Prefetch Support Relations in Object Bases

Carsten A. Gerlhof   Alfons Kemper
Universität Passau, Fakultät für Mathematik und Informatik,
D-94030 Passau, Germany
[*gerlhof* | *kemper*]@db.fmi.uni-passau.de

Abstract

In this paper we devise and assess a method for optimizing the execution of encapsulated operations (possibly with side-effects) based on the precomputation of the *page answer* (i.e., the set of referenced pages). For (important) parameter combinations, (1) the page answer, (2) the *reference frequency* of pages, (3) and the ordering of the page answer according to the first reference of a page during the execution of an operation are stored in a so-called *Prefetch Support Relation* (PSR). The PSR is used for two purposes: determining "good" candidates for prefetching and for replacement decisions. We devise the algorithms for maintaining the PSR in a consistent state and demonstrate selected benchmark results to assess the viability of the approach.

1 Introduction

In a recent paper [7], we investigated prefetching techniques in a client/server architecture. We concluded that prefetching yields a significant performance gain in situations of a highly utilized (page-)server; a reduction in running-time of 50% is not unrealistic. However, prefetching techniques crucially depend on the quality of their predictors [15, 4]. In [7] we investigated a page-oriented predictor as well as an object-oriented predictor and observed that page-oriented predictors exhibit a moderate prediction overhead but prediction quality often fails in case of poor clustering. In contrast, object-oriented predictors facilitate an excellent prediction quality but the overhead induced by an object-oriented predictor, especially in the case of good clustering, cannot be tolerated.

In this paper, we propose a new page-oriented predictor—called *Prefetch Support Relation* (PSR)—which combines the advantages of both techniques mentioned above. PSRs are very similar to the *Generalized Materialization Relations* (GMRs) introduced in [11, 12] which are used to store materialized (i.e., precomputed) function results. GMRs store the precomputed function result and are, therefore, only applicable for side-effect-free functions. On the other hand, PSRs facilitate support even for operations which do cause side effects, i.e., operations which modify the database. Instead of function results PSRs store the precomputed *page answer* of an operation, i.e., all pages which are referenced by a particular operation invocation.

The Prefetch Support Relation maintains a "compressed" list of pages for certain operation invocations with (particularly important) parameter combinations. This list is divided into (1) the raw page answer (i.e., the set of

```
type Vertex is
  public translate
  body [X, Y, Z: float;]
  operations
    declare translate: Vertex → void;
  implementation
    define translate(v) is
      begin
        self.X := self.X + v.X;
        self.Y := self.Y + v.Y;
        self.Z := self.Z + v.Z;
      end define translate;
end type Vertex;
```

```
type Cuboid is
  public volume, translate
  body [V1,···,V8: Vertex;
        Material: string, Weight: int]
  operations
    declare volume: → float;
    declare translate: Vertex → void;
  implementation
    define translate(v) is
      begin
        self.V1.translate(v);
        ...
        self.V8.translate(v);
      end define translate;
    ...
end type Cuboid;
```

Figure 1: Skeleton of the Type Definitions *Vertex* and *Cuboid*

referenced pages), (2) the frequency counts of pages, and (3) the ordering of the page answer according to the first reference of a page during the execution of an operation. The latter information is primarily utilized for prefetching. In addition, the frequency count associated with every page in the page answer can be exploited for making page replacement decisions. For this purpose, the buffer manager maintains actual usage statistics during the operation execution and compares them with the information stored in the PSR.

The paper is organized as follows. Section 2 gives the motivation and the definition of PSRs. In Section 3 we introduce our probability-based paging policy. Section 4 addresses the algorithms to maintain a PSR in a consistent state. In Section 5 we present some results of our quantitative analysis. Section 6 concludes the paper.

2 Static Aspects of Prefetch Support Relations

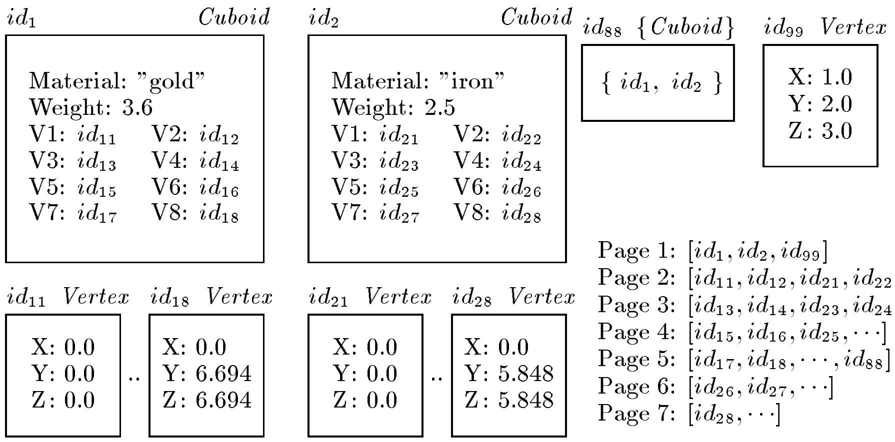
Figure 1 shows the definitions of the types *Cuboid* and *Vertex* in GOM [13] which serves as the running example throughout the remainder of this paper. A sample database and its corresponding (rather poor) mapping to pages is shown in Figure 2. The object identifiers are denoted by id_1, id_2, etc.

Now, assume that the following two program fragments are to be executed:

```
w : integer = 0;
foreach c in ext(Cuboid) begin
  w := w + c.volume;
end foreach;
```

```
extern persistent var v : Vertex;
foreach c in ext(Cuboid) begin
  c.translate(v);
end foreach;
```

During execution of either program all cuboids have to be visited and the function *volume* and the operation *translate* is invoked, respectively. For our sample database up to seven pages must be read from secondary storage. To expedite the evaluation of the first query the results of *volume* can be precomputed:



(a) Objects (b) Object Placement

Figure 2: Sample Database with Storage Structure

we call this the *materialization* of the function *volume* [11]. All function results are stored in a separate index structure called GMR. In this case we only need to read one page, namely the page containing the materialized results of function *volume*. However, materialization is only applicable for side-effect free functions.

Now, consider the second program fragment. In this case there is (a) no function result for *translate* and (b) the operation *translate* produces side-effects, i.e., it updates the attributes X, Y, and Z of all referenced vertices. Consequently, function materialization cannot be applied to support the second program fragment. Nevertheless, the operation result in terms of referenced pages (the so-called *page answer* of an operation) can easily be precomputed and stored in a separate index structure. Before invoking operation *translate* all pages containing the entire cuboid could then be requested within a single prefetch statement, thus, saving the (expensive) stepwise request for the subobjects of each cuboid:

foreach c **in** ext(Cuboid) **begin**
  prefetch($\langle\langle translate \rangle\rangle$, c, v);
  c.translate(v);
end foreach;

Clearly, because of the poor clustering, the above example is not very realistic—it is only chosen for its simplicity. A realistic application of PSRs would be in one of the following cases:

- Retrieval of large composite objects which cover many pages (as is the case in the OO7 benchmark [2]).
- Sequential scans over the extension of a specific type and its subtypes.
- Extensive traversals on a database with a high degree of sharing for which finding a good clustering is known to be very difficult [6].

Now let us define the Prefetch Support Relations as follows:

Definition 2.1 (Prefetch Support Relation)
Let $t_1, \cdots, t_{n+1}$ be types and let op be an operation with $op : t_1, \cdots, t_n \rightarrow t_{n+1}$. The prefetch support relation (PSR) $\langle\langle op \rangle\rangle$ for operation op is of arity $n + 3$ and has the following form:

$$\langle\langle op \rangle\rangle : \{ [O_1 {:} t_1, \cdots, O_n {:} t_n, \\ PA{:} \langle [pid : PageId, W_{pid} : int, L : Lock] \rangle, N{:} int, V{:} float] \}$$

The list of tuples in the page answer PA is ordered according to the first reference of page pid during the execution of operation op—the angle brackets $\langle \cdots \rangle$ denote a list and the square brackets $[\cdots]$ a tuple structure. □

The attributes $O_1, \ldots, O_n$ store the arguments, i.e., values if the argument type is atomic or references to objects if the argument type is complex; the attribute *PA* stores the entire page answer of an operation, i.e., the identifier of all pages that were accessed during the execution of that operation. The page answer is a list of tuples of the form $[pid, W_{pid}, L]$ where *pid* is the page identifier of a page, W_{pid} is the associated frequency count (see Definition 3.1), and L is the strongest lock on that page needed by operation *op*. This allows the server to set the appropriate lock upon the first request. If a page is already resident at the client, the server only sends an upgrade note to the client if the PSR tuple indicates a stronger lock for that page. Since clients have to piggyback information about dropped pages on other messages to the server due to the *Callback-Read* algorithm [5], this information is available to the PSR manager. The frequency count is used for paging, see Section 3 for details. The attribute V (standing for *validity*) indicates whether the stored results are currently valid,[1] the attribute N stores the cardinality of the page answer *PA*. In Section 3 it will become clear why we maintain an ordering of the page answer list. Operations that invalidate a precomputed operation are addressed in Section 4.

Example: The extension of the PSR $\langle\langle translate \rangle\rangle$ for the sample database in Figure 1 is depicted below. In this paper frequency counts are determined by precomputing operations in absence of pointer swizzling, thus every dot operation (.) in Fig. 1 generates a page reference.

$\langle\langle translate \rangle\rangle$				
$O_1{:}Cuboid$	$O_2{:}Vertex$	PA	$N{:}int$	$V{:}float$
id_1	id_{99}	$\langle(P_1, 32, S), (P_2, 12, X), (P_3, 12, X), (P_4, 12, X), (P_5, 12, X)\rangle$	5	1.0
id_2	id_{99}	$\langle(P_1, 32, S), (P_2, 12, X), (P_3, 12, X), (P_4, 6, X), (P_6, 12, X), (P_7, 6, X)\rangle$	6	1.0

◇

[1] Using a float value in the range of [0,1] instead of a boolean value allows us to express the validity as a probability.

3 Operation-Specific Probability-Based Paging

If the precomputed page answer of an operation is too large to fit entirely into the currently unoccupied buffer portion, we have to replace a page from the buffer when fetching the next page from the precomputed page answer. By exploiting PSRs we are able to devise a better page replacement policy than, for example, conventional LRU. Thus, we are able to set up a replacement policy which works for each operation separately, more precisely, for every parameter combination of an operation. Before we describe our probability-based paging policy in more detail, we briefly describe other paging information which could alternatively be stored in our PSRs (instead of the frequency counts):

- One could maintain the entire page reference string to compute the optimal replacement policy, but the costs for computing and storing this information in a PSR appear to be unacceptable for practical purposes.
- One could use a Markov-based paging policy [10] where the reference string of pages constitutes a Markov chain. All pages which are unreachable from the current state are candidates for replacement since these pages will never be accessed again.
- One could use Rete-like networks to cache selected subsets of entire page answers in order to support caching for multiple queries [9].
- One could use the page reference string to compute (and store) good values for the *Correlated Reference Period* and the *Retained Information Period* needed by the LRU-K algorithm [14].

In this section we outline a paging policy that is based on *frequency counts* for pages. Similarly to Markov paging we definitely know when a page is not used any more during an operation's execution. In fact, our buffering policy is a synthesis of prereleasing and prepaging. The implementation thereof is quite easy and is based on an efficient table lookup. Frequency-based page replacement has also been used successfully in [16].

Definition 3.1 (Weighted Page Answer)
Let $\langle pid_1, \cdots, pid_n \rangle$ be a reference string of page identifiers and let W_{pid} denote the number of occurrences of a page pid in the page reference string. The weighted page answer (WPA) is a list of tuples of the form (pid, W_{pid}) sorted by descending W_{pid}. □

The weighted page answer is a distribution of page access frequencies; the linear ordering is often called *probability ranking*. Based on the weighted page answer we can now define a probability-based paging policy which is derived from the IR model [3]. This will work well in practice if the ratio of the length of WPA to the number of buffer frames is not too large.

Definition 3.2 (WPA paging policy)
Let wpa be the weighted page answer of operation $op(o_1, \cdots, o_n)$ and let k be the number of buffer frames. Further, we define $\mathcal{P}_{max}(t)$ as the set of non-resident pages having the greatest W_{pid} value at time t. Then, the WPA paging policy for an operation $op(o_1, \cdots, o_n)$ whose precomputed page answer in a PSR is still valid is defined as follows:

- *On invocation of $op(o_1, \cdots, o_n)$ prefetch the first k non-resident pages according to the page answer stored in the PSR. The prefetch quantity k is determined at run-time and denotes the client's total number of available (i.e., free or unpinned) buffer frames at the time of the prefetch request.*
- *On a reference to page pid at time t do*
  1. *if $W_{pid} \geq 1$ update $W_{pid} := W_{pid} - 1$ in wpa.*
  2. *if $W_{pid} = 0$ then (pre-)release page pid by marking the corresponding page frame as free and prefetch a page from $\mathcal{P}_{max}(t)$.*
  3. *make page pid recently used.*
- *On a page fault at time t replace the least recently used page by the fault page and, optionally, replace up to $k-1$ resident pages pid_i which satisfy the condition $W_{pid_i} \neq 0 \wedge \exists pid \in \mathcal{P}_{max}(t) : W_{pid} > W_{pid_i}$, $1 \leq i < k$ by pages from $\mathcal{P}_{max}(t)$.* □

Example: Let $\langle P_1, P_1, P_2, P_1, P_2, P_3, P_4 \rangle$ be the page reference string and let $k = 2$. Thus, wpa = $\langle (P_1, 3), (P_2, 2), (P_3, 1), (P_4, 1) \rangle$. The replacement policy will become clear from the table below. Note that pages P_1 and P_2 are prefetched according to the ordering in the PSR, whereas pages P_3 and P_4 are prefetched according to the WPA.

buffer	——	$(P_1,3)$	$(P_1,2)$	$(P_1,1)$	$(P_2,1)$	$(P_3,1)$	$(P_4,1)$	$(P_4,1)$
contents	——	$(P_2,2)$	$(P_2,2)$	$(P_2,2)$	$(P_1,1)$	$(P_2,1)$	$(P_3,1)$	——
reference		P_1	P_1	P_2	P_1	P_2	P_3	P_4
prefetch	P_1, P_2				P_3	P_4		

◇

Paging Deficiencies. In general the IR model cannot compete with the more expressive Markov model because serial dependencies cannot be captured by the independent and identically distributed random variables of the IR model. Thus, we expect that there are cases for which the paging decisions made according to the WPA are not advantageous. For example, consider the reference string $\langle (P_1, \cdots, P_{k+1})^n, (P_x, P_y)^m \rangle$, $n \geq m$. If we have only k buffer frames, pages P_x and P_y will be unnecessarily fetched as soon as n drops below m. However, we expect that such "obscure" access patterns as sketched above are not very likely to happen in well-structured (mostly encapsulated) operations. If no such dependencies occur the prefetching policy can easily be proven to be optimal under the IR model [3]. Nevertheless, prefetching according to the IR model is optional in our paging policy and can be turned off at run-time.

4 Dynamic Aspects of Prefetch Support Relations

The main disadvantage of all predictors that are based on monitoring is that they generally provide no mechanism to check the validity of the collected information after a sequence of object updates. However, massively prefetching the wrong pages according to an "outdated" predictor dramatically degrades system performance. For this purpose, we investigate in this section the algorithms that are needed to check the validity of PSRs and to maintain them in a consistent state upon object base modifications.

In this section we describe the basic invalidation techniques which reduce the overhead significantly. For maintenance purposes, the PSR manager maintains so-called reverse references—in contrast to the Reverse Reference Relation proposed in [11] for precomputed function results—based on pages, therefore called *Reverse Page Reference Relation* (or RPRR in short). The RPRR contains tuples of the form $[pid,\ a,\ op,\ \langle o_1, \ldots, o_n \rangle]$. Here, pid is an identifier of a *page* which is utilized during the computation of the page answer of $op(o_1, \ldots, o_n)$ *and* this page contains an object with attribute a whose value has had some influence on the program flow of that operation. An attribute value can influence the program flow if that value is used inside a conditional statement during the execution of an operation (let us call this attribute *conditional attribute*) or because the attribute contains an object reference (a so-called *reference attribute*). An update of either type of attribute can change the navigation through the object base. The union of both sets of attributes constitutes the set of *relevant attributes*:

Definition 4.1 (Relevant Attributes) *Let* $op : t_1, \ldots, t_n \rightarrow t_{n+1}$ *be a precomputed operation and let* A *be an attribute identifier—for simplicity we assume that all attribute names are unique. Then, the set of relevant attributes is the union of the sets* $CondAttr(op)$ *and* $RefAttr(op)$ *which are defined as:*

$CondAttr(op) = \{A \mid$ *there exist objects* $o_1, \ldots, o_n$ *and an object* o *with attribute* A *such that* $o.A$ *has had some influence in an expression inside a conditional statement during execution of* $op(o_1, \ldots, o_n)\}$

$RefAttr(op) = \{A \mid$ *there exist objects* $o_1, \ldots, o_n$ *and an object* o *with attribute* A *such that* $o.A$ *contains an object reference and it is used during execution of* $op(o_1, \ldots, o_n)\}$ □

Thus, during precomputation we only need to insert pages into the RPRR containing objects that have relevant attributes. The RPRR is defined as follows:

Definition 4.2 (Reverse Page Reference Relation) *The Reverse Page Reference Relation RPRR is a set of tuples of the form*

$$[P\colon PageId,\ A\colon AttributeId,\ OP\colon OperationId,\ L\colon \langle OID \rangle].$$

For each tuple $r \in RPRR$ *the following condition holds: The page (with the identifier)* $r.P$ *has been accessed during the execution of operation* $r.OP$ *with the argument list* $r.L$ *and page* $r.P$ *contains an object such that for one of its attributes* $r.A \in CondAttr(r.OP)$ *or* $r.A \in RefAttr(r.OP)$ *holds.* □

The data-flow equations for computing the set of relevant attributes are given in the full paper [8]. Based on the RPRR we can now describe the algorithm for invalidating a page answer, i.e., the computations that have to be performed by

$\langle\langle$ *ScaleGoldIngots* $\rangle\rangle$			
O:{*Cuboid*}	*PA*	N:*int*	V:*float*
id_{88}	$\langle(P_1, 10, S), (P_2, 12, X), (P_3, 12, X), (P_4, 12, X), (P_5, 14, X)\rangle$	5	1.0

```
define ScaleGoldIngots(CuboidSet) is
var v : Vertex = (1.2, 1.2, 1.2);
begin
  foreach c in CuboidSet begin
    if c.Material = "gold" then
      c.scale(v);
    end if;
  end foreach;
end define ScaleGoldIngots;
```

```
define DetectGoldPlate(GoldIngot) is
begin
  if (GoldIngot.Weight / GoldIngot.Volume ≠ 19.3 ± ε) then
    GoldIngot.Material := "iron";
  end if;
end define DetectGoldPlate;
```

Figure 3: PSR $\langle\langle$*ScaleGoldIngots*$\rangle\rangle$ can be invalidated by *DetectGoldPlate*

the PSR manager when an object o has been updated. Because the RPRR contains only pages—remember the Reverse Reference Relation of GMRs contains objects—we are able to substantially reduce the overhead (otherwise associated with GMRs), i.e., we do *not* immediately invalidate a PSR at the time when an object becomes updated. We only record the identifiers of all updated attributes into the header of the buffer frame of the updated page. When a modified page has to be sent back to the server, the set of attribute identifiers is also sent along with the page in the attachment of the message. Then, the server invalidates all page answers which match a tuple in the RPRR containing that page and any of the updated attributes.

Example: Consider the PSR for the operation *ScaleGoldIngots* in Fig. 3. The vertices that will be referenced during the execution of *ScaleGoldIngots* depend only on the value of the attribute *Material*, hence $CondAttr(ScaleGoldIngots) = \{Material\}$. Note that $id_1.translate(v)$ does not affect the page answer of *ScaleGoldIngots* in any way. However, $DetectGoldPlate(id_1)$ would directly invalidate the page answer of $ScaleGoldIngots(id_{88})$ if cuboid id_1 were detected as painted gold, i.e., "degraded" from gold to iron. $\diamond$

The main advantages of this invalidation algorithm are that we (1) pay no additional I/O costs at the time of an update and (2) concerning the PSR all modifications of objects which reside on the same page are considered only once—in a bulk. In conclusion, the overall overhead for updates is reduced to a tolerable extent—as is also verified by the quantitative analysis of Section 5. The disadvantage is that we increase the granularity by giving up the object level. However, the precision of determining the exact set of operations whose page answers are invalid depends only on the page cardinality (in objects). Note that a page answer will only be invalidated erroneously when a modified page contains an object of the *same* type whose attribute is relevant and becomes updated.

Object Moving. When an object is moved from page p to page p' and page p is in the page answer of some operation *op* we generally do not know whether the correct page answer now contains p, p', or both. For this reason we don't want to replace each occurrence of p by p' in the PSR. Instead, we just add page p' to all page answers which include page p—the same has to be done for the corresponding entries in the RPRR. The new frequency count $W_{p'}$ is copied from that of page p, the old frequency count W_p is decreased by one. Additionally, we *weaken* the validity for all page answers containing page p by setting $V := V - 1/N$ where N is the cardinality of the original set of pages. However, if the probability for the validity reaches a low "water mark", the page answer will be finally invalidated.

Initial Setup. PSRs can be created for all operations which do not immediately invalidate themselves. The buffer manager of the client's run-time system monitors all page accesses, maintains the WPA statistics, and finally sends all WPA information (including page answer and statistics) to the server. This can be done either on-line or—to initially support operations with database updates—this can be done off-line using slightly modified operations in which all statements with side-effects are "neutralized".

5 Quantitative Analysis

In this section we present some typical benchmark results for PSR retrieval (Section 5.1) and for PSR update (Section 5.2).

We implemented the PSR manager in our multi-threaded server engine introduced in [7]. The PSRs are implemented by a hash table using *Filtered Hashing* [1] since in practice the keys, i.e., the combination of parameter values, are highly correlated. The default PSR bucket size is configured to 1KB, the initial size of both hash tables is 10 000 buckets. There is full support for concurrency and R1–R3 recovery[2] on PSRs.

5.1 PSR Retrieval

We generated a database segment with 10 000 pages of 4KB size. The PSR contained up to 10 000 entries with varying sizes of the page answer. The client fetched the entire database segment into it's buffer and referenced each page one time. As reported in [7] a highly utilized server is the overall bottleneck, thus we again conducted multi-client experiments to emphasize the need for prefetching in a client/server-environment. Fig. 4 shows the results for a varying number of additional heavy I/O-weighted clients (plotted on the x-axis) which operated on a segment which is different from the segment we prefetched, and varying prefetch cardinalities (indicated by curves labeled with *pref #*). All experiments were started "cold", i.e., with empty buffer caches (both the client and the server caches).

The most important result of Fig. 4(a) is that the break-even point in case of no additional clients can be found at a page answer of at least 6 pages. For smaller page answers the use of PSRs is counterproductive. However, this

[2] Only a page-oriented recovery technique is currently implemented in our system.

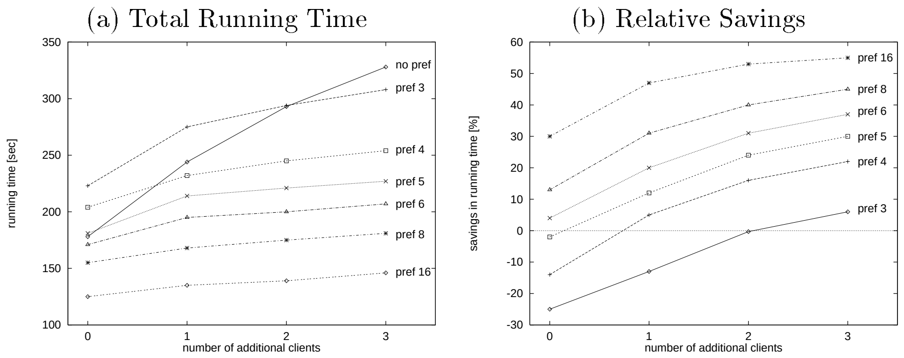


Figure 4: Performance of PSR Retrieval

changes as more clients join the server (witness curves *pref 3*, *pref 4*, and *pref 5*). Fig. 4(b) exhibits that the relative savings in total running-time compared to no prefetching can easily reach more than 50% when prefetching 16 pages.

5.2 PSR Update

In this experiment we designed a typical CAD-transaction where we fetched approximately 1 000 pages from the server (with and without prefetching) into the client's buffer and then updated one relevant attribute on a varying number of pages—this relative number is plotted (in %) on the x-axis. The relative savings to no prefetching are plotted on the y-axis. The RPRR recorded 1 000 pages with one relevant attribute each. We assume a lazy rematerialization for PSRs. Fig. 5 shows the results for a page answer of 6 and 16 pages.

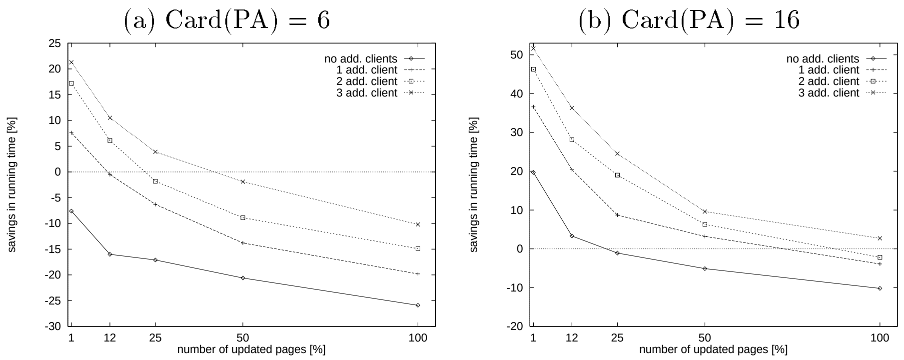


Figure 5: Performance of PSR Update

Both diagrams show that if we increase the number of updated pages the performance gain by prefetching smoothly decreases. This is because (1) the buffer hit rate for both, PSR pages and RPRR pages, is very high when performing a bulk update, and (2) due to the sequential access pattern many updated pages are already asynchronously written to disk at commit time. We also see that the break-even point significantly depends on the server utilization. In Fig. 5(a) server utilization must be very high to remedy the additional penalties induced by PSR update. However, Fig. 5(b) exhibits that 25% updates in case of low server utilization and 100% updates in case of high server utilization can be tolerated when prefetching 16 pages.

6 Conclusion

In this paper we devised a method for optimizing the execution of encapsulated operations based on the precomputation of the *page answer* (i.e., the set of referenced pages). For (important) parameter combinations, (1) the page answer, (2) the *reference frequency* of pages, (3) and the ordering of the page answer according to the first reference of a page during the execution of an operation are stored in a so-called *Prefetch Support Relation* (PSR). The PSRs were used for two purposes: determining "good" candidates for prefetching and for replacement decisions. We also devised and benchmarked the algorithms for the consistency of the PSRs under database modifications.

Our quantitative analysis indicates that in case of low server utilization one can expect a clear gain in running time if the page answer contains at least 6 pages. Operations which modify the database are not significantly penalized by the additional overhead incurred by maintaining the PSRs if relevant information on only a few pages becomes updated. Note that all techniques in [12] which are developed to reduce the invalidation overhead for materialized functions can be adapted to PSRs.

So far we have only considered (completely) consistent page answers in the PSRs. However, even (partially) invalid page answers are very useful for prefetching decisions—as long as the validity of the page answer does not drop below a certain threshold. In the future, we want to apply fuzzy logic to model the uncertainty of page answer consistency after database updates. Further, we will spend more work on static code analysis to facilitate finer grained control over the invalidation of PSRs. We will also investigate alternative paging policies which can make better use of the access patterns stored in a PSR.

Acknowledgement This work was supported by the German Research Council DFG under contract Ke 401/6-1. We thank Florian Waas for the prototypical implementation and the extensive benchmarking of various paging policies.

References

[1] I. Ahn. Filtered hashing. In *Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, volume 730 of *Lecture Notes in Computer Science (LNCS)*, pages 85–100, Chicago, IL, October 1993. Springer-Verlag.

[2] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 12–21, Washington, DC, USA, May 1993.

[3] E. G. Coffman and P. J. Denning. *Operating Systems Theory.* Prentice Hall, Englewood Cliffs, NJ, USA, 1973.

[4] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 43–53, Washington, DC, USA, May 1993.

[5] M. J. Franklin, M. J. Carey, and M. Livny. Local disk caching for client-server database systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 543–554, Dublin, Ireland, August 1993.

[6] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. In *Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, volume 730 of *Lecture Notes in Computer Science (LNCS)*, pages 301–316, Chicago, IL, October 1993. Springer-Verlag.

[7] C. A. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. In *Proc. of the Intl. Conf. Extending Database Technology (EDBT)*, volume 779 of *Lecture Notes in Computer Science (LNCS)*, pages 351–364, Cambridge, England, March 1994. Springer-Verlag.

[8] C. A. Gerlhof and A. Kemper. Prefetch Support Relations in Object Bases: Design, Implementation and Assessment. Technical report, Universität Passau, 94030 Passau, Germany, 1994. (forthcoming).

[9] N. Kamel and R. King. Intelligent database caching through the use of page-answers and page-traces. *ACM Trans. on Database Systems*, 17(4):601–646, December 1992.

[10] K. R. Karlin, S. J. Phillips, and P. Raghavan. Markov paging. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 208–217, October 1992.

[11] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 258–268, Denver, USA, May 1991.

[12] A. Kemper, C. Kilger, and G. Moerkotte. Function Materialization in Object Bases: Design, Implementation and Assessment. *IEEE Trans. Knowledge and Data Engineering*, 6(4), August 1994.

[13] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[14] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 297–306, Washington, DC, USA, May 1993.

[15] M. L. Palmer and S. B. Zdonik. FIDO: A cache that learns to fetch. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 255–264, Barcelona, September 1991.

[16] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. of the ACM SIGMETRICS*, pages 134–142, 1990.