoPlatform_{int}$	$NoSeeing_{int}$	$Name_{str}$

DASDBS-NSM_Platform

RootKey	{⟨PlatformsOfStation⟩}				
	OwnKey	PNr_{int}	$Noline_{int}$	$TCode_{int}$	$Inform_{str}$

DASDBS-NSM_Connection

RootKey	{⟨ConnectionsOfStation⟩}				
	ParentKey	{⟨ConnectionsOfPlatform⟩}			
		LNr_{int}	Key_{int}	Oid_{link}	$Times_{str}$

DASDBS-NSM_Sightseeing

RootKey	{⟨SightseeingsOfStation⟩}				
	SNr_{int}	$Descr_{str}$	Loc_{str}	$History_{str}$	$Remarks_{str}$

Figure 4: DASDBS-NSM representation of Station complex object: 4 (NF^2) tables

that have an identical root foreign key, i.e., that belong to the same tuple-object. Within such a cluster, we again cluster on the parent foreign key. We can *force* such a clustering by means of *nesting* on these attributes. Such a nesting has two advantages. First, the foreign root and/or parent keys are not replicated in all the 'sibling' tuples. Second, after this nesting only a *single* tuple per relation per object is left. Therefore, it becomes efficient to keep an additional table (index) with a *single* entry per object and a *fixed* and *limited* number of addresses in this entry, namely for each relation that stores the object the address of the single corresponding relation tuple. This transformation table immediately shows the addresses of all the tuples that together store an object, given the (logical) key of this object. Since this key is stored in all the tuples that keep data of the object anyway, namely as 'root foreign key,' a flexible and fast retrieval of the object is possible. We refer to this storage model as **DASDBS-NSM**, because it uses nesting and addresses as provided in DASDBS. Figure 4 shows an example. The number of disk I/O for this storage model can be estimated by using the equations of Section 3.1.

4 Analytical performance evaluation

In this section we present the results of our analytical performance evaluation. These results have been based on a DASDBS-like system. That is, if a nested tuple is too large to be stored on a single page, the structure information is mapped onto a set of *header pages*, which is disjoint from the set of *data pages* that store the data, as well as a minimum amount of structure information. Storing the structure information and the data on separate pages has the advantage that we can first retrieve and analyse the structure, whereupon we need to retrieve *only* those data pages that are used. However, as a result we may have some internal wasted space in large tuples, due to the fact that the header page(s) are not completely used.

Table 2 shows the average DASDBS-sizes of our benchmark tuples. These sizes were found by analyzing the DASDBS storage structures [14]. We took the average values of 1.60 Platform-, 4.10 Connection-,

RELATION	TUPLES PER Station	TUPLES IN TOTAL	S_{tuple}	k	p	m
DSM						
_Station	1.0	1500	6078	–	4	6000
NSM						
_Station	1.0	1500	154	13	–	116
_Platform	1.6	2400	168	11	–	219
_Connection	4.1	6144	170	11	–	559
_Sightseeing	7.5	11250	456	4	–	2813
DASDBS-NSM						
_Station	1.0	1500	154	13	–	116
_Platform	1.0	1500	277	7	–	215
_Connection	1.0	1500	668	3	–	500
_Sightseeing	1.0	1500	5213	–	3	4500

Table 2: Average DASDBS-sizes of benchmark tuples.

and 7.50 Sightseeing sub-objects per Station, and used the attribute sizes as shown in Figure 1. The number of tuples per page k, or pages per tuple p, as well as the total number of pages m that store a relation can be found given the DASDBS 'effective' page size of 2012 byte (2048 byte minus a header of 36 byte). Tuples of DSM_Station and DASDBS-NSM_Sightseeing are larger in size than a page, and therefore will be stored distributed over header and data pages, giving some wasted space.

Finally, we need an additional formula in order to include the effects of database caching. The queries 2 and 3 contain a loop. The same object may be referenced several times during this loop. Therefore, we might have buffer hits. Suppose we randomly select an object out of N_{tot} objects in total. Also, we execute this random selection N_{num} times. That is, we take N_{num} objects out of N_{tot} *with* putting them back. Since the probability that an object is *not* selected is equal to $(\frac{N_{tot}-1}{N_{tot}})^{N_{num}}$, the number of objects N_{sel} that is selected at least once is equal to:

$$N_{sel}\,(N_{tot}, N_{num}) = N_{tot} * (1 - \left(\frac{N_{tot}-1}{N_{tot}}\right)^{N_{num}}) \quad (9)$$

The results of our analytical performance evaluation are shown in Table 3. In the table we also showed the results for the imaginary situation without wasted disk space. The latter results have been marked with a prime. The results for NSM with index are shown as well. Notice that all results for query 1 have been normalized to values *per object*, and for query 2 and 3 *per loop* of retrieving an object, its children and its grand-children. Since we assumed a large cache, all estimates are *best case*.

With DSM a Station object is stored clustered on four pages. Therefore we have to use equation 3 to estimate the number of page accesses. With DASDBS-DSM equation 5 is used. Since the Sightseeing sub-

STORAGE MODEL	QUERY 1 (A)	QUERY 1 (B)	QUERY 1 (C)	QUERY 2 (A)	QUERY 2 (B)	QUERY 3 (A)	QUERY 3 (B)
DSM	4.00	6000	4.00	86.9	19.7	154	39.1
DASDBS-DSM	4.00	6000	4.00	43.4	9.87	76.8	19.5
NSM	–	3710	2.47	675	2.25	692	2.64
DASDBS-NSM	6.00	121	3.55	21.8	2.01	38.5	2.39
DSM'	3.00	4500	3.00	65.2	14.8	115	29.3
DASDBS-DSM'	3.00	4500	3.00	21.7	4.94	38.4	9.76
NSM+index	5.96	121	2.47	23.2	2.25	39.9	2.64
DASDBS-NSM'	5.00	120	2.55	21.8	2.01	38.5	2.39

Table 3: Estimates of the number of page I/Os.

STORAGE MODEL	QUERY 1 (A)	QUERY 1 (B)	QUERY 1 (C)	QUERY 2 (A)	QUERY 2 (B)	QUERY 3 (A)	QUERY 3 (B)
DSM	5.00	5360	3.57	72.0	57.7	139	111
DASDBS-DSM	5.00	5360	3.57	34.0	20.6	85.0	63.5
NSM	–	3820	2.55	700	2.33	715	3.76
DASDBS-NSM	10.0	145	3.54	18.0	2.05	34.0	3.48

Table 4: Measurements of the number of physical page I/Os $\lambda_{I/O\ pages}$.

objects are not used in query 2 and 3, we only need to retrieve the header page and a single data page in these queries. With NSM we have no identifiers (see Section 3.3), so query 1a is not relevant. We make the unrealistic assumption that all joins can be performed in main memory, which makes our analytical results *best case*. If NSM is supported by an index, a page is read from disk then and only then if a tuple it stores is requested. Table 3 shows the results for this index case as well. With DASDBS-NSM, finally, we use an in-memory table (index) to translate the object key to the (four) addresses of the corresponding tuples in the relations that store the object. Notice that with query 1b, only the root tuple of the object is selected based on a value selection, whereupon we use the addresses in the index table to retrieve all other data by address.

5 Results of DASDBS measurements

5.1 The number of disk I/Os

The results of measurements of the number of physical page I/Os are shown in Table 4. Notice that, although DASDBS is a storage system for nested relations, flat relations are a simple form of nested relations and therefore all storage models of Section 3 can be implemented in DASDBS. In the generated benchmark data, each Station object contained, on the average, 1.59 Platforms, 4.04 Connections, and 7.64 Sightseeings. We only measured the number of pages read or written from/to the database relations. So we did not account for additional I/Os needed to access the data dictionary, to retrieve the tables with addresses, etc. We will give some comments on the results.

The results of query 1a, select an object on identifier, depend on the particular object that is selected. The same holds for query 1b with DASDBS-NSM. We

STORAGE MODEL	QUERY 1			QUERY 2		QUERY 3	
	(A)	(B)	(C)	(A)	(B)	(A)	(B)
DSM	3.00	2730	1.82	35.0	29.6	49.0	36.9
DASDBS-DSM	3.00	2730	1.82	34.0	20.6	48.0	23.8
NSM	–	3820	2.55	700	2.33	703	3.38
DASDBS-NSM	9.00	144	2.18	18.0	2.05	22.0	3.10

Table 5: The number of I/O calls $\lambda_{I/O\ calls}$ needed to retrieve the pages as presented in Table 4.

STORAGE MODEL	QUERY 1			QUERY 2		QUERY 3	
	(A)	(B)	(C)	(A)	(B)	(A)	(B)
DSM	4.00	5360	3.57	71.0	77.7	133	149
DASDBS-DSM	4.00	5360	3.57	33.0	39.9	105	140
NSM	–	3830	2.55	1270	1240	1280	1260
DASDBS-NSM	6.00	143	3.54	16.0	21.6	31.0	41.2

Table 6: Measurements of the number of page fixes $\lambda_{buffer\ fixes}$ in buffer.

measured the number of I/Os for an 'average' object with 2 `Platform`-, 4 `Connection`-, and 7 `Sightseeing` sub-objects. For the direct storage models, the number of disk I/Os with query 1b and 1c is lower than estimated. This can be explained as follows: on the average a DSM_Station tuple contains a header page and 2.02 data pages. But, due to the ceiling function in equation 2, the analytical value of p is 4 rather than 3.02 (see Table 2). So an object that is somewhat larger than the average size requires no extra page (enough free space in data pages), whereas a slightly smaller object saves a data page. Therefore, the estimated values are somewhat too large.

For query 2a, we took a randomly selected `Station` that happened to have 4 children and 12 grandchildren. Since 12 grand-children is less than the average number of 16.7, we can explain the low number of page I/Os. With query 2b, the number of page accesses for DSM and DASDBS-DSM is much larger than estimated. But, as emphasized before, the estimates are best case estimates. With a buffer of 1200 pages and almost 1500 objects to retrieve, obviously a cache overflow occurs. With DSM the effect is larger than with DASDBS-DSM.

Taking into account the results for query 2, the results for query 3 are as expected, except for DASDBS-DSM. Additional measurements show that with DASDBS-DSM the number of page writes, with both query 3a and query 3b, is larger than expected. We will explain this observation later on, in Section 5.3. With query 3b, NSM and DASDBS-NSM show a larger number of writes than estimated as well. In both cases, the same relation has to be updated, namely (DASDBS-)NSM_Station. Since the tuples in these relations are small and consequently share pages, equation 4 says that all 116 pages are to be written back to disk. That makes 0.387 page writes per loop, but the measured value is 1.42 page writes. Probably due to cache overflow our estimate was too optimistic. The difference is small if measured in absolute values, but the relative difference is large.

5.2 The I/O calls and buffer fixes

Except for the number of pages for I/O, we measured some other parameters as well. Table 5 shows the results for the number of function calls needed to retrieve the number of pages as presented in Table 4. Table 6 shows the measurements for the number of pages that have been fixed in the buffer for read or write during the execution of the benchmark queries.

A short comparison of Table 5 with Table 4 shows the following. If small tuples or data units are *read* (NSM all queries, DASDBS-DSM and DASDBS-NSM query 2 and 3) a single page per I/O call is retrieved. With larger units, several pages are read in a single call. E.g., with DSM, about 2 pages are read per I/O call. The number of pages to be read per I/O call seems somewhat low. However, notice that DASDBS uses separate I/O calls to retrieve the root page (first page of a large object), the additional header pages (if any), and the data pages. With the *write* operation, more pages are handled in a single I/O call. This can be explained by the fact that pages are written to the database relations only then if either the query execution has been finished (database disconnect) or the page buffer overflows. The advantage appears to be the largest with DSM and DASDBS-DSM (on the average respectively 30 and 20 pages per write for query 3).

The results in Table 6, the number of page fixes in buffer, were reflected in the overall query response times. With query 2b and query 3b, e.g., the *average number* of page fixes *per loop* are presented in the table, and we have 300 of such loops. Consequently, with NSM the entire query 2b program uses more than 370,000 page fixes. On a Sun 3/60 workstation this program took about $2\frac{3}{4}$ hours, whereas the same query was executed within at most $\frac{1}{4}$ hour for the other storage models.

5.3 Varying the object size

In the original Altair benchmark, the `Sightseeing` sub-relation does not exist. We made the number of `Sightseeings` equal to zero to get the original benchmark, and ran the benchmark queries. In addition, we substantially increased the number of `Sightseeings` to a maximum of 30, in order to investigate the advantages of DASDBS-DSM as compared with (pure) DSM. The results are shown in Figure 5. Since 'pure' NSM has not shown to be particularly suited for complex object storage, we do not consider this storage model any longer in this paper. Also, we will only focus on those queries that give us enough information about the effect we investigate. In the database extensions we generated, the average number of `Sightseeings` per `Station` appeared to be 0, 7.64, and 15.3 for the three cases.

From Figure 5 we observe that the larger the sub-objects not used, the larger the advantage of DASDBS-DSM over DSM since, as far as possible, the pages that store these `Sightseeing` sub-objects are not retrieved.

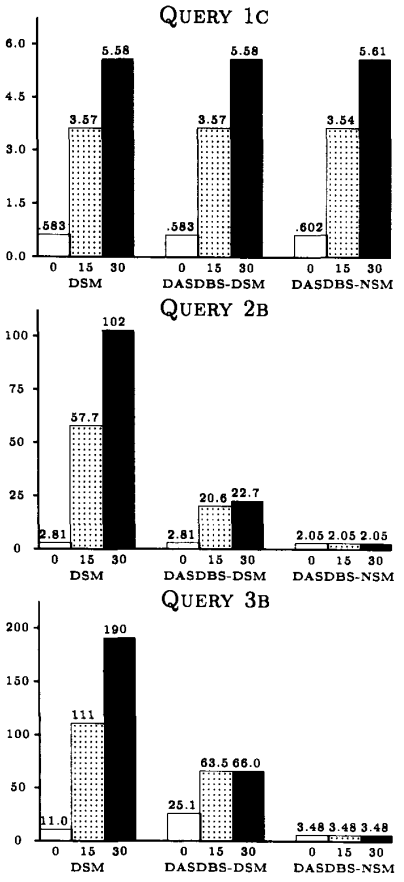

Figure 5: Measurements of $\lambda_{I/O\ pages}$ while the maximum number of **Sightseeings** is 0 (white), 15 (grey), and 30 (black).

With DASDBS-NSM, the results for query 2b and query 3b are independent of the number of **Sightseeings** because these queries do not use the **DASDBS-NSM_Sightseeing** relation (Figure 4). In general DASDBS-NSM needs the fewest disk I/Os, but for smaller objects the advantage of DASDBS-NSM over the direct storage models melts away. One reason is that with smaller objects the DSM_Station tuples become smaller than a page. Therefore they do not have separate header and data pages any longer. Rather, several objects will share a single page. Another reason is that the direct models suffer from cache overflow. This problem reduces with smaller objects. With the update query 3b, the advantage of DASDBS-NSM over the direct models remains. With DASDBS-NSM only small root tuples in the **DASDBS-NSM_Station** relation are updated, of which there are many on a single page. The direct storage models replace or update much larger tuples, of which there are only a few on a single page.

Again, we observe that DASDBS-DSM is bad with updates, in particular for small objects. This can be explained as follows. With the storage models DSM, NSM, and DASDBS-NSM the update has been implemented as a replacement of the entire (nested) tuple containing the root attributes. That is, we replace a tuple of **DSM_Station**, **NSM_Station**, or **DASDBS-NSM_Station** respectively (see Figure 2–4). On the average, 16.7 tuples are updated at the same time, which can be implemented in DASDBS as a single 'replace set of tuples' operation. With DASDBS-DSM, on the other hand, we cannot replace the entire tuple since for each tuple only those pages are retrieved that are actually needed (rather than all pages). Therefore the update has been implemented as an 'change attribute' operation for a root attribute. With a 'change attribute' operation we can change a set of atomic attributes of a *single* tuple. So, *per tuple* such an operation is needed. Unfortunately, in DASDBS each update operation allocates a page pool, of which *all* pages are written. With large objects (several pages), the influence on the results should not be so large. For small objects, however, this alternative update protocol makes the results for DASDBS-DSM very bad, even though the page pool is only a single page in size.

5.4 Database caching

Let us investigate the problem of caching into more detail. The effect of caching can be shown by varying the number of objects in the database, as shown in Figure 6. The figure shows the results for query 2b, in which we executed the query loop $\frac{1}{5}$*'database size' times. So, with 1500 objects the retrieval loop is executed 300 times and with 100 objects only 20 times, etc. In this way, about the same percentage of the total number of objects is retrieved for each database size. The horizontal axis has a logarithmic scale.

In the figure, we showed the analytically expected value of Table 3 as well (A^b). The analytical estimates are *best case* since we assumed there was no cache overflow. In a *worst case* situation, there will be no cache hits at all, which is (approximately) the case with query 2a. Therefore, we may regard the analytically calculated value for query 2a in Table 3 as a worst case estimate for query 2b. We have shown this worst case estimate in Figure 6 as well (A^w).

From the results, we clearly observe that DSM is the most, and DASDBS-NSM the least sensitive to cache overflow. For small database sizes there is no cache overflow. In the absence of cache overflow, the measured values are close to the expected best case values. For (DASDBS-)DSM the measured values are somewhat lower than the best case estimates because, as already mentioned in Section 5.1, in the estimate the number of pages per object is rounded upwards due to ceiling.

Without cache overflow, DASDBS-NSM needs the least disk I/Os (about 2 pages per loop) because the tuples that are retrieved (namely from the relations **DASDBS-NSM_Station** and **DASDBS-NSM_Connection**)

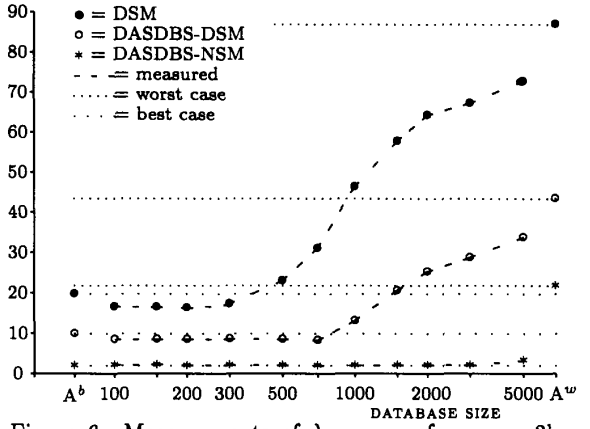

Figure 6: Measurements of $\lambda_{I/O\ pages}$ for query 2b while varying the database size, as well as the analytically expected best (A^b) and worst (A^w)case values.

STORAGE MODEL	$\lambda_{buffer\ fixes}$	$\lambda_{I/O\ calls}$	$\lambda_{I/O\ pages}$
CONNECTIONS: 0–34 (AVERAGE 3.99)			
DSM	77.8	29.1	56.9
DASDBS-DSM	39.6	19.5	19.7
DASDBS-NSM	21.1	1.95	2.01
CONNECTIONS: 0–8 (AVERAGE 4.04)			
DSM	77.7	29.6	57.7
DASDBS-DSM	39.9	20.6	20.6
DASDBS-NSM	21.6	2.12	2.12

Table 7: Measurements for query 2b while changing the maximum number of Connections per Station object.

STORAGE MODEL	$C_{processing}$		$C_{disk\ I/O}$		C_{total}
	$\lambda_{buf.\ fixes}$	C_{join}	$\lambda_{IO\ calls}$	$\lambda_{IO\ pages}$	
DSM	+ −	+ +	+ +	− −	−
DASDBS-DSM	+ −	+ +	+	−	+
NSM	− −	− −	− −	+	− −
DASDBS-NSM	+ +	−	−	+ +	+ +

Table 8: Overall evaluation of all storage models.

are small in size and many of them are stored on a single page. DASDBS-DSM needs more disk I/Os (about 8.5 pages per loop) since for each accessed object, at least a header and a data page is needed. DSM needs the most physical I/Os (about 16.5 pages per loop) since the not-used data pages of an object are retrieved as well.

With cache overflow, we observe that the measured values for the direct models seem to approach a limit that is lower than the worst case estimates. This is again due to the fact that due to the ceiling in equation 2 the average object size was taken to be 4 pages. In the generated extensions, however, the average number of pages per object will be smaller. The real (average) number of pages per object is even closer to 3 than to 4, which can be seen from the fact that if we take 3 pages per object in the analytical evaluation, the worst case estimate for DSM is given by the query 2a / DSM' case of Table 3: 65.2, which is very close to the measured value for large database sizes.

5.5 Data skew

For each Station object, the number of children is determined by the *probability* with which a Platform or Connection sub-object is generated (default 80%), and the *fanout* of these two subrelations (default 2). The average number of children is (fanout*probability)^3, the average number of grandchildren (number of children)^2. Data skew, the variation in object size and structure, might be important for the storage models. Therefore, we created a database with this probability equal to 20% (instead of 80%), and this fanout equal to 8 (instead of 2).

The results for query 2b are shown in Table 7. The average number of sub-objects appeared to be about the same as with the original benchmark extension: 1.57 Platforms and 3.99 Connections per Station. We kept the maximum of 15 Sightseeings. The maximum number of Platforms per Station appeared to be 6 (rather than 2), and the maximum number of Connections 34 (rather than 8). For many loops the number of children to be retrieved was 0, but in some loops up to 95 grand-children had to be retrieved. Although the number of physical I/Os was somewhat more concentrated into fewer loops, the overall figures are similar to those of the original benchmark, which we included in Table 7 as well. Notice, however, that in a distributed system the data skew might cause more effects, which could possibly be distinguishing for the storage models as well. For, with data skew the disk I/Os are likely to be less equally distributed over the nodes if we store a single object on a single node.

6 Discussion and conclusions

We introduced several storage models for complex objects, developed an analytical model for estimating the number of disk I/Os, and validated our analytical results by means of measurements on the DASDBS complex object storage system. The results of our validation experiments was in agreement with what we expected from our analytical results and, if not so, the deviations could be explained by taking the DASDBS specific features into account. Based on the results, and using the results of additional measurements and global considerations, we may give an overall evaluation of the four storage models that have been defined in Section 3. The (DASDBS-based) result is shown in Table 8. For each cost factor, we ordered the four storage models from the best (++) to the worst (− −). Our judgement is motivated as follows.

The judgement for disk I/O costs has been based on our validation tests. With respect to the number of pages for disk I/O, DASDBS-DSM needs less retrievals as compared with DSM. However, the direct storage

models need at least two page fetches per large tuple (header and data), which makes the normalized storage models superior. With NSM, however, no efficient addressing mechanism can be provided, which makes small queries inefficient. Finally, DASDBS-DSM appears to be disadvantageous with updates since we can not replace entire tuples if only a part of it has been retrieved, and therefore we have to use the less efficient change attribute operations in DASDBS.

Given the number of pages for I/O ($\lambda_{IO\ calls}$) we measured the number of function calls $\lambda_{IO\ pages}$ to retrieve these pages (Table 5). The normalized models need the most I/O calls per retrieved page. NSM even reads only a single page per retrieval call. With DSM we retrieve the largest number of pages per call. Notice that the number of pages per call is dependent on the number of pages in total, which seems to be a more important performance indicator.

We measured the number of page fixes in buffer, which is an indicator of the CPU load, as well, and concluded that NSM behaved very bad. DASDBS-NSM uses the least page fixes. The both direct storage models behave about equal, the DASDBS version better with reads but slightly worse with writes.

So far, we did not take into account that, e.g., when retrieving an entire object, with the normalized storage models we still have to join the data that has been retrieved. We omitted this join in both our analytical evaluation, and our measurements. Although we do not want to concentrate on all kind of join algorithms, it is nevertheless obvious that NSM suffers from these joins. Taking into account the large number of page fixes as well, it is clear that the processor costs are unacceptable large with NSM. With DASDBS-NSM we still have the joins, but can use the table with addresses to efficiently support the join execution.

As an overall conclusion, DASDBS-NSM seems to be the best and NSM the worst. Also, DASDBS-DSM is (more powerful thus) better than DSM.

Acknowledgements

This work has been produced while the first author was on a three-months leave from his university and joined the database research group of Prof. Schek at the ETH Zürich, in which DASDBS is maintained, further developed, and above all available as a base to experiment with. In particular, the authors express their thanks to Gisbert Dröge and Andreas Wolf for their support and advice, which was of great help in programming with and in DASDBS.

References

[1] B. R. M. van den Akker and H. M. Blanken, Geographic Data Modelling in TM, in: P. Sadanandan and T. M. Vijayaraman(Eds.), Advances in Data Management. Proceedings Third International Conference on Management of Data, Bombay, India, December 12–14, 1991 (McGraw-Hill, New Dehli, India, 1991) 107–126.

[2] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie, Jr., Query Processing in a System for Distributed Databases (SDD-1), ACM Transactions on Database Systems 6 (1981) 602–625.

[3] A. J. Berre, T. L. Anderson and M. Mallison, The HyperModel Benchmark, Oregon Graduate Center, Technical Report CS/E 88-031, Beaverton, OR, 1988.

[4] H. M. Blanken and A. Ybema, Storage of Versioned Objects in a CIM Environment, Proceedings International Conference on Data and Knowledge Systems for Manufacturing and Engineering, Hartford, CT, October 19–20, 1987 (IEEE Computer Society Press, Washington, DC, 1987) 65–74.

[5] R. G. G. Cattell, Object-Oriented DBMS Performance Measurement, in: K. R. Dittrich(Ed.), Advances in Object-Oriented Database Systems. Proceedings Second International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebenberg, FRG, September 27–30, 1988 (Springer-Verlag, Berlin, 1988) 364–367.

[6] D. J. DeWitt, P. Futtersack, D. Maier and F. Velez, A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems, in: D. McLeod, R. Sacks-Davis and H. Schek(Eds.), Proceedings of Sixteenth International Conference on Very Large Data Bases, Brisbane, Australia, August 13–16, 1990 (Morgan Kaufmann Publishers, Palo Alto, CA, 1990) 107–121.

[7] D. J. DeWitt and J. Gray, Parallel Database Systems: The Future of Database Processing or a Passing Fad?, SIGMOD RECORD 19 (1990) 104–112.

[8] U. Deppisch, H. -B. Paul and H. -J. Schek, A Storage System for Complex objects, in: K. Dittrich and U. Dayal(Eds.), Proceedings 1986 International Workshop on Object-Oriented Database Systems, Pacific Grove, CA (IEEE Computer Society Press, Washington, DC, 1986) 183–195.

[9] O. Deux et al., The Story of O_2, IEEE Transactions on Knowledge and Data Engineering 2 (1990) 91–108.

[10] R. Lorie and W. Plouffe, Complex Objects and their Use in Design Transactions, IEEE 1983 Proceedings of Annual Meeting – Database Week: Engineering Design Applications, San Jose, CA (IEEE Computer Society Press, Washington, DC, 1983) 115–121.

[11] H-B. Paul, DAS Datenbank-Kernsystem für Standard- und Nicht-Standard- Anwendungen – Architectur, Implementierung, Anwendungen – (Darmstadt, Germany, November 1988).

[12] H. -J. Schek and M. H. Scholl, The Relational Model with Relation-Valued Attributes, Information Systems 11 (1986) 137–147.

[13] H. J. Schek, H. -B. Paul, M. H. Scholl and G. Weikum, The DASDBS Project: Objectives, Experiences, and Future Prospects, IEEE Transactions on Knowledge and Data Engineering 2 (March 1990).

[14] W. B. Teeuw, C. Rich, M. H. Scholl and H. M. Blanken, An Evaluation of Physical Disk I/Os for Complex Object Processing, Departement Informatik, ETH Zürich, Technical Report 183, Zürich, Switzerland, September 1992.

[15] P. Valduriez, S. Khoshafian and G. Copeland, Implementation Techniques of Complex Objects, in: W. Chu, G. Gardarin, S. Ohsuga and Y. Kambayashi(Eds.), Proceedings of Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, August 25–28, 1986 (Morgan Kaufmann Publishers, Los Altos, CA, 1986) 101–110.