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 phenomenon 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' cars and addresses connected from each root object. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0218-1003.
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
\[ o_1, o_2, o_3, o_4, \ldots, o_{18} \].

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,
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.
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., \( o_{1+i}, o_{1+i+2}, \ldots \)) are connected from the object \( o_2 \), so as to completely fill the cache, then the objects prefetched, \( o_3 \sim o_{10} \), 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 \( o_2 \sim o_6, o_8 \sim o_{12}, o_{14} \sim o_{18}, \ldots \), 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 \( \Omega \). For example, the navigational root set in Fig. 1 is \( \{ o_1, o_7, o_{13} \} \) (we call this \( \Omega_1 \)). 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 \( a_0 \). 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 Fig. 1, the type-level path of \( o_1 \) is \( a_0 \); that of \( o_2, a_0\text{.}address \); that of \( o_3, a_0\text{.}owns\text{.}manufacturer \); and that of \( o_4, a_0\text{.}owns\text{.}drivetrain \).

The type-level path reference string for the navigational root set \( \Omega \) is a sequence of type-level paths referenced in accessing the objects that are connected directly or indirectly from \( \Omega \). For example, in Fig. 1, the type-level path reference string for \( \Omega_1 \) is

\[ [a_0, a_0\text{.}address, a_0\text{.}owns, a_0\text{.}owns\text{.}manufacturer, a_0\text{.}owns\text{.}drivetrain, \ldots, a_0\text{.}owns\text{.}drivetrain, a_0\text{.}owns\text{.}drivetrain\text{.}engine] \]

The type-level path graph for the navigational root set \( \Omega \) 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 \( \Omega \). 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

---

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

**Fig. 3.** An example university database schema.


TABLE 1
Summary of Notation

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>an object</td>
</tr>
<tr>
<td>o_i</td>
<td>the i-th object accessed</td>
</tr>
<tr>
<td>TLP(o)</td>
<td>the type-level path of object o</td>
</tr>
<tr>
<td>Ω</td>
<td>a navigational root set</td>
</tr>
<tr>
<td>CTLPG(Ω)</td>
<td>the current type-level path graph built so far for a navigation starting from Ω</td>
</tr>
</tbody>
</table>

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].
\]

**The properties of the operator \( \odot \):**

\[
[\ P_0] \odot [\ P_1, P_2, \cdots, P_k] = [\ P_0, P_0, P_1, P_0, P_2, \cdots, P_0, P_k].
\]

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

3.3.2 Iterative Patterns

ORDBMs 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, \( o_2, o_3, \) and \( o_5 \). For each section object, in turn, it accesses objects pointed by the values of the attributes `is_tought_by` and `has_TA`, finding professors and TAs for each section. The type-level path reference string for this navigation is

\[
[o_1, o_2, has_sections, o_3, has_sections, o_4, has_sections, o_5, has_sections, o_6, has_sections, o_7, has_sections, o_8, has_sections, o_9, has_sections, o_{10}, has_sections, o_{11}, has_sections, o_{12}, has_sections, o_{13}, has_sections].
\]

Note that the type-level paths, \( o_2, has_sections, o_3, has_sections.is_tought_by, o_4, has_sections, o_5, has_sections, o_6, has_sections.is_tought_by, o_7, has_sections, o_8, has_sections, o_{10}, has_sections, o_{11}, has_sections, o_{12}, has_sections, o_{13}, has_sections \), appear repeatedly. The string can be represented as

\[ A^+ \]

1. \( A^+ \) means \( A \) can appear one or more times.
If there are multiple root objects, the string will become as

\[ ([a_1] \cdot ([a_1, \text{has_sections}, a_1, \text{has_sections}.is\_taught\_by,\]
\[ a_1, \text{has_sections}.has\_TA])^+ \]

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):

\[ A' \rightarrow [a]^{iter(a)}. \] (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:

\[
\begin{align*}
A_0 & \rightarrow ([a_1] \circ ([] \cdot A_1))^{iter(a_1)} \\
A_1 & \rightarrow ([\text{has_sections}] \circ ([] \cdot B_1 \cdot B_2))^{iter(\text{has_sections})} \\
B_1 & \rightarrow [\text{is\_taught\_by}]^{iter(\text{is\_taught\_by})} \\
B_2 & \rightarrow [\text{has\_TA}]^{iter(\text{has\_TA})}
\end{align*}
\] (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], 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 \( a_1 \cdots a_j \) is cyclic if the type \( T_{a_{j-1}} \) referenced by \( a_{j-1} \) 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 \text{next}. Therefore, the object reference string is \( [a_1, a_2, a_3, \cdots, a_n] \), and the corresponding type-level path reference string is \( [a_1, a_0, \text{next}, a_0, \text{next}, \cdots, a_0, \text{next}] \times \text{next} \). In this type-level path reference string, the type-level paths, starting from the second one, contain cycles of the attribute \text{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.
The type-level path reference string for this navigation is represented as $A_0$, where the production rules are

$$\begin{align*}
A_0 &\rightarrow ([a_0] \circ ([ ] \bullet A_1))^{iter(a_0)}, \\
A_1 &\rightarrow ([next] \circ ([ ] \bullet A_1))^{iter(next)}.
\end{align*}$$

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 collection type and is involved in a cycle, the production rules for this case are as in (8):

$$\begin{align*}
A_0 &\rightarrow ([a_0] \circ ([ ] \bullet A_1))^{iter(a_0)}, \\
A_1 &\rightarrow ([has\_prerequisites] \circ ([ ] \bullet A_1))^{iter(has\_prerequisites)}.
\end{align*}$$

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:

$$([a_0] \circ ([address] \circ ([ ] \bullet (owns \circ (manufacturer \circ ([ ])))))) = a_0(address(), owns(manufacturer)).$$

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

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].

## 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

1. 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.

---

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

**Fig. 6.** An example where both an iterative and recursive patterns occur simultaneously. (a) The objects accessed. (b) The type-level path graph.
that follows a type-level path subgraph starting with the capturing process. Consider a navigational application rules that have been identified by the current time during

Current Type-Level Path Graph(CTLPG). CTLPG is an

iterative patterns and subsequent prefetching using the

level path) in the current object, we can produce a symbol (i.e., a type-level path reference string. That is, by accessing the value of

in the object reference string that corresponds to the type-

object (i.e., a valid pointer or null pointer to the child object)

the rule. To do so, we need to check the specific value of the

iter-rule, we also have to decide the number of iterations for

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 \( a_{i_1}..a_{i_j} \), where the attribute \( a_i \) is of a collection type. If the type-level path \( a_{i_1}..a_{i_j} \) 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 \( a_i \) 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

pattern captured. In summary, at the first iteration of a

Figure 7 shows the algorithm of Prefetch-Navigate, which

augments Basic-Navigate with 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 \( o_{curr} \) 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 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

accessed, the algorithm recognizes that the subpath \( a_i..a_j \)

completes a cycle. 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.

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.

5 For conservative prefetching, we can defer the decision on the recognition of a recursion until a cycle appears more than once.
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 \(o_{next}\), TLP\((o_{next})\) (= TLP\((o_{curr}, a)\)), is not in the CTLPG (line 5), it inserts 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 type-level path (lines 7-8). If the path 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 Prefetch-Objects, 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 MakeStateTransition, 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
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 elapsed 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. 9. The prefetch algorithm.**
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 IDs 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 \( o_{\text{curr}} \), TLP\( (o_{\text{curr}}) \), 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 odbRef [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 ( \( o_{\text{curr}} \) and \( o_{\text{next}} \)) 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, Prefetch-Navigate\( (r, \text{is\_taught\_by}) \) 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 ODISSEUS 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. 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.
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 part, 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. For 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.

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,
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), TypePrefetch 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 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


