The Scalability of an Object Descriptor Architecture OODBMS

Kwok K. Yu, Byung S. Lee, Michael R. Olson
University of St. Thomas, St. Paul, Minnesota, U.S.A.
kkyu@gps.stthomas.edu, bslee@stthomas.edu, michael.olson@westgroup.com

Abstract

An object database management system (OODBMS) has been often criticized for its alleged insufficient scalability for a large-scale production system. We investigated the scalability issue on a commercial OODBMS with a focus on the scalability with respect to the number of objects. Our approach was a benchmark experiment using the loading and indexing of SGML text documents as an application. The application was characterized by its small granularity of objects, which resulted in a huge number of objects in order to make a large database volume. The OODBMS we used was built in a so-called "object descriptor architecture (ODA)" as opposed to a "virtual memory mapping architecture (VMMA)". The results showed that the OODBMS scaled better than we had anticipated. It required, however, algorithmic resolutions to overcome the shortage of object cache space. Three key resolutions were made. First, we created indexes in fragments by committing a loading transaction before the object cache space became full, and subsequently merged the fragments into one master index. Secondly, we had the application release cached object descriptors (CODs) as soon as they became unnecessary. Thirdly, we utilized a query cursor mechanism to fetch the objects returned from a query piece by piece without overflowing the object cache space. Currently we are attempting to push the scalability up to filling up the maximum available hard disk space.

1. Introduction

In a production world, an object-oriented database management system (OODBMS) has been subject to a criticism for its alleged lack of scalability, especially with respect to data volume. We wanted to see it for ourselves by actually experimenting it on a commercial OODBMS. Our goal was to understand if there was any generic reason for an OODBMS not being scalable. Knowing that there are two existing representative architectures of an OODBMS — one called a virtual memory mapping architecture (VMMA) and the other object descriptor architecture (ODA) [2] — we deliberately picked one commercial product for each of the two architectures. The VMMA OODBMS has proven to be not scalable and a report was made in [1]. We continued with an ODA OODBMS and reaped interesting results, which we report in this paper.

Our method of the experiment was to conduct a benchmark testing on a commercial OODBMS. For a typical OODB application a large data volume means a large number of objects because objects are usually small in size (i.e., 10s to 100s of bytes). Therefore we needed an application that handled a large number of small objects than a small number of large objects. To this end, a Standard Generalized Mark-up Language (SGML) [6] document loading utility, which was used in [1], was reused as our application. Besides, our SGML application has its own practical significance in the sense that a simplified subset of SGML called XML [4, 5] is likely to be the predominant data format for web-enabled content applications.

We would like to claim the following points as our contributions to the research community:

- We verified that an ODA OODBMS scales far better than a VMMA OODBMS, at least as far as we are concerned with the scalability with respect to the number of objects (hence data volume).
- We confirmed that the number of objects was a more critical barrier to the scalability than the sheer data volume on an ODA OODBMS.
- We found that it was crucial to the scalability of an ODA OODBMS that an application is capable of cleaning up "in time" unnecessary cached object descriptors (CODs) from an object cache space.

The rest of this paper is organized as follows. In Section 2, we provide background information about ODA OODBMS and SGML documents. In Section 3, we describe the benchmark application including its object schema, how the loading utility works, and more specifics of index creation algorithms. The benchmark results are presented in Section 4, and finally conclusion follows in Section 5.

2. Background

In this section, we provide some background knowledge that is necessary to understand the rest of the paper.

2.1. Object Descriptor Architecture

As mentioned in the introduction, there are two different representative architectures of an OODBMS. [2] One is called a virtual memory mapping architecture (VMMA) and the other object descriptor architecture (ODA). VMMA and ODA employ distinct schemes of managing an object cache space during the execution of an application. More specifically, in a VMMA database address space is an extension of virtual memory space just as virtual memory space is an extension of real memory (RAM) space (See Figure 1), whereas in an ODA the system addresses cached objects indirectly via an object descriptor. Figure 3 and Figure 4 illustrate the mechanism of fetching and caching objects in an ODA for a sample application code shown in Figure 2. (The sample code is written in ODMG C++ [3].)

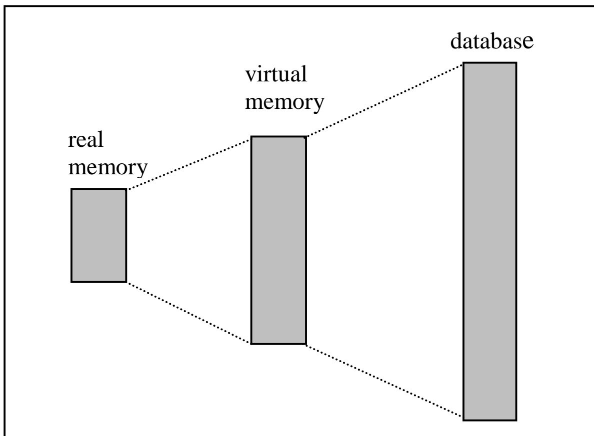

Figure 1. Address mapping in VMMA

```
class A: public d_Object {
public:
  d_Ref<B> b;
}

class B: public d_Object {
public:
  d_Ref<C> c;
}

main() {
   ...
   d_Ref<A> a1;
   // Get a1 from the database.
   d_oql_execute(query, a1);
   // Navigate to b and get b1.
   d_Ref<B> b1 = a1->b;
   // Navigate to c and get c1.
   d_Ref<C> c1 = b1->c;
   ...
}
```

Figure 2. A sample code of fetching objects

The ODA is characterized as follows.

- The object cache space of ODA is configured in two tiers -- object buffer and page buffer. (This is called a "dual buffering.")
- When an object is fetched into an object buffer, a cached object descriptor (COD) is allocated in the cache space for each reference that comes out of the object and the COD is entered in a COD table. (Note that the referenced object is not fetched until it is actually needed by the application. This is called a "deferred pointer swizzling.")

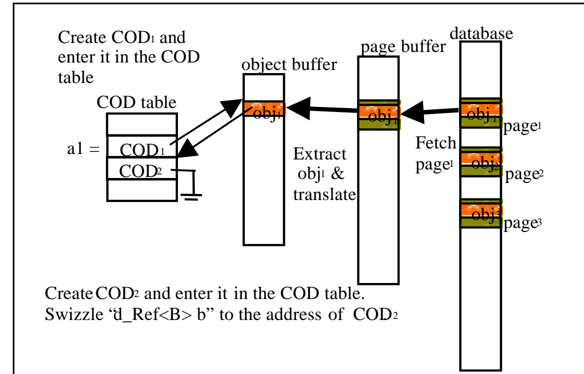

Figure 3. After d_oql_execute(query, a1);

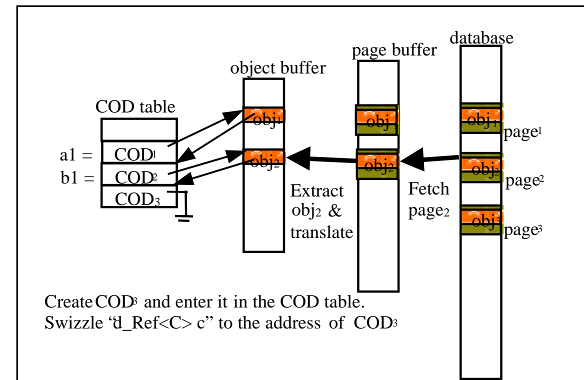

Figure 4. After d_Ref<B> b1 = a1->b;

Here comes a brief description of the object caching algorithm in ODA:

1. Locate the page containing a target object in the database and retrieve it into a page buffer.
2. Extract the target object from the page, translate it from a disk format to a main memory format, and place it in an object buffer.
3. Create a COD for the fetched object and enter it in a COD table.
4. For each external object reference in the fetched object, create a new COD and enter it in the COD table.

Object caching is the bottleneck of the scalability we are investigating because an object cache space is

allocated in a virtual memory and its size is limited. The system is "choked" if the virtual memory becomes full and no memory can be released from it.

2.2. SGML

Standard Generalized Markup Language (SGML) [7] is an international standard of a mark-up language for *describing* a document and is supported by the International Standardization Organization (ISO) and Department of Defense (DoD). The key idea is to separate document form (e.g., document structure, character font and size) and content (i.e., actual text). This separation facilitates the interchange of documents among heterogeneous platforms. An SGML document involves the following three components: an SGML declaration, a document type definition (DTD), and a document instance.

- The SGML declaration defines the coding scheme (e.g., characters and delimiters) of a document.
- The document type definition (DTD) defines the rules for marking up a class of documents. There are different DTDs for different document types such as letter, memo, and article.
- The document instance is a marked-up text document itself.

Figure 5 and Figure 6 show the examples of an SGML DTD and a document instance that complies with the DTD, respectively. Note the nested element tags (e.g., book, chapter, section, title, paragraph, xref), attributes (e.g., targref = P1, targid = P1), and an entity (e.g., SGML). The grammar specified in the exemplary DTD can be shown using an informal BNF notation as shown below. (`{symbol}` denotes a sequence of zero or more `symbols`. PCDATA is a parsed character data string.)

- book ::= title chapter {chapter}
- chapter ::= title section {section}
- section ::= title paragraph {paragraph}
- title ::= PCDATA
- paragraph ::= PCDATA
- xref ::= PCDATA (can appear anywhere)

As we see in the example, an SGML element can contain other elements nested in it, thus a perfect fit for a composite object. When SGML elements are stored as OODB objects, each element is assigned with its own object identity (OID), which can be used to implement references between elements without introducing artificial key values. What was particularly interesting about SGML for our benchmark experiment was the small size of elements. This forced us to create millions of SGML elements in order to make a large data volume.

```
<!DOCTYPE BOOK [ <!-- Note that “<!-- “ and “ -->“ are
comment delimiters -->

<!-- Defines a text that will replace &SGML in a document
instance. -->
    <!ENTITY SGML "Standard Generalized Markup
Language">

<!-- Typically, the first element name matches the DOCTYPE
name -->
    <!ELEMENT book  - -  (title, (chapter)+)  +(xref)>

<!-- The “-O” means the chapter element must have a start tag,
but that the end tag is optional. We can do this because the
document structure implies the presence of the end tag, whether
it is present or not. In other words, a chapter may end when the
end tag is reached, when the start of another chapter is found, or
when the end tag of the book element is reached. -->
    <!ELEMENT chapter   - O  (title, (section)+)>

    <!-- The plus sign (+) means the item must occur at least
one time and perhaps many times -->
    <!ELEMENT section   - O  (title, (paragraph)+)>

    <!-- The start and end tags for a title are optional -->
    <!ELEMENT title   O O  (#PCDATA)>

    <!ELEMENT paragraph  - O  (#PCDATA)>
    <!ATTLIST paragraph  targid  ID  #REQUIRED>

    <!-- Both the start and end tags for xref are required. -->
    <!ELEMENT xref  - -  (#PCDATA)>
    <!ATTLIST xref  targref  IDREF  #REQUIRED>
]>
```

Figure 5. An example of SGML DTD

```
<book>This title needs no begin or end tags.
<chapter><title>Here we chose to insert a begin tag without an
end tag.
<section>This time we include only the end tag of a title.</title>
<paragraph targid=P1>Notice that the xref may appear
anywhere inside the book element. This is allowed because of
the “+(xref)” inclusion in the content model for the book
element. The “P1” targref matches the “P1” targid, thereby
establishing a logical hypertext link. For example, <xref
targref=P2> link </xref>.
<paragraph targid=P2>Also notice that each paragraph is
required to have a unique targid. This is because of the type
specification of “ID” and the “#REQUIRED” in the content
model of the paragraph declaration.
<paragraph targid=P3> This was an example of an SGML
document instance. SGML stands for &SGML.</paragraph>
</book>
```

Figure 6. An example of SGML document instance compliant with the DTD in Figure 5.

Table 1. Classes used by the benchmark application.

Class	Description
DocComponent	An abstract class that *SGMLElement*, *SGMLEntity*, and *PCDATA* inherit from.
Document	Represents a document, which may contain an *SGMLElement* that is the outermost element.
PCDATA	Represents a character string data (i.e., content word) in an SGML document. Derived from *DocComponent*.
Publication	Represents a publication, which may contain *Document*s.
PubSeries	Represents a publication series, which may contain *Publication*s.
SGMLAttribute	Represents an attribute in SGML.
SGMLEntity	Represents an entity in SGML. Derived from *DocComponent*.
SGMLElement	Represents an element in SGML. Derived from *DocComponent*. May contain parts such as *SGMLEntity*s, *PCDATA*s, *SGMLAttribute*s, and other *SGMLElement*s.
WordDictionary	An index class that contains a dictionary with string as the key and *WordLocator* as the value.
WordLocator	Contains a key string and a set of links to all *SGMLElement*s that contain the key.

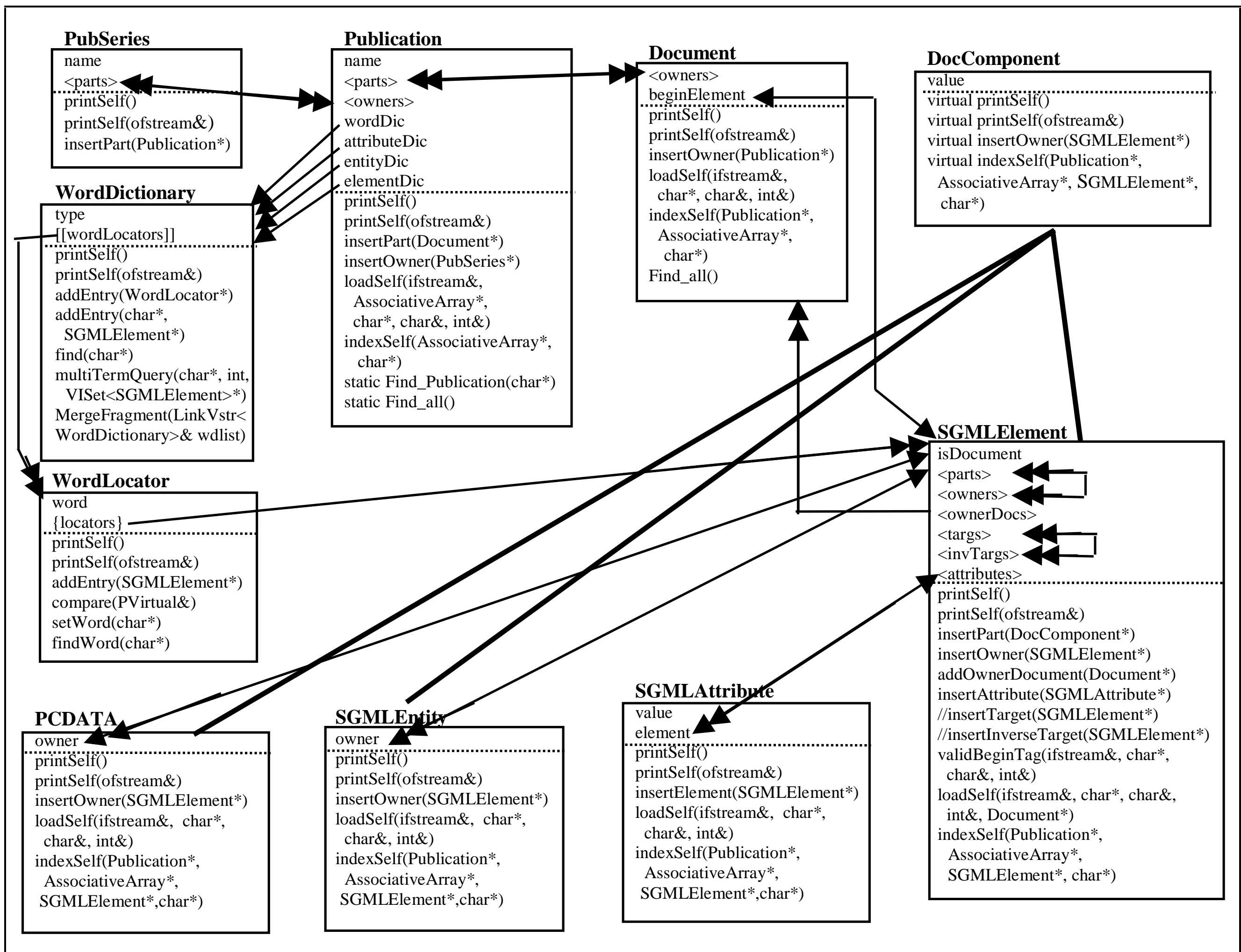

Figure 7. Benchmark Application Object Schema Diagram.

3. Benchmark Application

The benchmark application is basically a loading utility that creates SGML objects (e.g., elements, attributes, entities, and PCDATAs) and indexing SGML documents by those SGML objects. The application program simulates loading a large SGML text file by repeatedly loading a small SGML text file. It prompts users for an input file name and the number of times to repeat loading the file. It dumps status information to a file during and after the loading. If a crash occurs, the output file will contain an error message. Otherwise the output file will contain information such as the number of created SGML objects (i.e., elements, entities, attributes, and PCDATA's) and the time it took to create and index them. For the rest of this section, we give a brief overview of the object schema of the benchmark application in Section 3.1, explain how the loading utility works using an example in Section 3.2, and describe an index fragmentation and merging algorithm of the loading utility in Section 3.3.

3.1. Object Schema

Table 1 describes all the classes used by the benchmark application. Figure 7 is the object schema diagram that shows the relationships between these classes.

3.2. Overview of the loading utility

The loading utility is used to load the content of an SGML encoded ASCII stream to a user specified publication. It recognizes the basic building blocks of SGML (e.g. elements, attributes, entities, and PCDATA's) and creates them as objects in a database. More specifically, when a new SGML element is encountered, the loading utility allocates persistent storage space in the database for the element, its attributes, and component elements and their attributes. Once the creation of objects is finished, the element objects are indexed by an element index, an entity index, an attribute index, and a PCDATA index.

3.2.1. Creating SGML object

Let us look at an example to see how it works. Suppose we load the following SGML document:

<doc>
<body id=123>
Merry &XMAS
</body>
</doc>

Figure 8 shows the SGML objects and their linkage after the loading is completed. When the first line (<doc>) is loaded, the loading utility creates an element object in the object cache, identified by a value 'doc'. When the second line (<body id=123>) is loaded, the loading utility creates one more element object identified by a value 'body' and also creates an attribute object identified by a value 'id=123'. Since <body> is a sub-element of <doc>, the <body> element object is referenced in the 'parts' list of the <doc> element object. In addition, 'id=123' is an attribute of element <body>, so the attribute object is referenced in the 'attributes' list of the <body> element object. When line 3 (Merry &XMAS) is loaded, the loading utility creates a PCDATA object identified by a value 'Merry' and an entity object identified by a value 'XMAS.' Since the PCDATA 'Merry' and the entity 'XMAS' are parts of the <body> element, they are both referenced in the parts list of the <body> element. Line 4 (</body>) and line 5 (</doc>) signify the end of the element tags, and the creation of SGML objects are done.

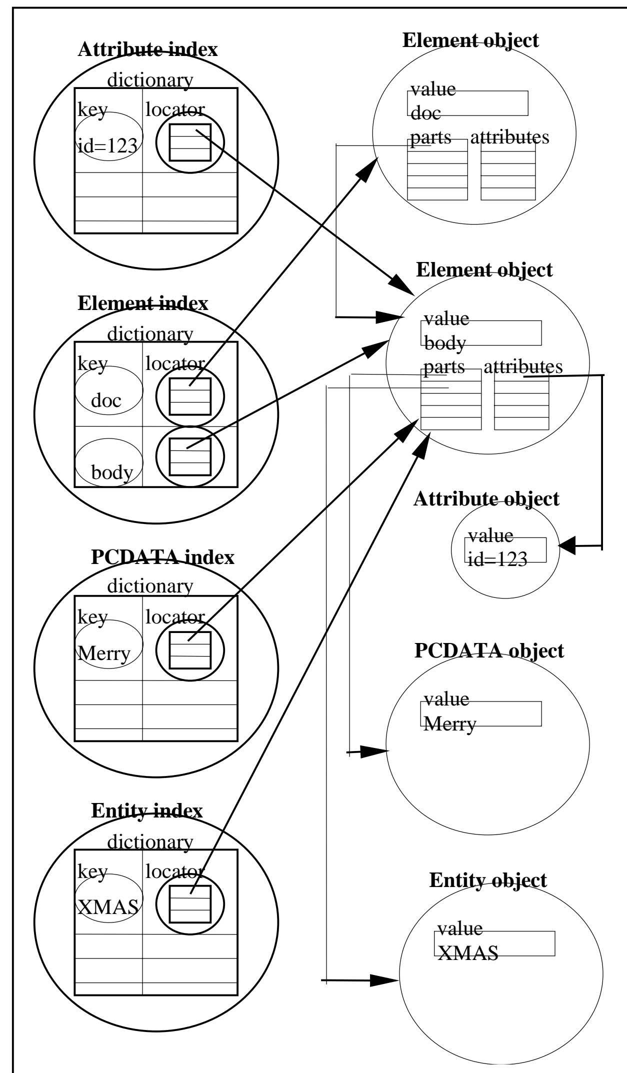

Figure 8. After the loading is done.

3.2.2. Indexing SGML objects.

Figure 8 also shows how it looks when the indexes are created. Since the attribute object 'id=123', PCDATA object 'Merry', and entity object 'XMAS' are parts of the element <body>, their corresponding indexes all reference the <body> element object.

Each index has a dictionary data structure. A dictionary is a collection object in which each entry has a key field and a value field. The key field of our index object is a string object. The value of the string depends on the type of an index. For instance, the PCDATA index of our example has an entry with a key of 'Merry' string object. The value field contains a locator object. A locator object has a set data structure that holds all the links to those element objects that contain the key.

3.3. Index fragmentation and merging

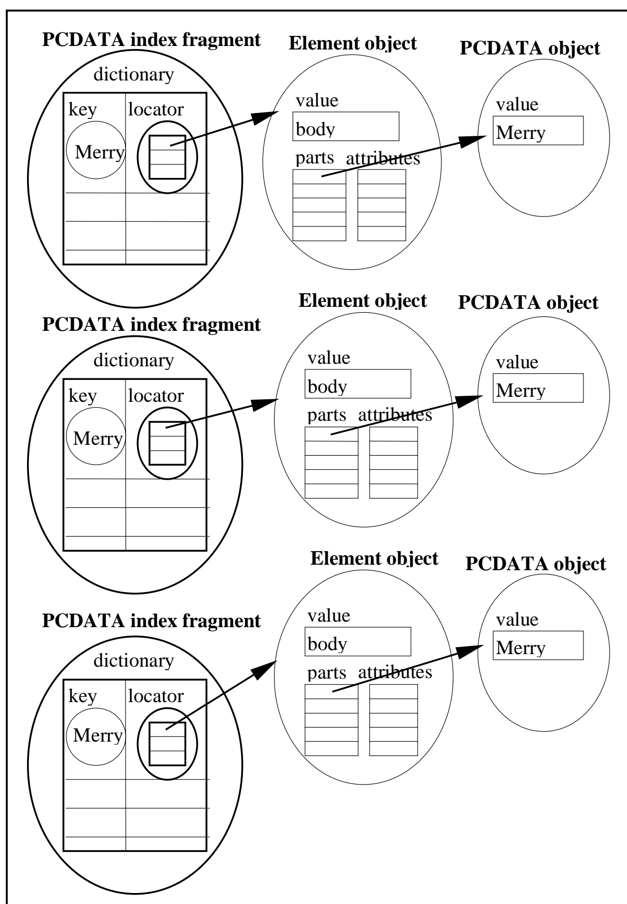

Figure 9. PCDATA index fragments before merging

The loading algorithm can be summarized to the following three steps.

1. Parse the input file and create SGML objects.
2. Create fragmented indexes and index the SGML objects.
3. Merge the index fragments into a master index.

The rationale behind the index fragmentation and merging is that SGML objects are created in the object cache of a limited size. The cache will run out of space if there are too many objects created within the same transaction. The resolution to this problem is to commit the transaction more often to flush out SGML objects and indexes to disk. In our benchmark, we committed a transaction every time a *fixed* threshold number of objects were created. (We thought about determining the threshold number dynamically by monitoring the use of cache space but did not implement it.) Note that the index is fragmented as the result of committing the loading transaction in intermediate stages. The index fragments are merged into one master index in the subsequent step.

Figure 9 shows the state of the PCDATA index prior to merging after our exemplary input file has been loaded three times with a commit after each loading. Figure 10 shows the state of the PCDATA index after the fragments are merged into one master index.

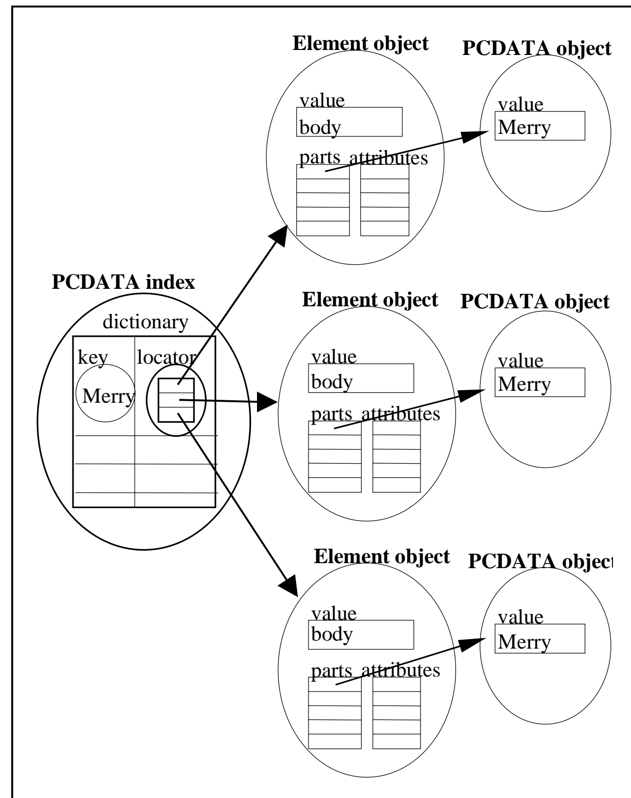

Figure 10. PCDATA master index after merging

Here comes a summary of the index merge algorithm:

```
Create an empty master index.
For each index fragment begin
    While the current index fragment is not
    empty begin
```

```
        Get the key of the first entry from the
            current index.
        Find entries with the same key from all
            index fragments.
        Retrieve all WordLocator objects from
            the found entries.
        Consolidate the retrieved WordLocator
            objects by merging their links to
            SGMLElement objects.
        Insert the key and the consolidated
            WordLocator object into the master
            index.
        Remove the entries with the same key
            from all index fragments.
    End While
    Delete the current index fragment from the
       database.
End For
```

4. Benchmark Result

4.1. Benchmark environment

The benchmark application, including the load utility program, was written in C++ and compiled on a Sun workstation. We initially used SunSparc10 running Solaris 2.5.1 with 224 Mbytes of RAM and 520 Mbytes of swap space. Later we migrated to another machine Ultra-1 Sparc running Sun Solaris 2.5.1, which was of a lesser capacity but was available entirely for the benchmark experiment. The new machine had 128 Mbytes of RAM with 550 Mbytes of swap space and 2 Gbytes of disk space. Eventually we increased the swap space to 832 Mbytes and added more disk space toward a total of 7.25 Gbytes.

Due to a difficulty in obtaining a large volume of real SGML document files, we loaded a small SGML file (25,320 bytes) repeatedly to simulate a large data volume. Since each SGML element in the file is created as one SGML element in the database even though its content is a duplicate of another, we are indeed simulating a large number of SGML objects. However, the repetition of the same file skews the distribution of indexed entries, hence generating the worst case scenario of indexing.

4.2. Problems and resolutions

We performed the experiment while increasing the size of input SGML files from 50M to 100 Mbytes, 250 Mbytes, 500 Mbytes, and 1 Gbytes. We ran into an "*out of heap memory*" error almost every time we increased the size. The error was caused by the object cache space overflowing with too many SGML objects (e.g., elements, attributes, entities, PCDATA's) and index objects (e.g., WordLocator), and cached object descriptors (CODs). Here comes a summary of what we have done in order to resolve the "out of heap memory" problem.

- Commit the loading transaction before the object cache becomes full. A commit releases objects from the cache space. (As mentioned earlier, this renders the indexes to be fragmented and thus necessitates merging them into one master index.)
- Release unused CODs from the cache space by calling a system function. This COD release operation should be used with care to avoid leaving invalid links in the object cache space. (The OODBMS we used does not do it automatically at commit. It would be far better if the OODBMS traced the reference count of each COD and purged out the COD at commit time if its reference count is zero.)
- Utilize a query cursor mechanism to deal with a too large query result. The cursor mechanism allows the application to control how many objects to return from a query at a time. Therefore if a huge number of objects are returned, a cursor can be used to control the number of objects brought into an object cache so that they can fit in without filling up the cache space.

Another error we ran into frequently was "*out of database volume*" error. This error occurred when the allocated database volume was filled up. We added more volumes manually every time the error occurred. This was not a generic problem but rather a deficient feature of the OODBMS we used. We wished the product supported an automatic expansion of database volume.

Details of all the problems encountered throughout the benchmark testing and their resolutions are described in [8].

4.3. Most recent performance data

After overcoming many system troubles, we were able to load a total of 500 Mbytes of SGML text file into a 2.4 Gbyte database successfully so far. In contrast, the maximum database size on a VMMA OODBMS was 250 Mbytes (on a 32-bit machine). [1] Note that the expansion ratio (i.e., loaded database size / raw data file size) is less than 5. This ratio is much better than what we observed on the VMMA OODBMS in [1]. It was more than 10 there.

We are in the process of loading 1Gbyte SGML file. Our first attempt failed because a query retrieved too many objects in the beginning of merging index fragments. When the loading utility merges index fragments, it issues a query to retrieve all WordLocator objects that contain the links to SGMLElements that contain the current key string. Since we are simulating large sized files by repeatedly loading the same SGML document, the query returns too many matching WordLocator objects to fit in the object cache. The situation would be alleviated if we were to use real data,

where the frequency of repeated words would not be so high.

The query cursor mechanism was utilized in order to resolve the problem of too large query result. The implementation of the cursor mechanism has been completed. The four different kinds of indexes are merged in the order of entity, attribute, element, and PCDATA, in an increasing order of the number of indexed objects. At this time of writing, the index merge fails with the "out of heap memory" error in the middle of merging element index fragments after successfully merging the first two. We suspect the resolution is to locate safe points of releasing CODs in the benchmark program. (By a "safe COD release point", we mean a point in the program where we can find objects whose CODs can be released from the object cache space without leaving dangling links from the objects.) We are still working on it.

What we desire is to reach the physical limit of our benchmark experiment by getting an "out of disk space" error after filling up all the available hard disk space. We will be able to reach this physical limit if we succeed in finding the safe point of releasing CODs.

4.4. Summary of our benchmark result

Here comes the summary of benchmark testing result obtained so far, as already briefed in the introduction of this paper.

- The "lack of scalability" labeled on an OODBMS is not always valid for the scalability of an ODA OODBMS with respect to the number of objects (hence data volume as well).
- The number of objects is a more critical barrier to the scalability than the data volume itself. Most problems that occurred were caused by insufficient object cache space that overflowed when an excessive number of objects and CODs were created.
- It is crucial to the scalability when millions of objects are dealt with that the application program is capable of releasing unused CODs before they overflow the object cache. The ultimate scalability depends on the feasibility of finding safe COD release points in the application program.

5. Conclusion

We conducted a benchmark experiment using a commercial object-oriented database management system built in object descriptor architecture. The goal of the benchmark was to see how scalable it was with respect to the number of objects that are loaded into a database. The application chosen for the experiment was an SGML document loading program, which was characterized by a huge number of small objects. The benchmark results showed that it scaled far better than the same benchmark performed on virtual memory mapping architecture previously. We found it most important that the application should be capable of releasing unused cached object descriptors (CODs) from the object cache before the cache space is filled up.

As to further work, we will first finish the on-going benchmark experiment by continuing our efforts of locating safe COD release points in our SGML loading program. Whether we be successful or not, the next step will be to try with one large (> 1 Gbytes) SGML text file, not a small file repeatedly loaded. Lastly, our experiment to date has been confined within a centralized OODBMS platform. We will extend the benchmark to a distributed OODBMS platform.

References

[1] M.R. Olson and B.S. Lee, "Object Databases for SGML Document Management", *Proceedings of the 30th Hawaiian International Conference on Systems Sciences (HICSS)*, Volume III, Maui, Hawaii, U.S.A., January 7-11, 1997, pp. 39-48.

[2] M.E. S. Loomis, *Object Databases: The Essentials*, Addison-Wesley Publishing Company, 1995, pp. 117 - 129.

[3] R.G.G. Cattell, et al. (ed.), *The Object Database Standard: ODMG2.0*, The Mogan Kaufmann Publishers, Inc., 1997.

[4] J. Bosak, "XML, Java, and the future of the Web," Sun Microsystems, 1997, http://mealab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm

[5] "The XML Revolution" *Web Matters*, October 1998, http://helix.nature.com/webmatters/xml.html

[6] "What is SGML?" Graphic Communication Association, 1998, http://www.gca.org/stanpub/sgml.htm

[7] E. Herwijnen, *Practical SGML (2nd ed.)*, Kluwer Academic Publishers, 1994.

[8] K.K. Yu, *Loading scalability of an object-oriented database management system in the object descriptor architecture*, Thesis Project Technical Report, Graduate Programs in Software, University of St. Thomas, St Paul, Minnesota, U.S.A., November 1998.