Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs

Wook-Shin Han, Kyu-Young Whang, Yang-Sae Moon
Department of Computer Science and Advanced Information Technology Research Center (AITrc)
Korea Advanced Institute of Science and Technology (KAIST)
373-1, Kusong-Dong, Yusong-Gu, Taejon 305-701, Korea
{wshan,kywhang,ysmoon}@mozart.kaist.ac.kr

Il-Yeol Song
College of Information Science and Technology
Drexel University, Philadelphia, Pennsylvania 19104, USA
songiy@drexel.edu

Abstract

Prefetching is an effective method for minimizing the number of round-trips between the client and the server in database management systems. In this paper, we propose new notions of the type-level access locality and the type-level access pattern. We also formally define the notions of capturing and prefetching to help understand the underlying mechanisms. We then develop an efficient prefetching policy based on these notions and the framework. The type-level access locality is a pheonomenon that repetitive patterns exist in the attributes referenced. The type-level access pattern is a pattern of attributes that are referenced in accessing the objects. Existing prefetching methods are based on object-level or page-level access patterns, which consist of object-ids or page-ids of the objects accessed. However, the drawback of these methods is that they work only when exactly the same objects or pages are accessed repeatedly. In contrast, even though the same objects are not accessed repeatedly, our technique effectively prefetches objects if the same attributes are referenced repeatedly, i.e., if there is type-level access locality. Many navigational applications in Object-Relational Database Management Systems(ORDBMSs) have type-level access locality. Therefore, our technique can be employed in ORDBMSs to effectively reduce the number of round-trips, thereby significantly enhancing the performance. We have conducted extensive experiments in a prototype ORDBMS to show the effectiveness of our algorithm. Experimental results using the OO7 benchmark and a real GIS application show that our technique provides orders of magnitude improvements in round-trips and several factors of improvements in overall performance over on-demand fetching and context-based prefetching, which is a state-of-the-art prefetching method. These results indicate that our approach provides a new paradigm in prefetching that improves performance of navigational applications significantly and is a practical method that can be implemented in commercial ORDBMSs.

1 Introduction

Object-relational database management system(ORDBMS) applications model a set of interrelated objects as complex objects using the reference and the collection attributes. Navigational applications of an ORDBMS navigate complex objects by accessing the component object one by one using these attributes. The navigational characteristics of these applications make the number of round-trips increase in client/server DBMSs and causes serious performance degradation. By placing a cache at the client-side, ORDBMSs minimize round-trips between the client and the server.

Existing object fetching policies are categorized into two: on-demand fetching[3] and prefetching[1, 5, 7, 13, 9, 16]. In on-demand fetching the objects are fetched from the server on request. The advantage of this policy is that it fetches only the objects that are eventually accessed. The disadvantage is that it causes a lot of round-trips since it causes one round-trip to the server per each object fetched. In prefetching, the objects that are expected to be accessed in the future are fetched in advance. Prefetching reduces round-trips and increases the system performance if the objects prefetched are indeed accessed. However, if the objects prefetched are not accessed eventually, the system performance will be get worse due to the overhead of fetching unnecessary objects. Therefore, to prefetch objects effectively, it is important to correctly predict the future access patterns of the applications.

Object-oriented navigational applications typically process objects by starting at some root object and traversing the other objects, connected from the root object, by using the references in the objects[2, 11]. For example in Figure 1, let us consider a navigational application that retrieves the professors' addresses and cars(and their manufacturers, drivetrains, and engines), where the professor's salary is more than $100,000. The application first issues a query to

1063-6382/01 $10.00 © 2001 IEEE

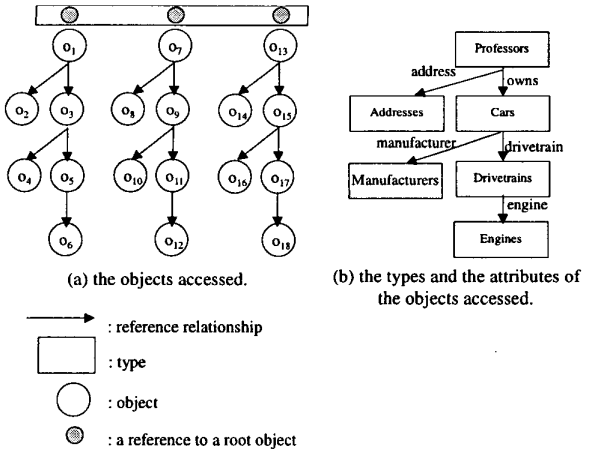


Figure 1. A navigational application to get professors' addresses and cars(and their manufacturers, drivetrains, and engines), where the professor's salary is more than $100,000.

get the references to the root objects: "SELECT * FROM Professors WHERE salary > 100000." Then, it navigates through the related objects to get information about the professors' cars and addresses connected from each root object.

The access patterns of navigational applications can be represented by the patterns of attribute references in accessing the objects rather than the patterns of object references themselves. Figure 1 shows the reference sequences of the objects accessed(*object reference string*) in the previous example. Here, the object reference string is $o_1, o_2, o_3, o_4, \cdots o_{18}$. We note that no repetitive pattern occurs in the object reference string. However, we observe that there is a repetitive pattern of attributes referenced: address, owns, manufacturer, drivetrain, and engine. We define the *type-level access locality* as a phenomenon that repetitive patterns exist in the attributes referenced. We define the *type-level access pattern* as a pattern of the attributes referenced. We provide more formal defitions in Section 3.2. Many navigational applications in ORDBMSs have type-level access locality. when an object pointed by an element reference of a collection is accessed, all the objects(e.g., $o_2 \sim o_n$) pointed by the other element references of the collection are prefetched.

The contributions of this paper are as follows. First, we formally propose new notions of the type-level access locality and the type-level access pattern. Second, we formally define the notions of the process of capturing and the process of prefetching to help understand the detailed mechanisms involved. Third, we propose a new prefetching algorithm that implements these notions. Fourth, to show the effectiveness of the proposed method, we perform extensive experiments and compare the results with those of the on-demand fetching and the context-based prefetching, which is the current state-of-the-art technology. The results show that our method significantly improves the performance compared with these methods.

The rest of the paper is organized as follows. Section 2 explains existing prefetching methods and reviews the advantages and disadvantages of the methods. Section 3 proposes the notion of the type-level access locality and the type-level access pattern. Section 4 presents the concept and implementation of our prefetching algorithm. Finally, Section 5 presents the experimental results, and Section 6 concludes the paper.

2 Related Work

Existing prefetching methods are classified into the following four categories based on the method of selecting the candidate objects to be prefetched: 1) page-based prefetching; 2) object-level/page-level access pattern based prefetching; 3) user-hint based prefetching; and 4) context-based prefetching.

The page-based prefetching method fetches all the objects together in the page containing the object requested[7, 10]. This method works well only when the objects in the same page are accessed consecutively. Otherwise, it loses the benefit of prefetching. Since the effectiveness of this method depends entirely on the clustering and of objects in a page, the method fails to work well when applications do not access objects in the clustered order.

The prefetching method based on object-level/page-level access patterns predicts future object/page access patterns from recent object/page access references[5, 13]. Palmer and Zdonik[13] proposed a prefetching method based on object-level access patterns. Their method captures an object access pattern from object references using a learning algorithm, and then, prefetches the objects based on the captured object pattern. Curewitz et al.[5] proposed a similar algorithm capturing the page-level access patterns using a compression algorithm. The drawbacks of these methods, however, is that they work only when the identical objects or pages are accessed repeatedly.

The user-hint based prefetching method prefetches objects based on hints provided by the user. Chang and Katz[6] proposed a method of prefetching objects based on the the user hints, such as "my prime access is via configuration relationships." Commercial ORDBMSs provide methods based on user-hints similar to this example[11]. However, the drawback of this method is that it relies on the users to provide the hints, putting a big burden on the users. This approach is also against the recent trend of developing auto-tuning DBMSs.

The context-based prefetching method[1], which is the most recent work on the prefetching, fetches all the objects together in the structure context of the object requested. A structure context of an object describes the structure, in which the object was fetched. Examples of structures are

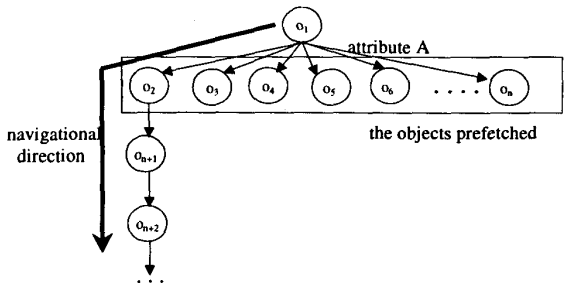


Figure 2. An example of context-based prefetching.

query results and collections. Figure 2 shows an example of objects prefetched in a context-based prefetching method. Here, we assume that the object o_2 is not in the client cache. The structure context of o_2 is the value of the attribute A of o_1. Therefore, when o_2 is accessed, all the objects $o_2 \sim o_n$ in the structure context of o_2 are prefetched. That is, when an object(e.g., o_2) pointed by an element reference of a collection is accessed, all the objects(e.g., $o_2 \sim o_n$) pointed by the other element references of the collection are prefetched. This method is effective when a navigational application traverses an object hierarchy in the bread-first-search(BFS) fashion. This method has been implemented in a commercial DBMS with performance increases of up to 70% over on-demand fetching.

The context-based prefetching method has a problem when an application traverses an object hierarchy in the depth-first-search(DFS) fashion. In this case, objects prefetched can be replaced from the cache even before the objects are actually accessed. For example, suppose that an application traverses the object hierarchy in Figure 2 in the DFS fashion. If a large number of objects (e.g., o_{n+1}, o_{n+2}, $\cdots$) are connected from the object o_2, so as to completely fill the cache, then, the objects prefetched, $o_3 \sim o_n$, will be replaced before being accessed. This problem is even more serious when the size of a structure context is large. Another drawback of this method is that it does not prefetch objects pointed by the value of non-collection reference attributes. For example, when accessing o_1 in Figure 1, it prefetches only o_7 and o_{13}, but does not prefetch objects $o_2 \sim o_6$, $o_8 \sim o_{12}$, $o_{14} \sim o_{18}$, which are connected from the objects o_1, o_7, and o_{13} by non-collection reference attributes.

3 Type-level Access Patterns

In this section, we define the notions of the type-level access locality and the type-level access pattern. We also identify type-level access patterns that occur frequently in object-oriented navigational application. Figure 3 depicts a university database schema, which we will use as explanatory examples. We first define some terminology in Section 3.1.

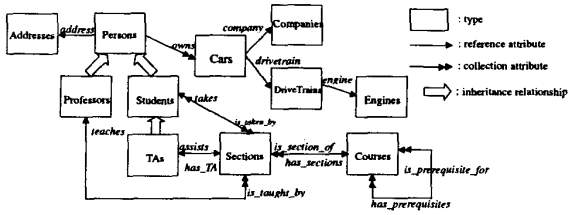


Figure 3. Example university database schema.

3.1 Terminology

A *navigational root set* is a set of root objects that the navigational application obtains to start navigation. We denote it by Ω. For example, the navigational root set in Figure 1 is $\{o_1, o_7, o_{13}\}$ (we call this Ω_1). ORDBMSs provide the query facility that can be used to obtain navigational root sets[11, 18]. An application can acquire multiple navigational root sets by issuing a query to the server for each navigational root set.

The *type-level path* of an object o is the sequence of attributes referenced from the root to the object. To express the type-level path of the root object, we assume that a navigational root set is the value of the virtual collection attribute of a virtual object. We denote the name of the virtual collection attribute by a_0. We then access the elements in a navigational root set as if we were accessing the elements of the collection attribute a_0. For example, in Figure 1, the type-level path of o_1 is a_0; that of o_2 a_0.address; that of o_3 a_0.owns; and that of o_4 a_0.owns.manufacturer.

The *type-level path reference string* for the navigational root set Ω is a sequence of type-level paths referenced in accessing the objects that are connected directly or indirectly from Ω. For example, in Figure 1, the type-level path reference string for Ω_1 is [a_0, a_0.address, a_0.owns, a_0.owns.manufacturer, a_0.owns.drivetrain, $\cdots$, a_0.owns.drivetrain, a_0.owns.drivetrain.engine].

The *type-level path graph* for the navigational root set Ω is a directed graph, where the nodes are the types of the objects accessed, and the directed edges the attributes referenced. All the edges are numbered to represent the order of insertion to the graph. This graph is used for capturing type-level access patterns. The types of the objects accessed are connected directly or indirectly from the root of the graph corresponding to Ω. The type-level path graph allows cycles of attributes to represent recursive patterns, which we discuss in Section 3.3.2. If there is a node on multiple paths(except for cycles) from the root node, the type-level path for the node becomes ambiguous. Thus, we disambiguate the paths by using a separate instance of the node for each distinct path. To capture the type-level access pattern, we keep additional information within the type-level path graph : *iterative attribute marking* and *recursive attribute marking*. If a certain attribute involves in

iteration, we mark the attribute as an iterative attribute. If it involves in recursion, we mark it as a recursive attribute. We dynamically build the type-level path graph by deriving the type-level paths of the objects as they are acceesed. We will show examples in Section 3.3. Table 1 summarizes the notation to be used throughout the paper.

Table 1. Summary of notation.

Symbols	Definitions
o	an object
o_i	the i-th object accessed
TLP(o)	the type-level path of object o
Ω	a navigational root set
CTLPG(Ω)	the current type-level path graph built for a navigation starting at Ω

3.2 Type-level Access Locality

ORDBMSs provide the reference and collection type attributes to model complex objects. Applications navigate complex objects using these attributes. If applications navigate complex objects of the same type, the same attributes are referenced repeatedly, even though the objects referenced are all different. We call this phenomenon the *type-level access locality*. We define it formally in Definition 1.

Definition 1 *The* **type-level access locality** *is a phenomenon that a large portion of substrings of the type-level path reference string is generated by repetitive use of a finite set of production rules[17].*

We define the type-level access pattern formally in Definition 2.

Definition 2 *A* **type-level access pattern** *is a finite set of production rules that generates type-level path reference strings that may appear as substrings of the type-level path reference string.*

It is not feasible to capture all possible type-level access patterns from the single type-level path reference string generated by a navigation at run-time. Thus, we predefine important classes of type-level access patterns and capture only these patterns at run-time.

3.3 Characteristics of Navigational Applications

We identify two important classes of type-level access patterns from the characteristics of navigational applications: iterative patterns and recursive patterns. Iterative patterns frequently occur due to the presence of collection attributes. Recursive patterns occur when navigating complex objects of recursive types. Recursive types are those that form a cycle in the schema graph. We frequently encounter recursive types in object-oriented applications[14].

To represent type-level access patterns, we use production rules. We also define an operator $\odot$ in Equation (1) to represent the concatenation operation between type-level paths. Here, $P_i (0 \leq i \leq k)$ is a subpath of a certain type-level path, and [ ] represents a type-level path reference string.

$$P_0 \odot [P_1, P_2, \cdots, P_k] = [P_0.P_1, P_0.P_2, \cdots, P_0.P_k] \quad (1)$$

3.3.1 Iterative Patterns

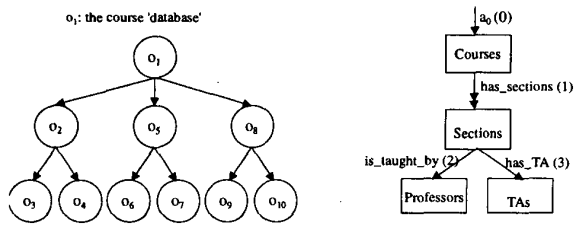
ORDBMSs provide a method for iteration(via Get_Next(), etc.) to access elements in a navigational root set or in the value of a collection attribute. Applications use iteration to access the objects pointed by the elements in a collection one by one. Therefore, the objects accessed through iteration tend to have the same type-level paths. The type-level access pattern that generates such type-level paths repeatedly is called the iterative pattern.

Definition 3 *An* **iterative pattern** *is a type-level access pattern that generates type-level path reference strings, in which identical type-level paths appear repeatedly.*

Figure 4 shows an example iterative pattern. In this figure the application finds the professor and TA for each section of the course 'database.' The object o_1(the course 'database') is the root object and has been obtained by the query, "SELECT * FROM Courses WHERE name='database'." The application iterates over the elements of the collection attribute 'has_sections' of the object o_1, accessing the section objects, o_2, o_5, and o_8. For each section object, in turn, it accesses objects pointed by the values of the attributes is_taught_by and has_TA, finding professors and TAs for each section. The type-level path reference string for this navigation is [a_0, a_0.has_sections, a_0.has_sections.is_taught_by, a_0.has_sections.has_TA, a_0.has_sections, a_0.has_sections.is_taught_by, a_0.has_sections.has_TA, a_0.has_sections, a_0.has_sections.is_taught_by, a_0.has_sections.has_TA]. Note that the type-level paths, a_0.has_sections, a_0.has_sections.is_taught_by, and a_0.has_sections.has_TA, appear repeatedly. The string can be represented as [a_0, [a_0.has_sections, a_0.has_sections.is_taught_by, and a_0.has_sections.has_TA]$^+$].[1] If there are multiple root objects, the string will become as [a_0, [a_0.has_sections, a_0.has_sections.is_taught_by, and a_0.has_sections.has_TA]$^+$]$^+$ forming a nested iterative pattern.

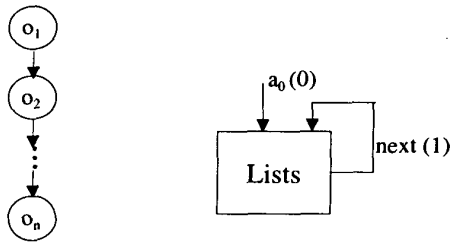
Iterative patterns can be nested in multiple-levels since iteration occurs whenever a collection attribute appears in the type-level graph. Suppose an application follows a path $a_0.a_1.a_2..a_n$ in the type-level path graph using iteration. Suppose that there are j(j $\leq$ n) collection attributes a_{c_1}, a_{c_2}, $\cdots$, a_{c_j} in the path. An iterative pattern for each collection

[1] [A]$^+$ means A can appear one or more times.



(a) the objects accessed. (b) the type-level path graph.

Figure 4. An example iterative pattern.



(a) the objects accessed. (b) the type-level path graph.

Figure 5. An example recursive pattern.

attribute is respectively $[a_0..a_{c_1}, \cdots]^+$, $[a_0..a_{c_2}, \cdots]^+$, $\cdots$, and $[a_0..a_{c_j}, \cdots]^+$. Therefore, the type-level access pattern for this navigation is $[\cdots [a_0..a_{c_1}, \cdots [a_0..a_{c_2}, \cdots [a_0..a_{c_j}, \cdots]^+ \cdots]^+ \cdots]^+ \cdots]$ forming a nested iterative pattern. Here, .. means omission of attributes in a type-level path, $\cdots$ means omission of type-level paths. Production rules for this iterative pattern are as in Equation (2) and are represented in a more compact form in Equation (3).

$$\left.\begin{array}{lcl} A_0 & \rightarrow & [a_0,\ a_0 \odot A_1]^{iter(a_0)} \\ A_1 & \rightarrow & [a_1,\ a_1 \odot A_2]^{iter(a_1)} \\ & \vdots & \\ A_k & \rightarrow & [a_k,\ a_k \odot A_{k+1}]^{iter(a_k)} \\ & \vdots & \\ A_n & \rightarrow & [a_n]^{iter(a_n)} \end{array}\right\} \quad (2)$$

$$\left.\begin{array}{lcll} A_k & \rightarrow & [a_k,\ a_k \odot A_{k+1}]^{iter(a_k)} & ,0 \leq k \leq n-1 \\ A_k & \rightarrow & [a_k]^{iter(a_k)} & ,k = n \end{array}\right\} \quad (3)$$

Here, the starting symbol is A_0. The superscript iter(a_k) is + if a_k is a collection attribute; otherwise, it is 1.

The iterative pattern for the application in Figure 4 can be represented as the production rules in Equation (4) with A_0 as the start symbol.

$$\left.\begin{array}{lcl} A_0 & \rightarrow & [a_0,\ a_0 \odot A_1]^+ \\ A_1 & \rightarrow & [has\_sections,\ has\_sections \odot [B_1, B_2]]^+ \\ B_1 & \rightarrow & [is\_taught\_by] \\ B_2 & \rightarrow & [has\_TA] \end{array}\right\} \quad (4)$$

The context-based prefetching method, in the absence of hints, can be regarded as the one having only one-level iterative patterns for prefetching. When an application accesses an object pointed by an element of the collection attribute a_i, this method only prefetches all the other objects pointed by the elements of a_i. That is, in the absence of hints, it does not prefetch other objects connected directly or indirectly from a_i. If some hints, such as MR[1][2], about the type-level paths that the application is to follow are given, the method can prefetch all the objects connected along these paths. The reference [1] points out that automatically issuing hints about such paths requires future work. Our method does automatically capture the access pattern generating the type-level paths that the application is to follow and prefetch objects using that pattern.

[2]However, these objects are limited to local contexts.

3.3.2 Recursive Patterns

When an application accesses objects of recursive types, cycles of attributes will appear repeatedly in the type-level paths. The path $a_i..a_j$ is cyclic if the type $T_{a_{i-1}}$ referenced by a_{i-1} is the same as the type T_{a_j} referenced by a_j including its subtypes or supertypes[8]. We define the recursive pattern in Definition 4.

Definition 4 *A* **recursive pattern** *is a type-level access pattern that generates type-level path reference strings having type-level paths, in which cycles of attributes appear repeatedly.*

Figure 5 shows an example recursive pattern. The application in this figure accesses a linked list. The application first accesses the root object, o_1, and then, recursively accesses the other objects directly or indirectly connected from o_1 using the attribute next. Therefore, the object reference string is $[o_1, o_2, o_3, \cdots, o_n]$, and the corresponding type-level path reference string is [a_0, a_0.next, a_0.next.next, $\cdots$, a_0.(next)$^{n-2}$.next]. In this type-level path reference string, the type-level paths, starting from the second one, contain cycles of the attribute next—forming a recursive pattern. The type-level path reference string for this navigation is represented as A_0, where the production rules A_0 and A_1 are $\{A_0 \rightarrow [a_0, a_0 \odot A_1], A_1 \rightarrow [\text{next}, \text{next} \odot A_1] | [\text{next}]\}$.

In this section, we have considered the recursion involving only one cycle. Theoretically, more complex types of recursion involving multiple cycles can be modelled. However, we only consider the case involving one cycle since most recursions in real-world applications are in this form. We leave the analysis of more complex types of recursive patterns as a future study.

4 Capturing Type-Level Access Patterns and Prefetching

This section presents the detailed techniques for capturing iterative and recursive patterns at run-time and prefetch-

ing based on the patterns captured. Section 4.1 describes the concept. Section 4.2 proposes the prefetching algorithm.

4.1 The Concept

We formally define the notions of capturing the type-level access pattern and prefetching based on the pattern captured.

Definition 5 Capturing *the type-level access pattern is the process of identifying the type-level path subgraph representing a useful pattern.*

A useful pattern is the one that makes the prefetching effective, i.e., that correctly anticipates the objects to be fetched in the future. In this paper we define the iterative and recursive patterns as useful patterns.

Definition 6 Prefetching *based on the type-level access pattern captured is the process of finding the object reference string that corresponds to the type-level path reference string generated by applying the production rules of the pattern in the order of the edges.*

Here, there is an interactive process between the type-level path reference string generated and the corresponding object reference string since the number of iterations or recursions in the type-level path reference string depends on the specific value of the object in the object reference string.

Now, we explain the detailed procedure of capturing iterative patterns and subsequent prefetching. Consider a navigational application that follows a type-level path subgraph starting with $a_0..a_i$, where the attribute a_i is of a collection type. If the type-level path $a_0..a_i$ appears in the type-level path reference string for the first time, we insert it into the CTLPG. When this type-level path appears again in the type-level path reference string, we recognize an iteration on the collection attribute a_i, mark the attribute a_i in the CTLPG as an iterative attribute, and capture an iterative pattern consisting of the subgraph having the attribute a_i as the root in the CTLPG. Next, the algorithm prefetches objects based on the pattern captured. In summary, at the first iteration of a collection attribute, the algorithm store in the CTLPG all the type-level paths traversed. At the second iteration, the algorithm recognizes the iteration and captures the iterative pattern, and then, prefetch objects based on the pattern captured.

Figure 6 illustrates the capturing and prefetching steps for an application having an iterative pattern. Figure 6 (a) $\sim$ (d) corresponds to the steps of accessing the objects $o_2 \sim o_5$ respectively. In each step, the left-hand side of the figure represents the objects accessed, and the right-hand side the captured type-level path graph, the CTLPG. The shaded circle represents the object being accessed in each step. In the CTLPG, all edges are numbered to represent the order of insertion to CTLPG. We suppose that a type-level path a_0 has been already inserted in the CTLPG by accessing the object o_1. In step (a), the type-level path of the object o_2, a_0.has_sections, is inserted in the CTLPG. Similarly, in steps (b) and (c), the type-level paths of the objects o_3 and o_4, a_0.has_sections.is_taught_by and a_0.has_sections.has_TA, are inserted in the CTLPG. In step (d), since the type-level path of the object o5, a_0.has_sections already exists in the CTLPG, the algorithm marks has_sections as an iterative attribute(denoted by a double arrow line) and captures the iterative pattern consisting of the subgraph having this attribute as the root. This pattern corresponds to [has_section, has_sections$\odot$[is_taught_by, has_TA]]$^+$. Then, it prefetches the objects, $o_5 \sim o_{10}$, based on the pattern captured. We expain how to determine the number of objects to prefetch in Section 4.2.

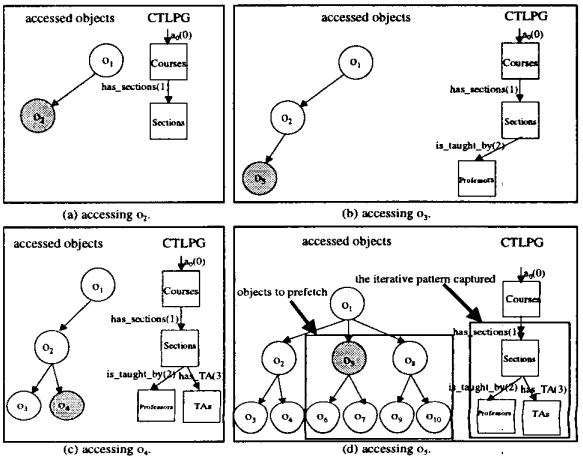


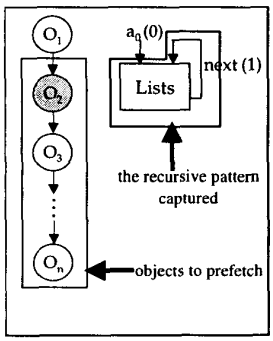
Figure 6. Capturing and prefetching steps for an application having an iterative pattern.

Capturing recursive patterns and prefetching based on them are done similarly. The recursive pattern is captured by checking if a subpath of the type-level paths forms a cycle. Consider an application that follows a type-level path, $a_0..a_i..a_j$, where the subpath $a_i..a_j$ makes a cycle. When an object whose type-level path is $a_0..(a_i..a_j)$ is accessed, the algorithm recognizes that the subpath $a_i..a_j$ completes a cycle.[3] Then, the algorithm captures the recursive pattern using the cycle. Next, the algorithm prefetches objects based on the pattern captured.

Figure 7 shows capturing and prefetching steps for the application in Figure 5. When accessing the object o_2, the type-level path of the object o_2, a_0.next, is inserted in the CTLPG. The algorithm recognizes that a subpath that has 'next' as the ending attribute(in this case, [next]) completes a cycle since the type 'Lists' referenced by the attribute next

[3]For conservative prefetching, we can defer the decision on the recognition of a recursion until a cycle appears more than once .

is the same as that referenced by the attribute a_0. Thus, the algorithm captures the recursive pattern consisting of the edges(attributes) and nodes forming the cycle. This pattern is equivalent to the production rule $A \rightarrow [next, next \odot A]|[next]$. Next, the algorithm prefetches the objects, $o_2 \sim o_n$, based on the recursive pattern captured. Here, n is restricted by the number of objects in the chain stored in the database or the number of objects to prefetch.



(a) accessing object o_2.

Figure 7. Capturing and prefetching steps for an application having a recursive pattern.

4.2 The Capturing and Prefetching Algorithm

Before presenting the proposed prefetching algorithm, we first define a basic navigational function that accesses an object pointed by a reference attribute or by an element of a collection attribute. Conventional ORDBMSs provide similar navigational functions[12, 18] for use in navigational applications. Thus, we explain how to augment the basic navigational function with the concept of capturing and prefetching.

Figure 8 shows the algorithm of the basic navigational function, Basic-Navigate. The inputs to Basic-Navigate are the object(o_{curr}) to access and the name of the attribute(a) to access. The output is the object o_{next} pointed by o_{curr}.a or an element of o_{curr}.a if a is of a collection type. The method of obtaining the oid of o_{next}, next_oid, differs according to the type of the attribute a(lines 1 $\sim$ 4). If the attribute a is of a collection type, we obtain next_oid using an associated cursor, which iterates over the elements in the collection. The function in line 2, nextElement(), iterates its cursor and returns an element referenced by the cursor. We assume that a cursor is opened when the function nextElement() is called for the first time. If the attribute a is of a non-collection reference type, next_oid is obtained directly as the value of o_{curr}.a(line 4). Next, the algorithm checks if the object whose oid is next_oid is in the cache. If it is not in the cache, the algorithm fetches it from the server(line 5 $\sim$ 6). Finally, the algorithm returns o_{next} as the output(line 7).

```
Basic-Navigate
Input:
  o_curr : object accessed
  a      : name of attribute to access
Output:
  o_next : object to return

begin
1:   if a is of collection type then
2:       next_oid = nextElement(o_curr.a's cursor);
3:   else
4:       next_oid = o_curr.a;
5:   if o_next, whose oid is next_oid, is not in cache then
6:     fetch o_next from server;
7:   return o_next;
end
```

Figure 8. The basic navigational function without prefetching.

Figure 9 shows the algorithm of Prefetch-Navigate, which augments Basic-Navigate with capturing and prefetching based on the type-level access pattern. The method of obtaining next_oid in Prefetch-Navigate is the same as in Basic-Navigate(lines 1 $\sim$ 4). Next, Prefetch-Navigate inserts type-level paths in the CTLPG, captures iterative and recursive patterns, and prefetches objects based on the captured pattern, instead of simply returning o_{next} from the cache or the server as in Basic-Navigate . We explain the method of obtaining the type-level path TLP(o_{curr}) of object o_{curr} in Section 4.3.2.

In lines 5 $\sim$ 12, the algorithm inserts type-level paths in the CTLPG, captures iterative and recursive patterns if exists. First, if the type level path of the object o_{next}, TLP(o_{next})(=TLP(o_{curr}.a)), is not in the CTLPG (line 5), it inserts TLP(o_{next}) into the CTLPG(line 7). It then checks if the inserted path completes a cycle. If there is a cyclic subpath that has 'a' as the ending attribute, it marks the attribute a in the CTLPG as a recursive attribute and captures a recursive pattern(line 8 $\sim$ 9). If the path TLP(o_{next}) already exists in the CTLPG (line 11), the algorithm marks the attribute a in the CTLPG as an iterative attribute if 'a' is of a collection type and captures the type-level access pattern consisting of the subgraph having 'a' as the root.

In lines 13 $\sim$ 20, the algorithm prefetches objects based on the patterns captured and returns o_{next}. If o_{next} is not in the cache (line 13), it should fetch it from the server. If there is a pattern captured(line 15), it will prefetch objects including o_{next} from the server based on the pattern captured(line 16). If no pattern has been captured, it simply fetches o_{next} from the server. Finally, in line 20, it returns the object o_{next}.

A CTLPG is allocated to each navigational root set. Whenever a query producing a navigational root set is issued, a CTLPG is created. When the navigation from the

```
Prefetch-Navigate
Input:
  o_curr : object to access
  a      : name of attribute to access
Output:
  o_next : object to return

begin
/* obtaining the next oid */
 1:   if a is of collection type then
 2:     next_oid = nextElement(o_curr.a's cursor);
 3:   else
 4:     next_oid = o_curr.a;
/* capturing the pattern */
 5:   if TLP(o_curr).a is not in CTLPG then
 6:     begin
 7:     insert TLP(o_curr).a into CTLPG;
 8:     if a subpath p in TLP(o_curr).a having 'a' as the ending attribute
        makes a cycle then
 9:       mark 'a' as a recursive attribute and capture the recursive pattern
          consisting of the cycle;
10:    end
11:  else  /* TLP(ocurr).a is in CTLPG */
12:    mark the attribute a in CTLP as an iterative attribute if a is of
         a collection type and capture the type-level access pattern consisting of
           the subgraph having 'a' as the root;
/* fetching or prefetching the objects */
13:  if o_next, whose oid is next_oid, is not in cache then
14:    begin
15:    if there is a captured pattern used for prefetching then
16:      prefetch objects connected from o_curr according to the captured pattern;
17:    else
18:      fetch o_next from server;
19:    end
20:  return o_next;
end
```

Figure 9. The navigational function augmented with capturing and prefetching.

navigational root set is completed, the CTLPG is destroyed. The size of a CTLPG is proportional to the number of types and attributes accessed in the navigation. Since it is generally small, the CTLPG can be resident in main memory.

5 Performance Evaluation

In this section we evaluate the performance of the proposed prefetching method(TypePrefetch) and compare it with those of on-demand fetching(OnDemandFetch) and context-based prefetching(ContextPrefetch). Section 5.1 explains the experimental environment and data sets. Section 5.2 presents the experimental results.

5.1 Experimental Data and Environment

We have performed three different experiments: 1) a navigational application described in Figure 4(*Iterative-Application*) having iterative patterns and one in Figure 5(*Recursive-Application*) having recursive patterns; 2) the OO7 benchmark[4, 16], which is a standard workload in object-oriented navigational applications. Here, we have used three different database sizes: small, medium, and large; 3) a real application for geographical information system(GIS) as the representative real-world application.

We have implemented the three fetching methods on the ODYSSEUS ORDBMS prototype being developed at KAIST. We have used a Sun Ultra-2 workstation for the client and a Sun Ultra-60 workstation for the server. The

Table 2. Experimental Results for Iterative-Application and Recursive-Application.

		Iterative-Application	Recursive-Application
OnDemandFetch	the relative # of round-trips	67.4	56.2
TypePrefetch	the relative wall clock time	2.49	6.12
ContextPrefetch	the relative # of round-trips	50.8	56.2
TypePrefetch	the relative wall clock time	1.95	6.12

object cache size for the client is 8 M bytes, and the page buffer size 16 M bytes. We have used a disk manager that directly handles raw disks to avoid any operating system's buffering effect.

As the performance measures, we have used the relative number of round-trips and the relative wall clock time.[4] To avoid the effects of noise and to increase the accuracy, we have run each experiment five times and averaged the results.

5.2 Experimental Results

Iterative-Application and Recursive-Application

The data set used for Iterative-Application is as follows. The total number of professors is 1,000, and the number of professors whose salary is more than $100,000 is 200. Each professor owns one car. The total number of car manufacturers is 5. The data set used for Recursive-Appilication is a linked-list similar to the one in Figure 5. Here, 500 objects are connected from the root object in sequence.

Table 2 shows the experimental results for Iterative-Application and Recursive-Application. TypePrefetch significantly outperforms not only OnDemandfetch but also ContextPrefetch in both Iterative-Application and Recursive-Application. Compared with OnDemandFetch, TypePrefetch reduces the number of round-trips by up to 64.7 times and improves the performance by up to 2.49 times. Compared with ContextPrefetch, TypePrefetch reduces the number of round-trips by up to 50.8 times and improves the performance by up to 1.95 times. The relative wall clock time is smaller than the relative number of round-trips because the former includes the disk access time to retrieve the objects from the database(i.e., the query processing time) in addition to the time for round-trip calls, but the disk access time can not be reduced by prefetching.

Table 2 shows similar results for Recursive-Application. The relative wall clock time of TypePrefetch improves in Recursive-Application because objects accessed in this experiment are clustered according to the order they are retrieved, thus, making the disk access cost smaller than in Iterative-Application. This indicates that prefetching can be

[4]We have measured the number of round-trips as the number of RPC's(remote procedure calls) between the client and the server and have counted all the RPC's, including transaction start and transaction commit, made by the application.

much more effective if objects accessed are well clustered in the server.

The OO7 Benchmark

In this experiment, we have executed traversal operations defined in the OO7 benchmark. We briefly explain traversal operations in OO7; we refer more details to the reference [4]. The database consists of modules(root objects), an assembly hierarchy, and composite part graphs connected from the leaf(base assembly) of the assembly hierarchy. The composite part graphs consist of atomic parts and connections. In the traversals T1, T2a, T2b, T2c, T3a, T3b, and T3c, applications traverse the assembly hierarchy and traverse the composite part graph. In the traversal T6, applications traverse the assembly hierarchy and traverse only the root part of the composite part graph.

Figure 10 shows the relative number of round trips of TypePrefetch for the OO7 benchmark. For the traversal T1, TypePrefetch reduces the number of round-trips by up to 195 times compared with OnDemandFetch and by 97.8 times compared with ContextPrefetch. For the traversals T2a $\sim$ T3c, the results, which are omitted in Figure 10, are similar to those in T1. For the traversal T6, TypePrefetch reduces the number by up to 63.3 and 35.4, respectively. We have more improvement in T1 than T6 since T1 accesses both the assembly hierarchy and the composite part graph while T6 does only the assembly hierarchy, but there are significantly more repetitive type-level accesses in the composite part graph than in the assembly hierarchy.

Figure 11 shows the wall clock time for the OO7 benchmark. In the traversal T1, TypePrefetch reduces the wall-clock time by up to 11.1 times compared with OndemandFetch and by up to 6.39 times compared with ContextPrefetch[5]; In T2a $\sim$ T3c, the results, which are omitted in Figure 11, are similar to those of T1. In T6, TypePrefetch improves performance by up to 3.57 times compared with OnDemandFetch and by up to 3.10 times compared with ContextPrefetch.

A Real GIS Application

In this experiment, we have used the map browser of a GIS system running on top of the ORDBMS. Here, we have used a real-world data set—the map of KangNam District of the Seoul city consisting of about 80,000 geometric objects. The database size is about 10 M bytes. Roads are modeled as polylines, and buildings as polygons. Both polylines and polygons consist of several line segments. In the experiment the map browser is to read the geometric objects of the entire map to the client.

[5]In the reference [1], experiments were performed using a relational DBMS. In this paper, we have used an ORDBMS instead. The implicit join operation in the ORDBMS, which follows the references, is relatively cheaper than the join operation in the relational DBMS, which uses value matching. Thus, the performance results for T1 $\sim$ T6 for ContextPrefetch appears better in this paper than in the reference [1].

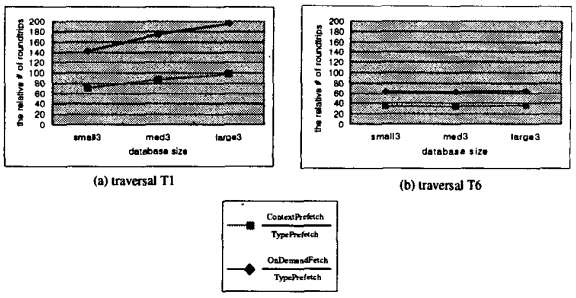


Figure 10. The relative number of round-trips for the OO7 benchmark.

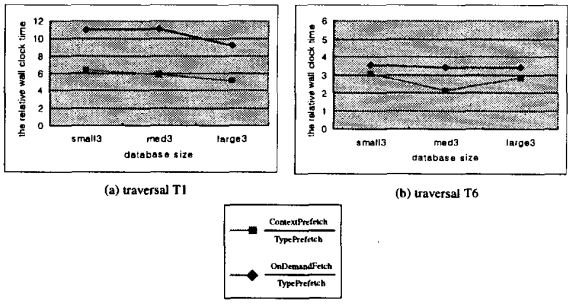


Figure 11. The relative wall clock time for the OO7 benchmark.

Table 3 shows the results for this experiment. Compared with OnDemandFetch, TypePrefetch reduces the number of round-trips by up to 402 times and improves the performance by up to 9.50 times. Compared with ContextPrefetch, TypePrefetch reduces the number of round-trips up to 51.3 times and improves the performance by up to 2.38 times. This indicates type-level access locality indeed occurs in real-world GIS applications and prefetching based on the type-level access pattern is effective in real-world applications as well.

Table 3. Experimental results for a GIS application using a real data set.

		A real GIS application
OnDemandFetch	the relative # of round-trips	402
TypePrefetch	the relative wall clock time	9.50
ContextPrefetch	the relative # of round-trips	51.3
TypePrefetch	the relative wall clock time	2.38

6 Conclusions

In this paper, we have proposed new notions of the type-level access pattern and the type-level access locality and presented a new prefetching method based on these notions. We also have proposed a formal framework for understanding underlying mechanisms of capturing and prefetching.

We have shown that the proposed method reduces round-trips and improves performance drastically compared with the existing fetching methods: on-demand fetching and context-based prefetching.

We have formally defined the type-level access pattern as a set of production rules, that generates the type-level path reference strings; capturing as the process of identifying the set of production rules representing a useful pattern; prefeching as the process of producing the object reference string that corresponds to the type-level path reference string generated by the type-level access pattern. We have identified two typical type-level access patterns in OR-DBMSs: the iterative pattern and the recursive pattern. Using these predefined access patterns, we have described the detailed mechanisms for capturing and prefetching which have incorporated in the algorithm proposed.

We have performed extensive experiments using various types of data sets including the OO7 benchmark and a real GIS application. Experimental results show that the proposed method significantly outperforms over both on-demand fetching and context-based prefetching. Compared with on-demand fetching, the proposed method reduces the number of round-trips by up to 402 times and improves the performance by up to 11.1 times. Compared with context-based prefetching, the proposed method reduces the number of round-trips by up to 97.8 times and improves the performance by up to 6.39 times. Especially, the proposed method provides large improvements in the OO7 benchmark and the real GIS application, where more complex object hierarchies are involved.

Overall, these results indicate that our approach provides a new paradigm for prefetching that improves performance in navigational applications significantly and is a practical method that can be implemented in commercial OR-DBMSs.

Acknowledgements The authors benefited from visiting the Computer Science Department of Stanford University in Summers 1999 and 2000 in completing the work presented in this paper. The authors are especially thankful for the incisive comments and suggestions by Gio Wiederhold and Hector Garcia-Molina. This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

References

[1] Bernstein, P. A., Pal, S., and Shutt, D., "Context-Based Prefetch for Implementing Objects on Relations," In *Proc. 21st Int'l Conf. on Very Large Data Bases*, Edinburgh, Scotland, pp. 327-338, Sept. 1999.

[2] Cattel, R. G. G. and Barry, D. K., *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997.

[3] Cooffman, E. G. Jr., and Denning, P.J., *Operating Systems Theory*, Prentice-Hall, 1973.

[4] Carey, M. J., DeWitt, D. J., and Naughton, J. F., "The OO7 benchmark," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, D.C., pp. 12-21, May 1993.

[5] Curewitz, K. M., Krishnan, P., and Vitter, J. S., "Pratical Prefetching via Data Compression," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, D.C., pp. 257-266, May 1993.

[6] Chang, E. E., Katz, R. H., "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," In *Proc Int'l Conf. on Management of Data*, ACM SIGMOD, Portland, Oregon, pp. 348-357, May 1989.

[7] Kim, W. et al., "Architecture of the ORION Next-Generation Database System," *IEEE Trans. on Knowledge and Database Engineering*, Vol. 2, No. 1, Mar. 1990

[8] Kim, W., *Introduction to Object-Oriented Databases*, The MIT press, 1990.

[9] Liskov, B. et al., "Safe and Efficient Sharing of Persistent Objects in Thor," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Montreal, Canada, pp. 318-329, June 1996.

[10] Lamb, C. et al., "The ObjectStore System," *Comm. ACM*, Vol. 34, No. 10, pp. 50-63, 1991.

[11] Oracle Corp., Oracle Call Interface Programmer's Guide Release 8.0, 1997.

[12] Park, C. M., Carey, M. J., and Dessloch, S., "MAJOR: A Java Language Binding for Object-Relational Databases," In *Proc. 8th Int'l Conf. Workshop on Persistent Object Systems*, Tiburon, California, Aug. 1998.

[13] Palmer, Z. and Zdonik, S. B., "Fido: A Cache That Learns to Fetch," In *Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, Catalonia, Spain, pp. 255-264, Sept. 1991.

[14] Rosenthal, A. et al., "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, D.C., pp. 166-176, May 1986.

[15] Stonebraker, M. and Brown, P., *Object-Relational DBMSs*, Morgan Kaufmann, 1999.

[16] Subramanian, M. and Krishnamurthy, V., "Performance Challeges in Object-Relational DBMSs," *IEEE Data Engineering Bulletin*, Vol. 22, No. 2, pp. 27-31, 1999.

[17] Taylor, R. G., *Models of Computation and Formal Languages*, Oxford University Press, 1998.

[18] UniSQL, Inc., UniSQL/X Application Program Interface Reference Guide, 1995.