A Logical Approach to the Management of Object Identifiers in Non-Conventional Database Management Systems

Carlos Roberto Valêncio¹, André Cid Ferrizzi¹, Fábio Renato de Almeida¹, Juliano Augusto Carreira¹, Mario Luiz Tronco²
1 Universidade Estadual Paulista, Depto. de Ciências de Computação e Estatística, São José do Rio Preto, Brazil
2 Universidade de São Paulo, Departamento de Engenharia Mecânica, São Carlos, Brazil
valencio@ibilce.unesp.br, ferrizzi@gmail.com, fabiorenato.aluno@hotmail.com, julianocarreira@gmail.com, mltronco@sc.usp.br

Abstract— Non-conventional database management systems are used to achieve a better performance when dealing with complex data. One fundamental concept of these systems is object identity (OID), because each object in the database has a unique identifier that is used to access and reference it in relationships to other objects. Two approaches can be used for the implementation of OIDs: physical or logical OIDs. In order to manage complex data, was proposed the Multimedia Data Manager Kernel (NuGeM) that uses a logical technique, named Indirect Mapping. This paper proposes an improvement to the technique used by NuGeM, whose original contribution is management of OIDs with a fewer number of disc accesses and less processing, thus reducing management time from the pages and eliminating the problem with exhaustion of OIDs. Also, the technique presented here can be applied to others OODBMSs.

Ⅰ. INTRODUCTION

One of the main characteristics of an OODBMS is the concept of an object identity [1]. Every object in the database has a unique OID (Object Identifier), and this OID is used to access the object and to reference it in relationships to other objects. The process of finding an object on disc through its OID must be performed as fast as possible, so the technique used for the mapping from the OID of an object to its physical address on disc influences the overall performance of the database system. Considering applications that deal with lots of complex objects, OODBMSs offer a better performance than RDBMSs [2], because in OODBMSs the relationships between objects are done through OIDs, and in RDBMSs the linked tuples must be accessed through Cartesian product.

Some techniques for the management of OIDs are available in literature, such as using the physical address of the object on disc as its OID, or by using a logical approach with or without auxiliary data structures to map logical OIDs to physical addresses. Each technique comes with pros and cons. In this paper we analyze these techniques and the main characteristics associated to them and present an improvement to the Indirect Mapping technique used by a non-conventional database management system, whose original contribution is management of OIDs with a fewer number of disc accesses and less processing, thus reducing management time from the pages and eliminating the problem with exhaustion of OIDs. Also, the technique presented here can be applied to others OODBMSs.

Ⅱ. PHYSICAL APPROACH

A database can be seen as a sequence of pages containing objects, where each object has a unique identifier (OID). To access an object on a database, it is necessary to know its physical address, that is, where on disc the object is located. The fastest way to do this is by using an OID that points directly to this physical address. This kind of OID is called a physical one [4].

The advantage of this kind of direct mapping is that, once the OID of an object is known, its location is also known. But this approach comes with the problem that the object itself cannot be relocated, since moving the object would also require changing its OID, invalidating all previous references to it. One solution to this problem would be to insert a pointer to its new position in the original position of the object. This situation is illustrated in Fig. 1, where the object o3 is relocated to page 9 and slot 0.

Though the problem of relocation can be solved by using forward pointers, this breaks the main purpose of a physical OID, which is of knowing the physical address of an object through its OID. Also, with forward pointers being inserted, the database becomes fragmented, using more physical space [4]. This situation can also be seen in Fig. 1. Despite these facts, one OODBMS that uses physical OIDs is O2 [5].

An alternative for the physical approach of an OID is a logical approach. By using a logical mechanism, the object can be relocated whenever it is necessary; therefore eliminating the problem of database fragmentation, but now another problem is introduced: the mapping time from the logical address to the physical location of the object. The next section discusses some of the available logical techniques in literature.
III. LOGICAL APPROACH

A logical OID can be seen as a surrogate for the physical address of an object on disc. Since this kind of OID does not contain any direct physical indication of the object on disc, the object can be relocated whenever it is necessary without invalidating its OID. This flexibility, however, comes with an extra overhead, the need for a technique to find where the object is located on disc by knowing its logical OID. Despite this overhead, forward pointers are no more necessary when logical OIDs are used, and this makes the logical approach still worthwhile when the problems faced by the physical one are considered [6].

In this section three techniques for the management of logical OIDs are presented: Hashing, B+ Trees, and Direct Mapping.

A. Hashing

Consider a database containing pointer pages and data pages. Pointer pages are pages that store object OIDs and their respective physical addresses on disc, while data pages are pages that store those objects. In the example illustrated in Fig. 2, pages 1 through 10 in the database were configured as pointer pages, and from page 11 on are stored data pages.

The idea behind the Hashing technique is that pointer pages will work as buckets, this way, every time an object is requested, its OID is passed to a hash function that will map it to one of the pointer pages (1 through 10 in the example illustrated in Fig. 2), then the mapped page will be brought from disc to memory and the physical address of the object will be known.

In Fig. 2, it is also possible to see that the object o2 was previously located at page 12. The relocation of it to page 502 does not disturb the system because all the references to the object are being made through its logical OID, in this case, 18.

Two OODBMSs that use the Hashing technique are Versant and Itasca [4].

B. B+ Trees

As in Hashing, the B+ Trees technique uses pointer pages to keep track of the physical address of the objects on disc based on their corresponding logical OIDs. But unlike the Hashing technique, B+ Trees are capable of adjusting themselves as the database goes acquiring more objects. In a B+ Tree structure, each node holds some OIDs along with some pointers that indicate the next pointer page in a database to be read in order to get closer to the desired physical address of the requested object. The node that holds this address is the leaf node, and like the other nodes, it corresponds to a physical page in the database.

The whole structure can be kept on disc. Part of it might be in the memory, thus reducing the running time to find an object address, but at the cost of a higher utilization of memory.

Two OODBMSs that use the B+ Tree technique are GemStone and Shore [4].

C. Direct Mapping

In order to get the best of both approaches, Direct Mapping implements the physical approach only to pointer pages in a database. This way, by knowing the OID of an object, its physical location is not known, but the physical location of its pointer is known.

To access an object, just one block read will be necessary to get its position, no matter how large the database is. Of course this brings the same problems faced by the physical approach mentioned before, but this time the problems are limited to pointer pages, so now it is possible to reorganize data pages.

In Direct Mapping, the pointer pages and data pages may be freely distributed across the file. In our sample structure, the pointer pages have space for five addresses (0 through 4) – these are the slots of the page. The data pages have six entries (0 through 5) reserved for the distribution of data inside the page – these are also the slots of the page. This situation is illustrated in Fig. 3, where the following notation inside pointer pages is used:

\[ s_i \rightarrow p_s, s_o \]

\[ s_i : \text{pointer page slot (0 through 4)} \]

\[ s_o : \text{data page slot (0 through 5)} \]

\[ c : \text{counter} \]

\[ p : \text{data page number} \]

Every time a new OID is requested, an available slot in a pointer page is delivered to the system. The availability of the slots may be implemented by using a bitmap technique that indicates which of the slots on a pointer page are not in use. For example, a bitmap sequence of 11000 indicates that the first two slots are busy, and the other ones are available.

In Fig. 3, the OIDs are in the order of page, slot, and counter. The page parameter represents the pointer page. The slot parameter (0 through 4) represents one of the slots inside the pointer page. The counter parameter is used as a version mechanism that is intended for reuse of the slot in case the object it points to is deleted from the database.

Suppose that the object o4 needs to be accessed. In Fig. 3 it is possible to see that the OID is (16, 3, 0), and this gives direct instructions of its pointer physical location, that is, the pointer of the object is located at page 16 and slot 3. Looking at page 16 and slot 3, the system confirms that the counter of the OID (which is 0 in this case) corresponds to the counter stored in its respective slot, and this means that the object is valid. So, the system goes on and gets the page indicated by the pointer: page 20. As soon as the page is in memory, slot 5 will be queried to see where on the page the requested object is located.
The deletion process is done in a way that the slots of the data and pointer pages become free and available for reuse. At the same time, all the references to the object are logically disconnected. In Fig. 3, the object o4 is referenced by the object o2, this way, the deletion of it must free its position in the data page and also in the pointer page, and somehow all the references to it must be notified that the object no longer exists anymore. Fig. 4 illustrates this situation.

During the deletion of the object o4, the slot of its OID has its counter parameter incremented by one, as shown in Fig. 4. By doing this, all the objects that were previously referencing the object o4 will continue to reference it, but this time, when the system tries to get it through its old OID (16, 3, 0), the page 16 and slot 3 will be queried and the counter parameter of the OID (zero for the deleted object o4) will no more be equal to the counter parameter in the slot (now it is 1), meaning that the object has been deleted [4]. This is an important mechanism because, for the deletion process, the slot of a pointer page might be reused, the space used by the object in the data page also becomes available, and all the objects in the database that are referencing the object being deleted need not be modified because the counter parameter invalidates each reference automatically.

Compared to the Hashing and B+ Tree techniques, the Direct Mapping technique needs only one access to disc to get the physical address of an object and does not depend on the size of the database [4].

IV. NuGeM

NuGeM (Multimedia Data Manager Kernel) is an evolution of NuGO (Object Manager Kernel), which has been built based on SIRIUS data model [3]. The infrastructure of NuGeM has been designed for the construction of a non-conventional database management system, although the semantic tier of it can be programmed for a conventional one [3]. In this section we analyze how NuGeM does the management of OIDs through its Indirect Mapping technique.

NuGeM uses a logical approach for dealing with OIDs, and like the Direct Mapping technique, it controls the reutilization of a slot that has been used by an object. However, instead of using a physical page indication to construct OID numbers, NuGeM uses surrogate numbers, a mathematical formula, and an indirection table to find where on disc the required data is located. The format of an OID in NuGeM is: (number, counter). While the counter parameter is used to control the reutilization of slots, the number parameter indicates, by using the formula, the logical address in the indirection table and also the slot in the page indicated by the indirection table.

Consider a sample database in which each page has five slots (0 through 4) available for the distribution of data inside the page. This sample database is illustrated in Fig. 5.

To demonstrate how the process of finding an object works in NuGeM architecture, consider an object o2 and its OID being (48, 7), as shown in Fig. 5. To find this object on disc, the first thing to do is locate where on the indirection table it is represented. Since its OID number is 48, and since the sample database has 5 slots per page, an integer division of 48 by 5 will give a result of 9 and a remainder of 3, and this indicates that the object o2 is represented in the indirection table by the logical address 9. Looking at the column “Logical Address” in the indirection table, the corresponding 9 index has the column “Page on disc” defined as 19, and that means that the page 19 is the page that contains the required object: o2 in this case. But where on the page is it? That’s when the remainder of the previous integer division, 3, comes into place, which means that the slot 3 of the page will have the object. Here is the following notation for slots in a page:

\[ s \in \{0, 1, 2, 3, 4\} \]
\[ c \in \mathbb{N} \]

Once the slot has been identified and accessed, its counter parameter (7 in this case) is checked against the counter parameter of the requested object OID (also 7 in this case), and if they are the same, the object is considered valid and the access to it continues by following its internal pointers, that is, an access to its data (d2 in this case) is requested and page 21 is brought into memory.

Since the indirection table is implemented as an in-memory vector, NuGeM is capable of achieving almost the same performance as the physical approach, and at the same time, allowing the relocation of all pages in the database. The relocation of all pages is possible because the OIDs are not directly related to physical pages on disc; for example, the relocation of page 19 to page 20 can be made by just copying the data in page 19 to page 20, and changing the value 19 in the column “Page on disc” of the indirection table to the value 20. By doing this, all the OID numbers ranging from 45 to 49 will now be mapped to page 20.

When an object is deleted from the database, its slot must also be released. This is done by incrementing by one the slot counter parameter and marking it as available for reuse. Suppose that the object o2 is deleted from the database. This situation is illustrated in Fig. 6.
From now on, all references to the object o2 becomes invalid because the counter parameter of its slot is not equal to 7 anymore, so if the object o1 tries to reference it through its data d1,1, it will do it through its old OID (48, 7), and all the process described previously to access the object will be done, but, when the time comes to check the object counter parameter against the slot counter parameter, the system will verify that they are not equal anymore (7 ≠ 8), stating that the object is not valid.

One important thing happens when the last object in a page is deleted. Consider that the object o3 is also deleted from the database. Besides having its slot counter parameter incremented by one, since it is the last object that occupies page 19 the NuGeM architecture releases the whole page, but before doing that, the maximum counter value used in the page must be identified and saved in the header of the page. Also, the respective “OID Counter” column in the indirection table must be updated with this value. This way, NuGeM ensures the uniqueness of OIDs in the whole database by filling all the slots in a new object page recently allocated with the value in the column “OID Counter” of its respective logical address.

However, the identification and filling of a maximum value in object pages have a performance penalty, since these tasks are done through iterative processes. Table I shows the average of logical entries in a page according to its size.

According to the table, if a page size of 4 KB is used in a database, every time an object page is released by the system, an iterative task of about 90 steps is done to get the maximum value of the counters in the page, and another iterative task of about 90 steps must also be done if a new object page needs to be allocated – this second task is the filling process.

To avoid this performance penalty, the indirection table was changed in a way that the identification and the filling processes are no longer necessary.

<table>
<thead>
<tr>
<th>Page size (KB)</th>
<th>Average of logical entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>181</td>
</tr>
</tbody>
</table>
The Hashing technique compared to other techniques as follows: if objects are allocated and deleted from a specific slot one thousand times a day, it would take about 11 thousand years to invalidate the use of the slot.

Indirect Mapping technique used by NuGeM has a slightly higher mapping time than the physical approach, because of the use of an indirection table, it offers the possibility of relocating all pages in the database. The Indirect Mapping technique allows for the relocation of all pages, and its performance is still comparable to the physical approach.

As the old slot counter values are kept in the database, there is no need for the identification and filling processes anymore. Also, the exhaustion of OIDs has been improved at the same time. Considering that the old approach used by NuGeM was to get the maximum counter in slots every time an object page needed to be deleted, and later set all the slots of a new page to this maximum value to ensure the uniqueness of OIDs, it is possible to conclude that there is a considerable loss of the individual capacity of each slot to be reused. This, however, does not happen with the improvement on the Indirect Mapping technique. As an example, knowing that counters are internally stored, now the page with all the counters is stored.

As the old slot counter values are kept in the database, there is no need for the identification and filling processes anymore. Also, the exhaustion of OIDs has been improved at the same time. Considering that the old approach used by NuGeM was to get the maximum counter in slots every time an object page needed to be deleted, and later set all the slots of a new page to this maximum value to ensure the uniqueness of OIDs, it is possible to conclude that there is a considerable loss of the individual capacity of each slot to be reused. This, however, does not happen with the improvement on the Indirect Mapping technique. As an example, knowing that counters are internally implemented as 32-bit fields in NuGeM, if objects are allocated and deleted from a specific slot one thousand times a day, it would take about 11 thousand years to invalidate the use of the slot.

The Indirect Mapping technique used by NuGeM is compared to other techniques as follows:

- **Mapping time:** B+ Trees have a logarithmic mapping time, while Hashing, Direct Mapping and the Indirect Mapping techniques have a constant mapping time. In Hashing, however, collisions might occur. Although the Indirect Mapping technique has a slightly higher mapping time than the physical approach, because of the use of an indirection table, it offers the possibility of relocating all pages in the database.
- **Memory use:** Only B+ Tree and Indirect Mapping use structures in memory to control the mapping mechanism. Considering 32-bit OIDs and 32-bit node pointers in a B+ Tree structure, with pages of 4 KB there are 31.5 millions of OIDs and an in-memory B+ Tree structure of about 240 MB, while the Indirect Mapping technique uses only 5.34 MB (45 times less).
- **Scalability:** An important fact for the Hashing technique is the previous knowledge of the size of the database, which is usually difficult to know. As the database grows, the B+ Tree structure also grows in memory, considering that at least part of it is being managed in memory. The Direct Mapping technique has no impact as the database becomes larger, but the relocation of pointer pages is not possible. The Indirect Mapping technique allows for the relocation of all pages, and its performance is still comparable to the physical approach.
- **Physical space utilization:** The Hashing technique tends to occupy 69% of the pages, while other techniques are capable of using about 100% of them.
- **Concurrency and recovery control:** With the Direct Mapping technique, Log recovery system and a fine-grained concurrency control can be used. NuGeM uses a recovery mechanism based on shadow pages.
- **Relocation:** The relocation of pages is an essential feature when dealing with clustering and compression of databases [13, 14]. Only B+ Tree and Indirect Mapping techniques allow for the relocation of all pages, but the latter uses less memory and has a constant mapping time.

In order to verify the efficiency of the improvement in the Indirect Mapping technique used by NuGeM, a test with the following characteristics was conducted:

- A database with a 4 KB page size was created.
- 31.5 million OIDs were generated.
- 90 objects per page were allocated.

Table II shows the result of testing some operations against this database. The operations that were considered are the creation, write, read, and deletion of object pages. The commit and abort functions were also tested. The total elapsed time and the mapping time of OIDs for each operation is also presented.

It is also necessary to consider the time that NuGeM spent when dealing with the process of managing the columns “Page on disc” and “OID Page” in the indirection table; for the test case it was computed as being 0.359 seconds.

Therefore, the total elapsed time for the operations was 46,983 seconds, and part of this time (0.546 seconds) was for the management of OIDs, as shown in Table II. Considering the penalty for the use of an indirection table, 0.359 seconds was added to the computed mapping time in Table II, that is, the total mapping time becomes 0.546 + 0.359 = 0.905 seconds, which corresponds to 1.926% of the total elapsed time.

The auxiliary structures used in memory by NuGeM have taken up about 5.34 MB. This way, with low memory utilization (5.34 MB), and a performance penalty of only 1.926%, it is possible to achieve almost the same performance as with a direct mapping technique, and at the same time have total flexibility for the relocation of all pages in the database.
This article presented the main concepts involving the management of OIDs and compared two approaches: physical and logical. Besides that, were presented some of the techniques that use the logical approach, including the Indirect Mapping technique, which is used by the infrastructure of NuGeM. Finally, an improvement to this technique was proposed.

Although the Indirect Mapping technique used by NuGeM has a constant mapping time, uses less memory than the B+ Tree technique, has no problems like collisions, and allows total flexibility for the relocation of all pages in the database, there is a performance penalty associated to the process of identification and filling of a maximum OID during the deallocation and allocation of OID pages.

The improvement stated in this paper considered this performance penalty and solved it by changing the way NuGeM uses the column “OID Counter” in its indirect table in memory. By substituting this column for a “OID Page” column, the improvement eliminated the need for the identification and filling process of a maximum OID. Also, the exhaustion of OIDs has been improved at the same time, since now every time an OID page is deallocated, its OIDs are kept for future utilization.

For all these reasons, the achieved result testifies that with a performance penalty of only 1.926% and a low utilization of memory, the Indirect Mapping technique used by NuGeM together with the presented improvement is a viable alternative to direct mapping.

This paper presented an original contribution through the improvement of Indirect Mapping. Although the work was done considering the NuGeM architecture, the idea behind the improvement is applicable to other OODBMSs.

ACKNOWLEDGMENT

Project financed by FAPESP.

TABLE II. ELAPSED TIME FOR THE PROCESSING OF 31.5 MILLION OBJECTS IN A 4 KB DATABASE PAGE SIZE

<table>
<thead>
<tr>
<th>Function</th>
<th>Total time (seconds)</th>
<th>Mapping time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation</td>
<td>17.566</td>
<td>0.038</td>
</tr>
<tr>
<td>Write</td>
<td>15.442</td>
<td>0.169</td>
</tr>
<tr>
<td>Read</td>
<td>3.196</td>
<td>0.152</td>
</tr>
<tr>
<td>Deletion</td>
<td>1.330</td>
<td>0.031</td>
</tr>
<tr>
<td>Commit</td>
<td>6.036</td>
<td>0.065</td>
</tr>
<tr>
<td>Abort</td>
<td>3.413</td>
<td>0.091</td>
</tr>
<tr>
<td>Total</td>
<td>46.983</td>
<td>0.546</td>
</tr>
</tbody>
</table>

REFERENCES


