A Formal Framework for Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs

Wook-Shin Han, *Member*, *IEEE*, Kyu-Young Whang, *Senior Member*, *IEEE*, and Yang-Sae Moon, *Member*, *IEEE*

Abstract—Prefetching is an effective method for minimizing the number of fetches between the client and the server in a database management system. In this paper, we formally define the notion of prefetching. We also formally propose new notions of the type-level access locality and type-level access pattern. 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. We then develop an efficient capturing and prefetching policy based on this formal framework. 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 fetches, thereby significantly enhancing the performance. We also address issues in implementing the proposed algorithm. We have conducted extensive experiments in a prototype ORDBMS to show effectiveness of our algorithm. Experimental results using the OO7 benchmark, a real GIS application, and an XML application show that our technique reduces the number of fetches by orders of magnitude and improves the elapsed time by several factors 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.

Index Terms—Prefetching, type-level access patterns, type-level access locality, object-relational DBMSs.

1 Introduction

Object-relational database management system (ORDBMS) [20] 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 tend to make the number of fetches increase in client/server DBMSs and causes serious performance degradation. By placing a cache at the client-side, ORDBMSs minimize fetches between the client and the server.

Existing object fetching policies are categorized into two: on-demand fetching [4] and prefetching [2], [7], [11], [13], [18], [21]. 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 fetches since it causes one fetch 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 fetches 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 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 [3], [16]. For example, in Fig. 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 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.

- *W.-S. Han is with the Department of Computer Engineering, Kyungpook National University, 1370 Sankyuk-dong, Book-gu, Daegu 702-701, Korea. E-mail: wshan@knu.ac.kr.*
- *K.-Y. Whang is with the Department of Computer Science and Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology (KAIST), 373-1 Koo-Sung Dong, Yoo-Sung Ku, Daejeon 305-701, Korea. E-mail: kywhang@cs.kaist.ac.kr.*
- *Y.-S. Moon is with the Department of Computer Science, Kangwon National University, 192-1, Hyoja2-Dong, Chunchon, Kangwon-Do 200-701, Korea. E-mail: ysmoon@kangwon.ac.kr.*

Manuscript received 29 Oct. 2003; revised 29 Nov. 2004; accepted 17 Feb. 2005; published online 18 Aug. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0218-1003.

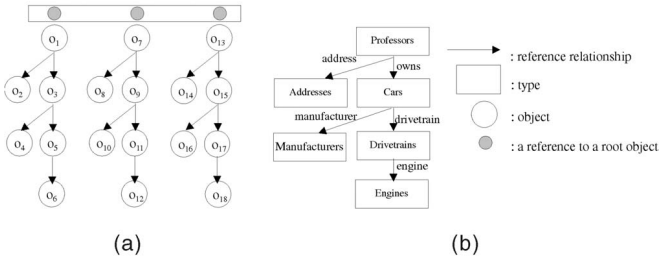

Fig. 1. A navigational application to get professors addresses and cars (and their manufacturers, drivetrains, and engines), where the professor's salary is more than $\$100,000$. (a) The objects accessed and (b) the types and the attributes of the objects accessed.

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. Fig. 1a shows the sequence 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 definitions in Section 3.2. Many navigational applications in ORDBMSs have type-level access locality. Therefore, by capturing type-level access patterns in these applications, we can predict future accesses to objects and effectively prefetch the objects based on these patterns.

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 model of capturing and prefetching to help understand the detailed mechanisms involved. Third, we propose a new prefetching algorithm that implements this model. 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.

In page-based prefetching, we fetch all the objects together in the page containing the object requested [11], [14]. 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 of objects in a page, the method fails to work well when applications do not access objects in the clustered order.

In object-level/page-level access pattern based prefetching, we predict future object/page access patterns from recent object/page access references [7], [18]. Palmer and Zdonik [18] 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. [7] 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.

In user-hint-based prefetching, we prefetch objects based on hints provided by the user. Chang and Katz [8] proposed a method of prefetching objects based on the user hints, such as "my prime access is via configuration relationships." Commercial ORDBMSs provide methods based on user-hints similar to this example [16]. 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. Our algorithm works in a way similar to the hint-based approach, which would produce the best performance if the hints were properly provided. That is, our approach can be regarded as automatically generating proper hints utilizing the type-level access patterns dynamically captured at runtime.

In context-based prefetching [2], which is the most recent work on prefetching, we fetch 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 query results and collections. Fig. 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

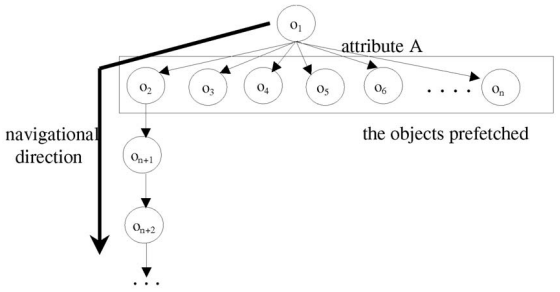

Fig. 2. An example of context-based prefetching.

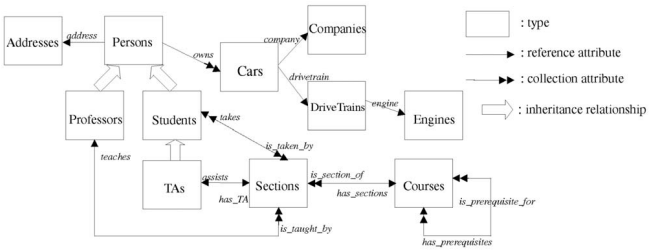

Fig. 3. An example university database schema.

hierarchy in the breadth-first-search (BFS) fashion. This method has been implemented in a commercial DBMS with performance increases of up to 70 percent 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 Fig. 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 noncollection reference attributes. For example, when accessing o_1 in Fig. 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 noncollection 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. Fig. 3 depicts a university database schema, which we will use as an explanatory example. We first define some terminology in Section 3.1.

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 Fig. 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 [16], [23]. An application can acquire multiple navigational root sets by issuing a query to the server for each navigational root set.

A *navigational complex object* is a set of objects that are accessed during a navigation and that are recursively connected from a navigational root set. Therefore, in prefetching, we are interested in finding a navigational complex object instead of an entire complex object stored in the database.

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_Ω. We then access the elements in a navigational root set as if we were accessing the elements of the collection attribute a_Ω. For example, in Fig. 1, the type-level path of o_1 is a_Ω; that of o_2 $a_\Omega.address$; that of o_3 $a_\Omega.\text{owns}$; and that of o_4 $a_\Omega.owns.manufacturer$.

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 Fig. 1, the type-level path reference string for Ω_1 is

$$[a_{\Omega_1}, a_{\Omega_1}.address, a_{\Omega_1}.owns, a_{\Omega_1}.owns.manufacturer, a_{\Omega_1}.owns.drivetrain, \cdots, a_{\Omega_1}.owns.drivetrain, a_{\Omega_1}.owns.drivetrain.engine].$$

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. The type-level path graph can be extended to store the information about attributes actually accessed instead of storing all the attributes [10]. 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 3.3.3. 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 evolves in iteration, we mark the attribute as an iterative attribute. If it evolves 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 accessed. We will show examples in Section 3.3. Table 1 summarizes the notation to be used throughout the paper.

3.2 Type-Level Access Locality

ORDBMSs provide the reference and collection type attributes to model complex objects. Applications navigate navigational complex objects using these attributes. If applications navigate navigational complex objects of the same type, the same attributes are referenced repeatedly, even though the objects referenced are all different. We call

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 so far for a navigation starting from Ω

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 a small number of type-level access patterns.*

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 [22] that generates type-level path reference substrings appearing in a given type-level path reference string.*

Here, a production rule is in the form of $A \rightarrow x$, where A is a nonterminal and x is a string consisting of terminals and/or nonterminals.

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

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 [19].

3.3.1 Representation of Type-level Path Reference Strings

Before predefining important classes of type-level access patterns, we first introduce the formal representation of type-level path reference strings. We use the string concatenator $\bullet$ in (1) and a newly defined operator $\odot$ called the *path constructor* in (2) as follows:

The properties of the operator $\bullet$:

$$[\,P_0,\,\cdots,P_m] \bullet [\,P'_0,\,\cdots,P'_n] = [\,P_0,\,\cdots,P_m,\,P'_0,\,\cdots,P'_n]. \quad (1)$$

The properties of the operator $\odot$:

$$[P_0] \odot [\,P_1,\,P_2,\,\cdots,P_k] = [P_0, P_0.P_1,\,P_0.P_2,\,\cdots,P_0.P_k]. \quad (2)$$

Here, m, $n \geq 0$, P_i, P'_i is a subpath of a type-level path, and [] is the string constructor.

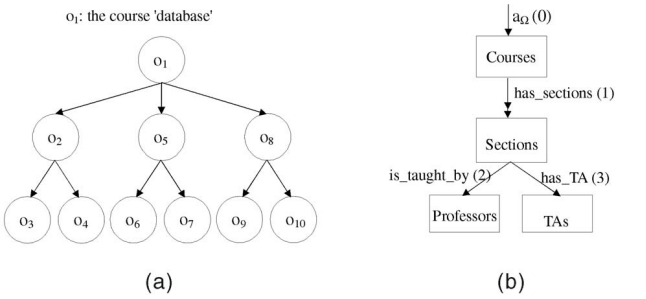

Fig. 4. An example iterative pattern. (a) The objects assessed. (b) The type level path graph.

$S \bullet S'$ represents the concatenation of the two input type-level path reference strings, S and S'. $[P] \odot S$ represents a type-level path reference string generated from the type-level path reference string S by prefixing the type-level subpath P to each element of S and by including P itself. Typically, S is generated from a type-level path subgraph. Thus, the operator $\odot$ represents concatenation of a type-level subpath with a type-level path subgraph.

3.3.2 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. Thus, 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.*

Example 1. Fig. 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

$$[a_\Omega, a_\Omega.has\_sections, a_\Omega.has\_sections.is\_taught\_by,$$
$$a_\Omega.has\_sections.has\_TA, a_\Omega.has\_sections,$$
$$a_\Omega.has\_sections.is\_taught\_by, a_\Omega.has\_sections.has\_TA,$$
$$a_\Omega.has\_sections, a_\Omega.has\_sections.is\_taught\_by,$$
$$a_\Omega.has\_sections.has\_TA].$$

Note that the type-level paths, $a_\Omega.has\_sections$,

$$a_\Omega.has\_sections.is\_taught\_by,$$

and $a_\Omega.has\_sections.has\_TA$, appear repeatedly. The string can be represented as[1]

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

$$[a_\Omega] \bullet ([a_\Omega.has\_sections, a_\Omega.has\_sections.is\_taught\_by, a_\Omega.has\_sections.has\_TA])^+.$$

If there are multiple root objects, the string will become as

$$([a_\Omega] \bullet ([a_\Omega.has\_sections, a_\Omega.has\_sections.is\_taught\_by, a_\Omega.has\_sections.has\_TA])^+)^+$$

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 path graph. Suppose an application follows a path $a_\Omega.a_1.a_2..a_n$ 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 attribute is $[a_\Omega..a_{c_1}, \cdots]^+$, $[a_\Omega..a_{c_2}, \cdots]^+, \cdots$, and $[a_\Omega..a_{c_j}, \cdots]^+$, respectively. Therefore, the type-level access pattern for this navigation is

$$[\cdots] \bullet ([a_\Omega..a_{c_1}, \cdots] \bullet ([a_\Omega..a_{c_2}, \cdots] \bullet ([a_\Omega..a_{c_j}, \cdots])^+)^+)^+$$

forming a nested iterative pattern. Here, .. means omission of attributes in a type-level path. Production rules for this iterative pattern are as in (3):

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

Here, the starting symbol is A_0, and a_0 is a_Ω. The superscript $iter(a_k)$ means zero or more iterations, and is dynamically determined according to the type and the cardinality of a_k: If a_k is a collection attribute, $iter(a_k)$ is the number of element references in the collection attribute; if a noncollection attribute, it is 0 or 1 (we denote this special case as $iter_1(a_k)$). If $iter(a_k)$ is greater than zero, we call the corresponding rule the *iter-rule*. If $iter(a_k)$ is equal to zero, we call the corresponding rule the *NULL-rule*.

Since each node(type) in a type-level path graph can have more than one outgoing edge(attribute), (3) should be extended to include this case. In this section, we limit our discussion to a type-level path tree, which is a type-level path graph having no cycles. We will discuss type-level path graphs having cycles in Section 3.3.3 regarding the recursive pattern. We construct production rules for the iterative pattern by mapping each edge in the type-level path tree to a production rule by using the following rule:

Rule 1. *Suppose that the nonleaf node A in the type-level path tree is connected through the incoming edge a, and nodes $B_1, B_2, \cdots, B_j (j \geq 0)$ through the outgoing edges $b_1, b_2, \cdots, b_j$ from A, respectively. Edges b_i are numbered in the order of the subscript. Then, the production rule A' corresponding to the edge a is represented as follows*[2]*:*

$$A' \rightarrow ([a] \odot ([\,] \bullet B'_1 \bullet B'_2 \bullet \cdots \bullet B'_j))^{iter(a)}, \quad (4)$$

where $B'_1, B'_2, \cdots, B'_j$ are the left sides of the production rules corresponding to the edges $b_1, b_2, \cdots, b_j$, respectively.

In (4), if $j = 0$, the resulting rule becomes (5):

$$A' \rightarrow [a]^{iter(a)}. \quad (5)$$

We note that (5) represents a special case of (4) when the node A is a leaf.

Using Rule 1, the iterative pattern for the application in Fig. 4 can be represented as the production rules in (6) with A_0 as the start symbol:

$$\left.\begin{array}{lcl} A_0 & \rightarrow & ([a_\Omega] \odot ([\,] \bullet A_1))^{iter(a_\Omega)} \\ A_1 & \rightarrow & ([has\_sections] \odot ([\,] \bullet B_1 \bullet B_2))^{iter(has\_sections)} \\ B_1 & \rightarrow & [is\_taught\_by]^{iter_1(is\_taught\_by)} \\ B_2 & \rightarrow & [has\_TA]^{iter_1(has\_TA)} \end{array}\right\}. \quad (6)$$

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 [2],[3] 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 [2] 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.

3.3.3 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 [12]. 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.*

Example 2. Fig. 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 $[a_\Omega, a_\Omega.next, a_\Omega.next.next, \cdots, a_\Omega.(next)^{n-2}.next]$. 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

2. Here, we consider the order only among the outgoing edges of each incoming edge for the sake of a clearer concept. That is, we only allow navigations doing depth first-search over the type-level path graph. We discuss the general case of navigation without this limitation in Appendix A which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm.

3. However, these objects are limited to local contexts.

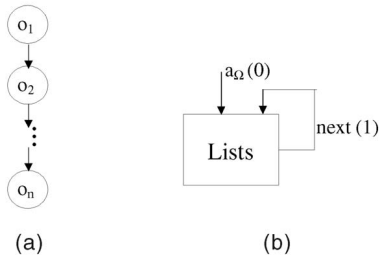

Fig. 5. An example recursive pattern. (a) The objects accessed. (b) The type-level path graph.

type-level path reference string for this navigation is represented as A_0, where the production rules are

$$\{A_0 \rightarrow ([a_\Omega] \odot ([\,] \bullet A_1))^{iter_1(a_\Omega)},$$
$$A_1 \rightarrow ([next] \odot ([\,] \bullet A_1))^{iter_1(next)}\}.$$

In general, the number of attributes that appear repeatedly in a type-level path generated by a recursive pattern is equal to or greater than one, being the same as the number of attributes comprising the cycle in the schema graph. For example, consider an application that follows a type-level path $..a_i..a_j$. We assume that the path $a_i..a_j$ makes a cycle. The type-level path reference string for this navigation is

$$[\cdots, ..a_i, ..a_i.a_{i+1}, \cdots, ..a_i..a_j, ..a_i..a_j.a_i, ..a_i..a_j.a_i.a_{i+1},$$
$$\cdots, ..(a_i..a_j)^2, ..(a_i..a_j)^2.a_i, \cdots].$$

We construct production rules for the recursive patterns by mapping each edge in the cycle in the type-level path graph to a production rule by using the following rule.

Rule 2. *Suppose that the path $a_i..a_j$ makes a cycle. Suppose that a node A_k in the type-level path graph is connected through $a_k (i \le k \le j)$. Then, the production rule A'_k for the edge a_k is defined as follows:*

$$\left.\begin{array}{lcll} A'_k & \rightarrow & ([a_k] \odot ([\,] \bullet A'_{k+1}))^{iter(a_k)} & , i \le k < j \\ A'_k & \rightarrow & ([a_k] \odot ([\,] \bullet A'_i))^{iter(a_k)} & , k = j \end{array}\right\}. \quad (7)$$

Rule 2 also represents cases where some of the attributes making a cycle in the schema graph are of collection types. Thus, each nonterminal in (7) also has two alternative rules, depending on the value of $iter(a_k)$: one iter-rule and one NULL-rule. Using such attributes, we can also model mixtures of iterative and recursive patterns. For example, in Fig. 6, the application retrieves all the prerequisites of each course. Since the attribute $has\_prerequisites$ is of a collection type and is involved in a cycle, the production rules for this case are as in (8):

$$\left.\begin{array}{lcl} A_0 & \rightarrow & ([a_\Omega] \odot ([\,] \bullet A_1))^{iter(a_\Omega)} \\ A_1 & \rightarrow & ([has\_prerequisites] \odot ([\,] \bullet A_1))^{iter(has\_prerequisites)} \end{array}\right\}. \quad (8)$$

We note that the grammar generated by Rules 1 and 2 is a regular tree grammar [5], [15]. Thus, a type-level path reference string can be represented as an equivalent ordered labeled tree generated by a regular tree grammar. For example, the type-level reference string in Example 1 can be replaced by an equivalent ordered labeled tree as follows:

$$([a_\Omega] \odot ([address] \odot ([\,]) \bullet ([owns] \odot (manufacturer \odot ([\,])))))$$
$$= a_\Omega(address(), owns(manufacturer())).$$

Here, $l(T_1, T_2, \cdots, T_n)$ represents a tree consisting of a labeled node l and n subtrees [5].[4]

In this section, we have considered the recursion involving only one cycle. Theoretically, more complex types of recursion involving multiple cycles can be modeled. However, we only consider the case involving one cycle since most recursions in real-world applications are in this form [1].

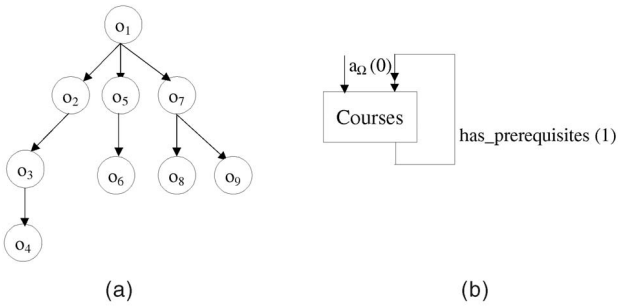

Fig. 6. An example where both an iterative and recursive patterns occur simultaneously. (a) The objects accessed. (b) The type-level path graph.

4 Capturing Type-Level Access Patterns and Prefetching

This section presents the detailed techniques for capturing iterative and recursive patterns at runtime and prefetching based on the patterns captured. Section 4.1 describes the concept. Section 4.2 proposes the prefetching algorithm. Section 4.3 explains implementation issues.

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 (which corresponds to a set of production rules) representing a useful pattern.*

A useful pattern is the one that makes the prefetching effective, i.e., the one that correctly anticipates the objects to be fetched in the future. In this paper, we define the iterative and recursive patterns as useful patterns. According to Rules 1-2, there exists a mapping between

4. The reference [5] deals with ranked trees that have predefined bound on the number of children of each node, whereas the reference [15] deals with unranked trees (i.e., no predefined bound on the number of children of each node) using an encoding scheme. In our paper, since the cardinality of a collection attribute is not predefined, we can use the encoding scheme in the reference [15] to handle unranked trees. We omit the detailed discussion of the regular tree grammar since it is not a main focus of our paper.

the type-level path subgraph and the type-level access pattern (i.e., a set of production rules).

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 captured according to 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.

Formally, let G be the grammar for the pattern captured and A_0 be the starting nonterminal symbol of G. The derivation of the type-level path reference string S by using the grammar G can be represented as $A_0 \overset{*}{\underset{G}{\Rightarrow}} S$. At each step of the derivation, a nonterminal A is substituted with the rightside of the production rule for A. However, since each nonterminal has two alternative rules, a iter-rule and a NULL-rule, we have to decide which rule to apply to generate the type-level path reference string S. In case of the iter-rule, we also have to decide the number of iterations for the rule. To do so, we need to check the specific value of the object (i.e., a valid pointer or null pointer to the child object) in the object reference string that corresponds to the type-level path reference string. That is, by accessing the value of the current object, we can produce a symbol (i.e., a type-level path) in S for the next object to access.

Now, we explain the detailed procedure of capturing iterative patterns and subsequent prefetching using the Current Type-Level Path Graph(CTLPG). CTLPG is a dynamic data structure for representing the production rules that have been identified by the current time during the capturing process. Consider a navigational application that follows a type-level path subgraph starting with $a_\Omega..a_i$, where the attribute a_i is of a collection type. If the type-level path $a_\Omega..a_i$ 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 stores 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.

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, $a_\Omega..a_i..a_j$, where the subpath $a_i..a_j$ makes a cycle. When an object whose type-level path is $a_\Omega..(a_i..a_j)$ is accessed, the algorithm recognizes that the subpath $a_i..a_j$ completes a cycle.[5] Then, the algorithm marks every attribute in the cycle as a recursive attribute and captures the recursive pattern. Next, the algorithm prefetches objects based on the pattern captured.

```
Prefetch-Navigate
Input:
  o_curr : object to access
  a      : name of attribute to access
Output:
  o_next : object to return
/* obtaining the next oid */
 1:  if (a is of collection type)
 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)
 6:      insert TLP(o_curr).a into CTLPG;
 7:      if (a subpath p in TLP(o_curr).a having a as the ending attribute makes a cycle)
 8:          capture the recursive pattern consisting of the cycle
             by marking every attribute in the cycle as a recursive attribute;
 9:  else  /* TLP(o_curr).a is in CTLPG */
10:      capture the iterative pattern consisting of the subgraph having a as the root
          by marking the attribute a in CTLPG as an iterative attribute if a is of a collection type;
/* fetching or prefetching the objects */
11:  if (o_next, whose oid is next_oid, is not in cache)
12:      if (a is marked as an iterative attribute or a recursive attribute)
13:          let currEdge be the edge corresponding to the attribute a in p;
14:          o_set = PrefetchObjects(o_curr, currEdge);
15:          insert objects in o_set into the object cache;
16:      else
17:          fetch o_next from server;
18:  return o_next;
```

Fig. 7. The navigational function augmented with capturing and prefetching.

Appendix B which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm contains illustrative examples of capturing and prefetching steps for iterative and recursive applications.

4.2 The Capturing and Prefetching Algorithms

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 [17], [23] for use in navigational applications. Thus, we explain how to augment the basic navigational function with the concept of capturing and prefetching. Fig. 19 of Appendix C (which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm) shows the algorithm Basic-Navigate for the basic navigational function without prefetching. 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 by an element of $o_{curr}.a$ if a is of a collection type. (Due to space limit, we present it in Appendix C (which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm)).

Fig. 7 shows the algorithm of Prefetch-Navigate, which augments Basic-Navigate with capturing and prefetching based on the type-level access pattern. First, Prefetch-Navigate obtains next_oid (lines 1-4). Next, Prefetch-Navigate inserts type-level paths in the CTLPG, captures

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

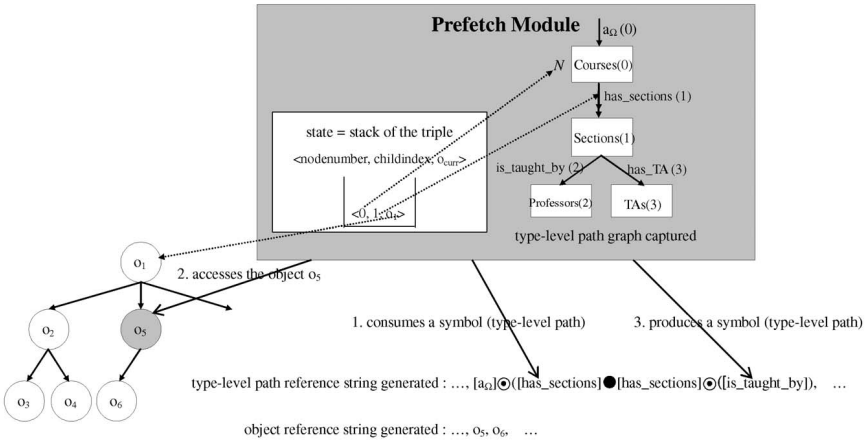

Fig. 8. Architecture of the prefetch module.

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 -10, the algorithm inserts type-level paths in the CTLPG, captures an iterative or recursive pattern if any exists. First, if the type level path of the object o_{next}, $\mathrm{TLP}(o_{next})(= \mathrm{TLP}(o_{curr}.a))$, is not in the CTLPG (line 5), it inserts $\mathrm{TLP}(o_{next})$ into the CTLPG (line 6). 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 (lines 7-8). If the path $\mathrm{TLP}(o_{next})$ already exists in the CTLPG (line 9), 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 11-18, objects are prefetched based on the patterns captured, and o_{next} is returned. If o_{next} is not in the cache (line 11), it should be fetched from the server. If a is marked as an iterative attribute or a recursive attribute (i.e., if a is part of a pattern captured) (line 12), objects including o_{next} will be prefetched from the server based on the pattern captured by calling the function PrefetchObjects (lines 13-15). If no pattern has been captured, o_{next} is simply fetched from the server. Finally, in line 18, the object o_{next} is returned.

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 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.

Before presenting the detailed algorithm for PrefetchObjects, we describe the connection between the formal framework and the prefetch algorithm. Fig. 8 shows the overall architecture of our prefetch module. The module contains a type-level path graph (= a type-level access pattern) and a state. Here, a state is a stack of the triple $< nodenumber, childindex, o_{curr} >$, where $nodenumber$ points to a node N in the type-level path graph, and $childindex$ means the edge connected to a child node from the node N. As described in Definition 6, the prefetch module

1. consumes a type-level path generated by applying the production rules of the pattern captured,
2. prefetches an object using the type-level path consumed and the state information, and
3. produces a type-level path for the object to access next.

Here, the resulting type-level path reference string is a string in the language generated by the grammar, in particular, by the production rules of the pattern. Here, the generation of the string is controlled by the interactive process between the type-level path reference string and the corresponding object reference string by checking the specific value of the object (i.e., a valid pointer or null pointer to the child object) in the object reference string. If the pointer is valid, then apply the iter-rule; if null, apply the NULL-rule.

Fig. 9 shows the detailed algorithm for PrefetchObjects. The inputs to PrefetchObjects are the object accessed (o_{curr}) and the current edge of CTLPG ($currEdge$). The output is a set of objects prefetched (o_{set}). It consumes the current symbol and makes a state transition by calling MakeState-Transition, where MakeStateTransition prefetches objects by using o_{curr} and the symbol consumed (type-level path) and produces a next input symbol.

We now describe the algorithm MakeStateTransition in detail. In lines 3-4, if the value of $o_{curr}.a$ is NULL or EndOfCursor ($o_{curr}.a$) is false, no objects (an empty set) are prefetched for the subgraph having $currEdge$ as the incoming edge to its root. Thus, we follow sibling edges or backtrack to the parent by calling FollowSiblingORBackTrack.

In lines 5-18, objects are prefetched based on the iterative pattern. If a is marked as an iterative attribute, the algorithm fetches the object pointed by each element in the collection attribute a one by one navigating the subgraph connected from the object. This case corresponds to applying an iter-rule with n iterations to generate the type-level path reference string. Here, the originating edge

```
PrefetchObjects
Input:
  o_curr : object accessed
  currEdge: current edge of CTLPG
Output:
  prefetchBuffer : prefetch buffer
1:   produce an input symbol using currEdge
2:   PushStack(s, <GetSourceNodeNumber(currEdge), GetChildIndex(currEdge)>, o_curr);
3:   while (not EmptyStack(s) and prefetchBuffer is not full)
4:       consume a current symbol and assign it to currEdge;
5:       MakeStateTransition(s, currEdge, prefetchBuffer);
6:   return prefetchBuffer;
MakeStateTransition
Input:
s: state
currEdge: input symbol
InOut:
prefetchBuffer: prefetch buffer
1:   a = GetAttributeName(currEdge);
2:   <nodeNo, childIndex , o_curr,> = GetStackTop(s);
3:   if ((o_curr.a is NULL) or (EndOfCursor(o_curr.a's cursor) is true) )
       /* apply the NULL-rule: iter(a) = 0 in Rules 1 and 2 */
4:       FollowSiblingORBacktrack(s);
5:   else if (a is marked only as an iterative attribute)
6:       if (EndOfCursor(o_curr.a's cursor) is false)
7:           next_oid = nextElement(o_curr.a's cursor);
8:           fetch the object o_next whose oid is next_oid and insert o_next into prefetchBuffer;
9:           if (currEdge has the outgoing edges b_1, b_2, …, b_j)   /* follow children */
                 /* corresponding to A' → ([a]⊙([ ]•B_1'•…•B_j')) iter(a) , when iter(a) > 0 in Rule 1 */
10:              PushStack(s, <GetNodeNo(nodeNo, childIndex), 1, o_next>)   /* follow the first child */
11:              produce an input symbol GetEdge(GetNodeNo(nodeNo, childIndex), 1);
12:          else FollowSiblingORBacktrack(s);
13:  else if (a is marked neither as an iterative attribute nor as a recursive attribute)
14:      fetch the object o_next whose oid is o_curr.a and insert o_next into prefetchBuffer;
15:      if (currEdge has the outgoing edges b_1, b_2, …, b_j) /* follow children */
             /* corresponding to A' → ([a]⊙([ ]•B_1'•…•B_j')) iter1(a), when iter_1(a) = 1 in Rule 1 */
16:          PushStack(s, <GetNodeNo(nodeNo, childIndex), 1, o_next>) /* follow the first child */
17:          produce an input symbol GetEdge(GetNodeNo(nodeNo, childIndex), 1);
18:      else FollowSiblingORBacktrack(s);
19:  else if (a is marked only as a recursive attribute)
20:      fetch the object o_next whose oid is o_curr.a and insert o_next into prefetchBuffer;
21:      let A_k+1 be the destination node for currEdge; /* we only deal with the case involving one cycle*/
       /* corresponding to A'_k→([a_k]⊙([ ]•A'_k+1)) iter1(ak) or A'_k→([a_k]⊙([ ]•A'_i)) iter1(ak), when iter_1(a_k) = 1 in Rule 2 */
22:      PushStack(s, <k+1, 1, o_next>)
23:      produce an input symbol GetEdge(GetNodeNo(nodeNo, childIndex), 1); /* follow the first child */
24:  else if (a is marked both as an iterative attribute and a recursive attribute)
25:      if (EndOfCursor(o_curr.a's cursor) is false)
26:          next_oid = nextElement(o_curr.a's cursor);
27:          fetch the object o_next whose oid is next_oid and insert o_next into prefetchBuffer;
28:          let A_k+1 be the destination node for currEdge; /* we only deal with the case involving one cycle*/
       /* corresponding to A'_k→ ([a_k]⊙([ ]•A'_k+1))iter(ak) or A'_k→([a_k]⊙([ ]•A'_i))iter(ak) , when iter(a_k) > 0 in Rule 2 */
29:          PushStack(s, <k+1, 1, o_next>)
30:          produce an input symbol GetEdge(GetNodeNo(nodeNo, childIndex), 1);
FollowSiblingORBacktrack
Input:
s: state
1:   while (not EmptyStack(s))
2:       <nodeNo, childIndex, o_curr> = PopStack(s); /* backtrack to the parent */
3:       a = GetAttributeName(GetEdge(nodeNo, childIndex));
4:       if (a is marked as iterative attribute and EndOfCursor(o_curr.a's cursor) is false)
5:           PushStack(s, <nodeNo, childIndex, o_curr>); /* follow the remaining element references in a */
6:           produce an input symbol GetEdge(nodeNo, childIndex);
7:           break;
8:       else if (GetEdge(nodeNo, childIndex + 1) exists) /* follow the remaining siblings */
9:           PushStack(s, <nodeNo, childIndex+1, o_curr>);
10:          produce an input symbol GetEdge(nodeNo, childIndex + 1);
11:          break;
```

Fig. 9. The prefetch algorithm.

of the iterative pattern is of a collection type, but the other edges in the pattern may be either of a collection type or of a noncollection type. In lines 5-12, the case having edges of collection types is handled; in lines 13-18, the case having edges of noncollection types is handled.

In lines 19-30, objects are prefetched based on the recursive attribute. If a is marked as a recursive attribute, the algorithm follows the cycle corresponding to the recursive pattern.[6] In lines 24-30, as stated in Section 3.3.3, we handle the case (the mixed pattern) where some of the edges making the cycle are of collection types.

An additional issue in prefetching is how many objects we should prefetch at one time. In this paper, to avoid ineffective prefetch, we use a heuristic. Here, we use the (prefetch hit) ratio of the number of objects that have actually been accessed to the number of objects prefetched in the past to determine the number of objects to prefetch in the future. If the ratio is $\leq$ X percent, the number of objects to prefetch is reduced by Y percent. If the ratio is $>$ X percent, the number of objects to prefetch is increased by Y percent with a constraint that the total space for the objects fetched should not exceed the maximum prefetch buffer size. In our experiments, even if we increase the prefetch buffer size above some threshold, it affects the number of fetches, but much less does the total elased time. This is because, once the number of fetches improves significantly, the query processing time would be a dominant factor in the total elapsed time. Further improvement in the fetches has a marginal effect on the total elapsed time. Thus, we have determined the prefetch buffer size to

6. For the recursive pattern, by checking if the objects themselves accessed form a cycle, we prevent the algorithm from getting into an infinite loop.

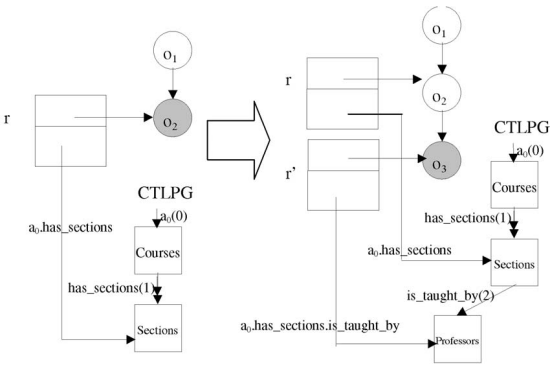

Fig. 10. Maintenance of type-level paths using smart pointers.

be 1M bytes. The rationale for setting X (= 50 percent) is as follows: Even if half of the objects prefetched are indeed accessed, we still have performance gain since the round-trip cost occupies more than 50 percent of the total performance in our experiments. The rational for setting Y (= 50 percent) is as follows: By adapting the prefetch buffer size exponentially based on the recent prefetch hit ratio (Y = 50 percent), we can easily accommodate even the worst cases.

4.3 Implementation Issues

4.3.1 Client/Server Prefetching Architecture

The prefetching method based on the type-level access pattern is implemented as a module for capturing the access pattern (*the capturing module*) and a module for prefetching based on the pattern captured (*the prefetching module*). Thus, where to put these modules in the client/server architecture is an important issue that can affect the performance.

The capturing module is placed at the client-side for the following reasons: Navigation is accomplished by calling the navigational function in the client. Thus, if an object to be fetched already exists in the client cache, any message from the navigation function will not be delivered to the server and, thus, the server is not able to know accurate access patterns of an application. On the other hand, the prefetching module is placed at the server-side because the oids of the objects to prefetch can only be obtained by navigating through the objects stored in the server according to the type-level access pattern. Since the capturing module is placed on the client-side, however, we need to send the pattern captured to the server on every request for prefetching. Experimental results show, however, the overhead of shipping the pattern to the server constitutes only around 5 percent of the total processing time, becoming negligible. We can further minimize this overhead by caching the pattern in the server rather than sending the pattern from the client to the server at each prefetch request.

4.3.2 Maintenance of Type-Level Paths

To maintain the type-level path of an object o_{curr}, $\mathrm{TLP}(o_{curr})$, we use the smart pointer with an extension. The *smart pointer* is a data structure that can point either to the object in the cache or to the object in the database. Examples of smart pointers are ORRef [17] and odb_Ref [23]. This extension augments the data structure of the smart pointer pointing to an object to be able to store the pointer to the node in the CTLPG that corresponds to the type-level path of the object. To use smart pointers, we need to replace the input/output parameters (o_{curr} and o_{next}) of Prefetch-Navigate by smart pointers to these objects. The pointer to the root of the CTLPG is stored in the cursor of the navigational root set.

Example 3. Fig. 10 shows how we maintain type-level paths of objects using smart pointers. Suppose that, in accessing the object o_2 in Fig. 4, $\mathrm{Prefetch\text{-}Navigate(r, is\_taught\_by)}$ is called. Here, r is the smart pointer to the object o_2 and has a pointer to the node sections, which corresponds to the type-level path of o_2. Prefetch-Navigate augments the CTLPG by connecting the new node Professors through the attribute is_taught_by and stores a pointer to this new node in the output smart pointer r′.

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 four different experiments:

1. a navigational application described in Fig. 1 (*Iterative-Application*) having iterative patterns and one in Fig. 5 (*Recursive-Application*) having recursive patterns;
2. the OO7 benchmark [6], [21], which is a standard workload in object-oriented navigational applications. Here, we have used three different sizes of the database: small, medium, and large;
3. a geographical information system (GIS) application as the representative real-world application;
4. an XML database application for exporting XML data from the database.

Due to a space limit, we present experimental results for the XML database application in Appendix D which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm.

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

As the performance measures, we have used the relative number of fetches and the relative elapsed time.[7] To avoid the effects of noise and to increase the accuracy, we have run each experiment five times and averaged the results.

7. We have measured the number of fetches as the number of RPCs (remote procedure calls) between the client and the server and have counted all the RPCs, including transaction start and transaction commit, made by the application.

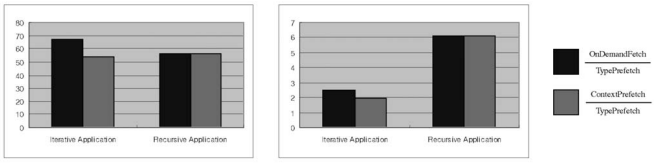

Fig. 11. Experimental results for iterative-application and recursive application. (a) Relative number of fetches and (b) relative elapsed time.

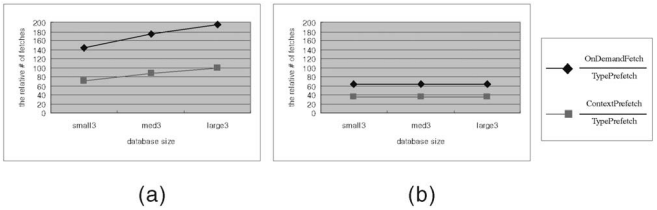

Fig. 12. The relative number of fetches for the OO7 benchmark. (a) Traversal T1. (b) Traversal T6.

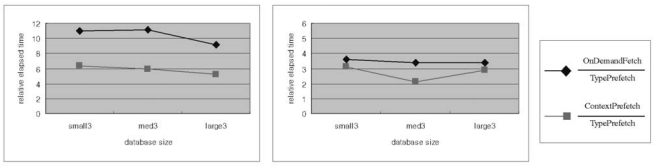

Fig. 13. The relative elapsed time for the OO7 benchmark. (a) Traversal T1. (b) Traversal T6.

5.2 Experimental Results

5.2.1 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. Here, for each professor object, five objects (address, car, drivetrain, engine, manufacturer) are connected, and the number of manufacturers of the qualifying professors' cars is 3. Thus, the number of objects accessed in the application is 1,003 (=200*5(professor, address, car, drivetrain, engine) + 3(three manufacturers)). The data set used for Recursive-Appilication is a linked-list similar to the one shown in Fig. 5. Here, 500 objects are connected from the root object in sequence.

Fig. 11 shows the experimental results for Iterative-Application and Recursive-Application. In both of these applications, TypePrefetch significantly outperforms not only OnDemandFetch but also ContextPrefetch. Compared with OnDemandFetch, TypePrefetch reduces the number of fetches by up to 67.4 times and improves the elapsed time by up to 2.49 times. Compared with ContextPrefetch, TypePrefetch reduces the number of fetches by up to 54.13 times and improves the elapsed time by up to 1.95 times. The analysis for the experimental results is in Appendix F which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm.

The relative elapsed time is smaller than the relative number of fetches 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 fetches, but the disk access time can not be reduced by prefetching.

Fig. 11 shows similar results for Recursive-Application. The relative elapsed 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 much more effective if objects accessed are well clustered in the server.

5.2.2 The OO7 Benchmark

Table 2 in Appendix E which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm summarizes the parameters of the OO7 benchmark database. The data sets we used (small3, med3, large3) consist of modules (root objects), a 7-level assembly hierarchy, and 500 composite part graphs connected from the leaf (base assembly) of the assembly hierarchy. The composite part graphs consist of atomic parts and connections. The data sets small3 and med3 are similar in all parameters except that med3 is larger than small3 by a factor of 10 in NumAtomic and Documentsize; similarly, large3 is identical to med3 except for large3 is larger than med3 by a factor of 10 in NumModules.

In this experiment, we have executed traversal operations defined in the OO7 benchmark. We briefly explain traversal operations in OO7; more details can be found at [6]. In the traversal T1, applications visits root modules, and then traverses the assembly hierarchy connected from the root. For each leaf-level assembly, applications traverse its composite parts connected from the leaf-level assembly. For each composite parts, applications traverse its atomic parts and connections. In the traversals T2a, T2b, T2c, T3a, T3b, and T3c, applications do the same traversal as T1, but update some of objects. In the traversal T6, applications traverse the assembly hierarchy and traverse only the root part of the composite part graph.

Fig. 12 shows the relative number of fetches of TypePrefetch for the OO7 benchmark. For the traversal T1, TypePrefetch reduces the number of fetches by up to 195 times compared with OnDemandFetch and by 97.8 times compared with ContextPrefetch. For the traversals T2a-T3c, the results, which are in Appendix G which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, are similar to those for T1 since the traversal patterns are as same as T1 except some updates. For the traversal T6, TypePrefetch reduces the number by up to 63.3 and 35.4, respectively. We have more improvement for T1 than for T6 since T1 accesses both the assembly hierarchy and the composite part graph while T6 does only the assembly hierarchy. There are significantly more repetitive type-level accesses in the composite part graph than in the assembly hierarchy.

Fig. 13 shows the relative elapsed time for the OO7 benchmark. For the traversal T1, TypePrefetch reduces the total elapsed time by up to 11.1 times compared with OndemandFetch and by up to 6.39 times compared with ContextPrefetch;[8] For T2a-T3c, the results, which are in Appendix G which can be found on the Computer Society

8. In the reference [2], 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 [24], which uses value matching. Thus, the performance results for T1-T6 for ContextPrefetch appears better in this paper than in the reference [2].

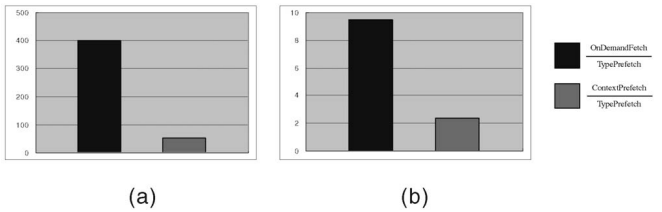

Fig. 14. Experimental results for a GIS application using a real data set.

Digital Library at http://computer.org/tkde/archives.htm, are similar to those for T1. For T6, TypePrefetch improves the total elapsed time by up to 3.57 times compared with OnDemandFetch and by up to 3.10 times compared with ContextPrefetch.

We note that, for the first measure (#of fetches), TypePrefeth leads to better improvement for the large3 database than the med3 database since there are more repetitive type-level accesses in the large database with the same schema. However, for the second measure (total elapsed time), the improvement for the large3 database is slightly worse than the med3 database. This is because the total elapsed time includes the query processing time, in particular, the disk access time to retrieve the objects from the database. Once the number of fetches improves significantly, the query processing time would become one of the dominant factors in the total elapsed time. Further improvement in the fetches has a marginal effect on the total elapsed time. However, the query processing cost highly depends on clustering, which is not controlled by TypePrefetch. Thus, to speed up further for large databases, we need a good clustering algorithm. As a future research, we are working on the use of the materialized view (well clustered for each query) in TypePrefetch.

5.2.3 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 10M 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.

Fig. 14 shows the results for this experiment. Compared with OnDemandFetch, TypePrefetch reduces the number of fetches by up to 402 times and improves the elapsed time by up to 9.50 times. Compared with ContextPrefetch, TypePrefetch reduces the number of fetches by up to 51.3 times and improves the elapsed time by up to 2.38 times. This indicates that type-level access locality indeed occurs in real-world GIS applications and that prefetching based on the type-level access pattern is effective in real-world applications as well.

6 Conclusions and Future Work

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 the number of fetches and improves elapsed time 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 ORDBMSs: the iterative pattern and the recursive pattern. Using these predefined access patterns, we have described the detailed mechanisms for capturing and prefetching and incorporated them in the algorithm.

We have performed extensive experiments using various types of data sets including the OO7 benchmark, a real GIS application, and an XML export 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 fetches by up to 402 times and improves the elapsed time by up to 11.1 times. Compared with context-based prefetching, the proposed method reduces the number of fetches by up to 97.8 times and improves the elapsed time by up to 6.39 times. Especially, the proposed method provides large improvements in the OO7 benchmark and in 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 significantly in navigational applications and is a practical method that can be implemented in commercial ORDBMSs.

As future work, we address the following issues. First, our algorithm can be extended to handle more general types of recursion. Currently, our algorithm handles iterative patterns and limited yet useful recursive patterns involving one cycle (linear recursion). Second, our algorithm can be extended to reduce the query processing time during prefetch. One possibility is to use the materialized view, which is well clustered for a certain query pattern.

Acknowledgments

An earlier version of this paper was presented in the *IEEE International Conference on Data Engineering (IEEE ICDE)* held in Heidelberg in 2-6 April, 2001. The paper has been significantly extended by adding the formal framework for capturing and prefetching (Sections 3.3, 4.1, and Appendix A (which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm), the detailed prefetching algorithm (Section 4.2), implementation issues (Section 4.3), and extended experiments for an XML application (Appendix D which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm). The most important contribution of extension in the *TKDE* submission is the formal framework for capturing and prefetching, which was somewhat preliminary and was not quite sufficiently elaborated in the ICDE version. A part of a preliminary version of this paper "PrefetchGuide: Capturing Navigational Access Patterns for Prefetching in Client/Server Object-Oriented/Object-Relational DBMSs" has appeared in Information Sciences [10]. This paper deals with an extension of a small part of the ICDE version constituting less than

45 percent of the contents of the current paper. The authors wish to acknowledge the contribution of Il-Yeol Song to the earlier version that was presented in ICDE 2001. This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc). This work was performed while Wook-Shin Han and Yang-Sae Moon were with the AITrc at KAIST.

REFERENCES

[1] E. Bertino et al., "Object-Oriented Query Languages: The Notion and The Issues," *IEEE Trans. Knowledge and Dat Eng.*, vol. 4, no. 3, pp. 223-237, June 1992.
[2] P.A. Bernstein, S. Pal, and D. Shutt, "Context-Based Prefetch for Implementing Objects on Relations," *Proc. 25th Int'l Conf. Very Large Data Bases,* pp. 327-338, 1999.
[3] R. Cattel and D.K. Barry, *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.
[4] E.G. Cooffman Jr. and P.J. Denning, *Operating Systems Theory.* Prentice-Hall, 1973.
[5] H. Common et al., *Tree Automata Techniques and Applications,* http://www.grappa.univ-lille3.fr/tata/tata.pdf, 1999.
[6] M.J. Carey, D.J. DeWitt, and J.F. Naughton, "The OO7 Benchmark," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 12-21, 1993.
[7] K.M. Curewitz, P. Krishnan, and J.S. Vitter, "Pratical Prefetching via Data Compression," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 257-266, 1993.
[8] E.E. Chang, R.H. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 348-357, 1989.
[9] W. Han, K. Whang, Y. Moon, and I. Song, "Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs," *Proc 17th IEEE Int'l Conf. Data Eng.,* pp. 651-660, 2001.
[10] W. Han, K. Whang, and Y. Moon, "PrefetchGuide: Capturing Navigational Access Patterns for Prefetching in Client/Server Object-Oriented/Object-Relational DBMSs," *Information Sciences,* vol. 152, nos. 1-4, pp. 47-61, 2003.
[11] W. Kim et al., "Architecture of the ORION Next-Generation Database System," *IEEE Trans. Knowledge and Data Eng.,* vol. 2, no. 1, Mar. 1990.
[12] W. Kim, *Introduction to Object-Oriented Databases.* The MIT Press, 1990.
[13] B. Liskov et al., "Safe and Efficient Sharing of Persistent Objects in Thor," *Proc. Int'l Conf. Management of Data,* pp. 318-329, 1996.
[14] C. Lamb et al., "The ObjectStore System," *Comm. ACM,* vol. 34, no. 10, pp. 50-63, 1991.
[15] T. Milo, D. Suciu, and V. Vianu, "Typechecking for XML Transformers," *Proc. ACM Symp. Principles of Database Systems,* pp. 11-20, 2000.
[16] Oracle Corp., *Oracle Call Interface Programmer's Guide Release 8.0.* 1997.
[17] C.M. Park, M.J. Carey, and S. Dessloch, "MAJOR: A Java Language Binding for Object-Relational Databases," *Proc. Eighth Int'l Conf. Workshop Persistent Object Systems,* 1998.
[18] Z. Palmer and S.B. Zdonik, "Fido: A Cache That Learns to Fetch," *Proc. 17th Int'l Conf. Very Large Data Bases,* pp. 255-264, 1991.
[19] A. Rosenthal et al., "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 166-176, 1986.
[20] M. Stonebraker and P. Brown, *Object-Relational DBMSs.* Morgan Kaufmann, 1999.
[21] M. Subramanian and V. Krishnamurthy, "Performance Challeges in Object-Relational DBMSs," *IEEE Data Eng. Bull.,* vol. 22, no. 2, pp. 27-31, 1999.
[22] R.G. Taylor, *Models of Computation and Formal Languages.* Oxford Univ. Press, 1998.
[23] UniSQL Inc., *UniSQL/X Application Program Interface Reference Guide.* 1995.
[24] K. Whang and R. Krishnamurthy, "Query Optimization in a Memory-Resident Domain Relational Calculus Database System," *ACM Trans. Database Systems,* vol. 15, no. 1, pp. 67-95, 1990.

Wook-Shin Han received the BS degree in computer engineering from Kyungpook National University in 1994 and the MS and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 1996 and 2001, respectively. He is currently an assistant professor in the Department of Computer Engineering at Kyungpook National University. His research interests include object-oriented/object-relational databases, XML databases, and information retrieval. He is a member of the IEEE and the ACM.

Kyu-Young Whang graduated (summa cum laude) from Seoul National University in 1973 and received the MS degrees from the Korea Advanced Institute of Science and Technology (KAIST) in 1975 and Stanford University in 1982. He received the PhD degree from Stanford University in 1984. From 1983 to 1991, he was a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights, New York. In 1990, he joined KAIST, where he currently is a full professor in the Department of Computer Science and the director of the Advanced Information Technology Research Center (AITrc). His research interests encompass database systems/storage systems, object-oriented databases, multimedia databases, geographic information systems (GIS), data mining/data warehouses, and XML databases. He is an author of more than 90 papers in refereed international journals and conference proceedings (and more than 140 papers in domestic ones). Dr. Whang served as an IEEE Distinguished Visitor from 1989 to 1990, received the Best Paper Award from the Sixth IEEE International Conference on Data Engineering (ICDE) in 1990, served ICDE seven times as a program cochair and vice chair from 1989 to 2005, and served on the program committees of more than 90 international conferences including VLDB and ACM SIGMOD. He was the program chair (Asia and Pacific Rim) for COOPIS'98 and the program chair (Asia, Pacific, and Australia) for VLDB 2000. He is the general chair of VLDB 2006, PAKDD 2003, and DASFAA 2004. He twice received the External Honor Recognition from IBM. Dr. Whang is an editor-in-chief of the *VLDB Journal* having served the editorial board as a founding member for 13 years. He was an associate editor of the *IEEE Data Engineering Bulletin* from 1990 to 1993 and an editor of *Distributed and Parallel Databases Journal* from 1991 to 1995. He is on the editorial boards of the *IEEE Transactions on Knowledge and Data Engineering* and *International Journal of Geographic Information Systems*. He was a trustee of the VLDB Endowment from 1998 to 2004 and currently is a steering committee member of the DASFAA Conference and the PAKDD Conference. He served the IEEE Computer Society Asia/Pacific Activities Group as the Korean representative from 1993 to 1997. Dr. Whang is a senior member of the IEEE, a member of the ACM, and a member of IFIP WG 2.6.

Yang-Sae Moon received BS (1991), MS (1993), and PhD (2001) degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST). From 1993 to 1997, he was a research engineer at Hyundai Syscomm, Inc., where he participated in developing 2G and 3G mobile communication systems. From 2002 to 2005, he was a technical director in Infravalley, Inc., where he participated in planning, designing, and developing CDMA and W-CDMA mobile network services and systems. He is currently an assistant professor at Kangwon National University. His research interests include data mining, knowledge discovery, storage systems, access methods, mobile/wireless comunication systems, and network comunication systems. He is a member of the IEEE and the ACM.