# Reducing the Cost of Perfect Match Accesses in Object Database Systems

Kjetil Nørvåg

Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
Kjetil.Norvag@idi.ntnu.no

**Abstract.** In many application areas for object database systems (ODBs), the access pattern is navigational, and a large fraction of the accesses are perfect match accesses/queries on one or more words in text strings in the objects. One example of such an application area is XML data stored in ODBs. In this paper, we give an overview of SigCache approach, which can significantly reduce the average object access cost for such queries in ODBs, with only a marginal increase in update the costs. Although the discussion in this paper is done in the context of ODBs, the SigCache approach can also be employed in relational and object-relational database systems experiencing a navigational access pattern.

## 1 Introduction

In many of the emerging application areas for database systems, data is viewed as a collection of objects and the access pattern is navigational. A typical example of such an application area is XML/Web storage. XML data should preferably be stored in an object database system (ODB) [9], but it can also be stored in a relational database system or an object-relational database system.

A typical characteristic of the new applications is that a large fraction of the accesses are perfect match accesses on one or more words in text strings in the objects/tuples in these databases. For such accesses, *signature*[1] *files* can be used to reduce the query cost.

The main drawback of traditional signature files is that signature file maintenance can be relatively costly. If one of the attributes contributing to the signature in an object (or a tuple) is modified, the signature file has to be updated as well. In order to be beneficial, a high read to write ratio is necessary. This is also the case for dynamic signature files. In addition, high selectivity is needed at query time to make it beneficial to read the signature file in addition to the objects themselves.

In this paper, we give an overview of the *SigCache* approach. Instead of storing the signatures in separate signature files, the signatures are stored together with the objects, and the most frequently accessed signatures are in addition stored in a *signature cache* (SigCache). A signature is in general much smaller than an object, so that the number of signatures we can keep in the SigCache is much higher than the number of objects we can store in the main memory buffer. When an object is updated and the signature is stored on the same page, the extra insert cost is only marginal. The signatures can also be used to reduce the CPU cost when the objects are already in memory. In addition to a detailed description of the use and maintenance of the SigCache, we also show some results from a performance analysis. As for all access methods, the gain depends on access patterns. We show that the gain from using the SigCache approach is significant for most access patterns.

[1] A signature is a bit string, which is generated by applying some hash function on some or all of the attributes of an object. Note that *signatures* are also often used in other contexts, e.g., function signatures and implementation signatures.

The organization of the rest of the paper is as follows. In Section 2 we give an overview of related work. In Section 3 we give a brief introduction to signatures. In Section 4 we describe the SigCache approach. In Section 5 we show some performance results based on the use of cost models. Finally, in Section 6, we conclude the paper and outline issues for further research.

## 2 Related work

Several studies have been done on using signatures as a text access method, e.g. [2–4, 6]. Less has been done in using signatures in ordinary query processing, but signature file techniques have been shown to be beneficial in queries on set-valued objects [5].

Also related to the work in this paper is XML extensions to commercial database systems (for example Oracle and IBM DB2). These systems currently use text indexes (provided by text extenders) to speed up queries. There are also systems storing XML data as objects, for example Tamino [8] and Xyleme [1]. They should be able to benefit from the techniques described in this paper.

## 3 Signatures

In this section we describe how to generate signatures, how to use signatures to reduce the cost of perfect match accesses (PMA), and signature storage alternatives.

### *3.1 Signature generation*

A signature can be generated by applying a hash function on some or all of the attributes of the object. By applying this hash function, we get a signature of $F$ bits. If we denote the attributes of an object $O_i$ as $A_1, A_2, \ldots, A_n$, the signature of the object is $s_i = S_h(A_j, \ldots A_k)$, where $S_h$ is a hash value generating function, and $A_j, \ldots A_k$ are some or all of the attributes of the object (not necessarily including all of $A_j, \ldots A_k$).

It is possible to generate the signature from the hash value of the concatenation of one or more of the attributes. However, such signatures can only be used for queries on the same set of attributes that were used to generate the signature. For this purpose, using an index will in many cases have a lower cost. In order to be able to support several query types, that do perfect match on different sets of attributes, a technique called *superimposed coding* can be used. In this case, a separate attribute signature is generated for each attribute. The object signature is generated by performing a bitwise OR on each attribute signature. For example, for an object with 3 attributes the signature is $s_i = S_h(A_0)$ OR $S_h(A_1)$ OR $S_h(A_2)$. This results in a signature that is very flexible in use. It can support several types of queries, with different attributes.

It is not necessary to use all attributes when creating the superimposed signature. If we know that only $D$ of the attributes will be frequently used in PMAs, we can generate the signature from these attributes only. In the case of string attributes, for example in an XML object, we will generate a separate signature from each distinct word in the string attributes, and superimpose these word signatures. $D$ will in this case be the number of distinct words in the object.

### 3.2 Using signatures

A typical example of the use of signatures, is a query to find all objects in a collection of objects where the attributes match a certain number of values, i.e., the query predicate is $Q = (A_j = v_j, \ldots, A_k = v_k)$. This can be done by calculating the query signature $s_q$ of the query: $s_q = S_h(A_j = v_j, \ldots, A_k = v_k)$ (or $s_q = S_h(v_j)$ OR $S_h(v_i)$ OR ... OR $S_h(v_k)$ if superimposed coding is used) The query signature $s_q$ is then compared to all the signatures $s_i$ in the signature file to find possible matching objects. A possible matching object, a *drop*, is an object that satisfies the condition that all bit positions set to 1 in the query signature, also are set to 1 in the object's signature. The drops forms a set of candidate objects. An object can have a matching signature even if it does not match the values searched for, so all candidate objects have to be retrieved and matched against the value set that is searched for. The candidate objects that do not match are called *false drops*.

### 3.3 Signature files

Traditionally, the signatures have been stored in one or more signature files, separate from the objects/tuples. The files contain $s_i$ for all objects $O_i$ in the relevant set. The sizes of these files are in general much smaller than the size of the relation/set of objects that the signatures are generated from, and a scan of the signature files is much cheaper than a scan of the whole relation/set of objects. Two well-know storage structures for signatures are *Sequential Signature Files* (SSF) and *Bit-Sliced Signature Files* (BSSF).

In the simplest signature file organization, SSF, the signatures are stored sequentially in a file. A separate *pointer file* is used to provide the mapping between signatures and objects. In an ODB, the pointer file will typically be a file with OIDs, one for each signature. During each search for perfect match, the whole signature file has to be read. Updates can be done by updating only one entry in the file.

With BSSF, each bit of the signature is stored in a separate file. With a signature size $F$, the signatures are distributed over $F$ files, instead of one file as in the SSF approach. This is especially useful if we have large signatures. In this case, we only have to search the files corresponding to the bit fields where the query signature has a "1". This can reduce the search time considerably. However, each update implies updating up to $F$ files, which is very expensive. So, even if retrieval cost has been shown to be much smaller for BSSF, the update cost is much higher, 100-1000 times higher is not uncommon [5]. Thus, BSSF based approaches are most appropriate for relatively static data.

To better support insertions, deletions, and updates, several dynamic signature file methods have been proposed. These are multiway tree variants and hash file variants.

## 4 The SigCache approach

An alternative to store the signatures in separate signature files is to store them together with the objects on the object pages, and in addition store the most frequently accessed signatures in a *signature cache* (SigCache). This is illustrated in Figure 1. The SigCache is a lookup table where replacement of signatures is done according to an LRU policy. In this paper we assume that a clock algorithm with one access bit for each signature is used for this purpose.

A signature is stored on the same page as its object. This implies that if an object is resident in main memory (i.e., the page where the object resides is resident in the page buffer), its signature will also be resident, because it is stored in the same page. However,

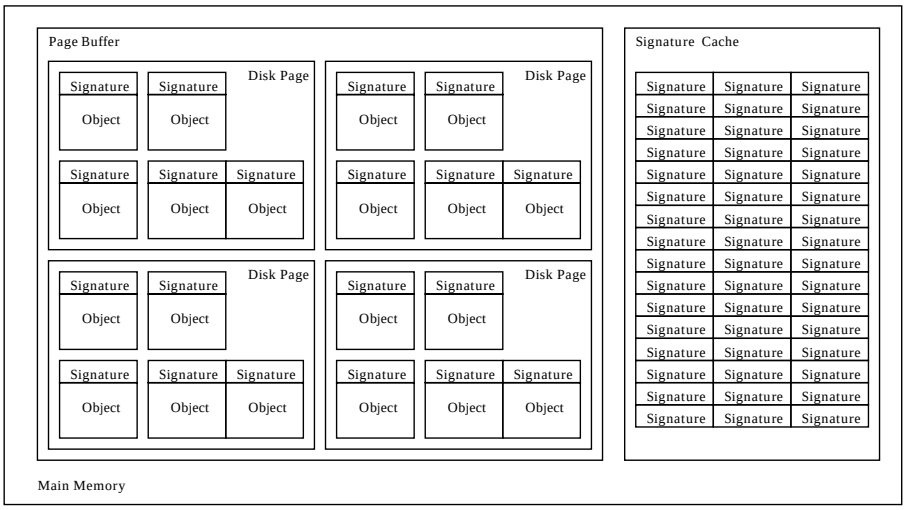


**Fig. 1.** Storage of object pages and signatures in main memory. Page buffer to the left, and signature cache (SigCache) to the right.

the opposite is not true: a signature can be resident in the SigCache even though its object is not resident in the buffer. When a page is discarded from the page buffer, signatures of some of the objects on the page can be resident in the SigCache.

Perfect match accesses can use the signatures to reduce the number of objects that have to be retrieved from disk or another node. Only the candidate objects, with matching signatures, need to be retrieved. In order to identify a signature in the SigCache each signature need to have an unique identifier. In an ODB this will be the OID, in a relational- or object-relational database system this can be a physical tuple identifier or the concatenation of relation and key.

A signature is in general much smaller than an object, so that the number of signatures we can keep in the SigCache is much higher than the number of objects we can store in the main memory buffer. The fact that the effective space utilization in an object page buffer is low because of bad clustering on object pages further increase the amount of relevant signatures relative to relevant objects. The optimal size of the SigCache depends on access pattern and total buffer size relative to the total database size. For optimal performance, the SigCache size can have a size that is adaptively changed by using the cost model presented in [7]. However, even a fixed SigCache size can give a relatively stable performance for a wide range of workload parameters [7].

Signatures are not maintained for all objects in the system, only when it is beneficial. This can for example be decided on the granularity of an object class or object container (also called file). Even when signatures exist, they are only used in a query if it is considered beneficial with respect to query cost. This is similar to traditional secondary indexes, where an index is created and maintained only if it is considered beneficial (this is decided by the database administrator), and the index is only used in a query if it is considered beneficial (this is decided during query planning).

### *4.1 The advantages of the SigCache approach*

The SigCache approach has many advantages. The most important are:

- Traditionally, signatures have only been beneficial for relatively static data (i.e., a low update rate), for example text documents. Even when dynamic signature files

are used, the update rate must be low if the use of signatures should improve performance. Dynamic signature files also have higher space requirements and search cost compared to SSF and BSSF.
In contrast to using separate signature files, the SigCache approach is also useful in the case of high update rates. A signature is in general much smaller than the object it is created from, so that when an object is updated and the signature is stored on the same page, the extra insert cost is only marginal. If only a moderate amount of main memory is used for the SigCache, the page buffer hit rate is only marginally reduced.

- Read accesses in a relational database system are frequently set accesses which can benefit from traditional signature files. In contrast, read accesses in ODBs are mostly navigational. Even in the case of collection (for example a set) queries, navigation will often be the result. Unlike a relational database system where the queried set is a relation on storage, in an ODB, a collection can be a collection of references to objects (OIDs) rather than the objects themselves (an object can in this way belong to more than one collection). This navigation can make the average signature retrieval cost high if the signatures are stored in separate files. If the most frequently accessed signatures are stored in the SigCache, this cost can be significantly reduced.
- Previously, signatures have mostly been used to reduce the I/O-costs. However, signatures can also be used to reduce the CPU costs. Even if an object is resident in main memory, a signature comparison can be used before matching the attributes. Especially in the case of many or large attributes, this can reduce the CPU cost for a PMA.

### *4.2 Query processing using the SigCache*

When a PMA is done on one or more attributes of an object, the following algorithm is used to determine if an object $O_i$ is a match, and at the same time maintaining the contents of the SigCache:

1. A lookup is done for the object's signature $s_i$ in the SigCache. If successful, the *access bit* of the signature in the SigCache is set.
   If this lookup is not successful, the signature has to be retrieved from the page where the object is stored. This page may be resident in the page buffer, but if it is not, the page has to be retrieved from disk or from its home node. When the page is found, the signature $s_i$ which is stored on the page, is inserted into the SigCache. The access bit for the signature is set when the signature is inserted.
2. The signature $s_i$ is compared to the query signature $s_q$. If not all bit positions set to 1 in $s_q$ are set to 1 in $s_i$, we know for sure that the object does not match the query predicate. If all bit positions set to 1 in $s_q$ are set to 1 in $s_i$ the object is a possible match, and we have to retrieve the object in order to compare the value of its attributes with the query predicate. The page where the object is stored on might already be in the page buffer (because it has been accessed recently, or has been brought into the buffer in order to retrieve the signature of one of the objects on the page), if not, the page has to be retrieved from disk or from its home node.

A query on a collection of objects is done in the same way, once for each object. Also note that by employing signatures as outlined in the algorithm above, the CPU cost can also be reduced, because the signatures are compared before the object is compared with the search predicate.

### *4.3 Scan operations*

One single scan operation can make all the signatures stored in the SigCache to be replaced. This is often not desired. One reason for this, is that the signatures retrieved during a scan operation will in general have less chance of being used again, it is not likely that the whole collection or container to be scanned represents a hot set. Even if this is the case, it is possible that the number of signatures retrieved during the scan is larger than the number of signatures that fits in the SigCache. In this case, even if we shortly after do a new scan over the collection/container, we will have a SigCache hit probability of 0. This is similar to general buffer management in the case of scan operations.

We have several possible strategies to use in order to avoid the problem with scan operations involving a large number of objects. One strategy is to not insert signatures retrieved in a scan operation into the SigCache (but the signatures already stored in the SigCache can be used when appropriate). If this strategy is used, we avoid the SigCache pollution, but at the same time we loose opportunities to benefit from the signatures, as many of these scan operations will involve a small number of objects. A variant of this strategy is to insert signatures into the SigCache if the scan operation involves a small or moderate number of objects, but not insert signatures if the scan involves a large number of objects.

It is often difficult to know the number of objects involved in a scan. It will probably be more safe to simply limit the number of signatures from objects of a certain class to be stored in the SigCache. For example, to define that only 25% of the slots in the SigCache can be occupied by one class. This can be achieved by maintaining the number of signatures of objects from each object class in the SigCache. In many page servers the object class is not known to the server. In this case, the limitation can be the number of objects from a particular container/file (the container/file identifier is encoded into the OID in most ODBs).

### *4.4 Object updates and SigCache maintenance*

Every time an object is modified, its signature has to be modified as well. A signature is stored together with its object, and with respect to concurrency control, logging and recovery, the signature is treated as a part of the object. If the signature of the object is resident in the SigCache, the signature in the SigCache has to be updated as well. This implies that a SigCache lookup has to be done for each object update. However, compared to the number of CPU instructions necessary to provide persistent storage of an object, this lookup cost is only marginal.

Only signatures that can contribute in read queries are beneficial to keep in the SigCache, so that when a signature is updated in the SigCache, the access bit is not set. Read accesses are necessary to make a signature stay in the SigCache.

### *4.5 Redundant signatures in main memory*

A signature is stored in the same page as its object, so that if the object of a signature in the SigCache is resident in main memory, the signature is actually stored two places. In order to optimize memory usage, it is possible to only store signatures of objects that are not resident in main memory in the SigCache. This can be done by removing all signatures of objects in a page from the SigCache when the object page is retrieved into the page buffer. However, this is not a good strategy:

1. Removing and reinserting all signatures from a page significantly increases the CPU cost.
2. When a page is to be discarded from the page buffer, it is difficult to know which signatures should be reinserted into the SigCache (it is difficult to maintain the LRU policy). The only easy solution is to insert all the signatures of the objects on the page into the SigCache. However, in general only a few of the signatures on a page are "hot spot signatures", so this will pollute the SigCache. The result will be a serious reduction in the SigCache hit ratio, effectively killing all benefits from this memory "optimization".

The second problem in particular is very serious, so we do not consider this strategy any further.

### *4.6 Signatures and object-orientation*

An object is not necessarily just a collection of simple value attributes as a tuple in a relational database system. An object can contain methods and references to other objects, and the objects in a queried collection can be objects of different classes, due to the concept of inheritance. We now describe how these issues can be handled with respect to signatures.

*Methods.* The use of signatures is quite straightforward in queries only involving value attributes. However, in an ODB it is also possible to access data from a general programming language, for example C++ or Java, and through method calls in queries (although not all vendors provide this functionality in their query languages).

Although the use of signatures is most beneficial in queries on value attributes, they can also be used in methods. One solution is to extend the equality operator in the programming language to compare the signatures before comparing the values. The advantage with this solution is that the use of signatures is transparent to the application programmer. Another solution is to explicitly use signatures by a separate function call before doing the comparison in the general programming language.

*Complex objects and path expressions.* When an object signature is generated, its attributes are simply treated as bit strings. If an attribute is an OID (a reference to another object), the OID is simply hashed like any other attribute.

Although not considered in this paper, a value on a path expression could be used in the signature generation process, similar to path expression indexing. For example, it could be possible to define that instead of simply hashing a reference attribute `objPtr`, the value of `objPtr->objPtr2->attrib` should be hashed instead. This would significantly increase the cost and complexity of signature creation, because every time `attrib` is updated, all signatures based on the value of `attrib` have to be updated as well. However, if queries on path expressions are common, this approach could still be beneficial.

*Inheritance.* Signatures for objects of two object classes `A` and `B` are *compatible*, i.e., can be compared, if 1) the attributes used for creating signatures for objects of one of the classes are a subset of the attributes used for creating signatures for objects of the other class, 2) the signature size is the same, and 3) the same hash function is used. This can be the case for signatures of a superclass and one of its subclasses.

Frequently, we want a larger signature size for a subclass when the number of attributes is higher. If this is the case, and we do a query on a collection of objects from the superclass as well as the subclass, a separate query signature has to be generated for each subclass (but note that only one signature for each object is generated and stored). When comparing the signature of an object with a query signature, the appropriate query signature is used, based on the class the object belongs to.

Example: Assume we have a class A, a class B that inherits from class A, and that we use a different signature sizes for class A and B. If we do a query over a collection of A objects, this collection can also contain B objects, because a B object is also an A object. When the query is executed, two query signatures are generated, one query signature $s_q^A$ for objects of class A, and one query signature $s_q^B$ for objects of class B.

### *4.7 Signature recalculation*

It is not strictly necessary to store the signatures of the objects on disk. The signatures can be recalculated when the objects are retrieved. This saves disk space and increases the page buffer hit rate (because the total number of object pages is less), but increases the CPU cost. CPU speed is increasing at a much higher rate than disk performance, so even if signature recalculation instead of signature storage is not beneficial today, it is likely that this alternative will become attractive in the near future. In the analytical study in this paper we will assume that the signatures are stored on disk, but in [7] we have also studied how much the I/O costs can be reduced by using recalculation.

### *4.8 Optimal signature size*

The signature size is a tradeoff. Using large signatures reduces the false drop probability, but large signatures also reduces the number of signatures that can be kept in the SigCache. Too large signatures will also break the assumption that signatures are much smaller than the objects, and in that way increase the signature maintenance cost. The signature size should be chosen so that it minimize the average object retrieval cost for a large range of parameter values.

When deciding the signature size it is also important to keep in mind that the signature size can not easily be changed afterwards if it is too small. If this should be done, reorganizing the database is necessary because there is not reserved space for larger signatures on the object pages.

### *4.9 SigCache vs. indexes*

Another alternative to the SigCache is to use traditional indexing techniques. Indexes have a higher update and storage cost, and are also less suitable for text/multi-attribute search. However, in the case of workloads with low update rates, or smaller amounts of perfect match accesses, using indexes will be beneficial. In this case, indexes *as well as signatures* can be maintained. Cost functions or tuning can be used to resize the SigCache in this case.

### *4.10 SigCache vs. traditional signature files*

Traditional signature files have a relatively high update cost, but will be cheaper than using the SigCache approach if the update rate is low, and queries are mostly done on a constant set of objects (for example all objects of a certain class). However:

- In an ODB, queries are often performed on dynamic sets of objects. This can be collections like sets or bags.
- Due to inheritance, one object can implicitly be the member of more than one set (class), cf. Section 4.6.
- Accessing one object in a set often involves navigation to another object.

This makes traditional signature files less beneficial for many application areas.

### *4.11 Signature caching in parallel and distributed systems*

To provide the necessary computing power *and data bandwidth*, a parallel architecture can be necessary. One of the problems with a parallel page server ODB based on shared-nothing multicomputers is limited scalability, but the use of signature caching increases the scalability, because caching signatures on remote nodes can be regarded as "cheap replication". This technique should also be applicable in a distributed ODB, where the potential gain is even high because of the higher communication cost.

## 5 Gain from using signatures

In order to study the gain from using signatures, we have developed a cost model that models the object access costs. The cost model focuses on disk access costs, as this is the most significant cost factor. We do not consider the additional update cost caused by storing the signatures, because the size of a signature compared to an object is small (between 3% and 6% in the study in this paper). The purpose is to analyze the effect of using signatures in an ODB, so we focus on navigational accesses and restrict this analysis to PMAs and signatures generated by the use of the superimposed coding technique.

### *5.1 Workload model*

In this analysis, we consider a database in a stable condition, with a total of $N_{obj}$ objects. For the workload, we consider two main cases:

*Case I:* This is the workload of a traditional application. The object size is relatively small, and the signatures are generated from a small number of attributes. In such applications, there will be a mix of access types, and only some of them can benefit from using signatures.

*Case II:* This is the workload of one of the emerging application areas, for example XML storage. As described by DeWitt et al. [9], the space overhead would be very high if each XML element was stored as a separate object. Instead, one object is used to store one XML document, and the XML elements stored as "light-weight objects" inside one storage object. The result is larger objects than case I, and each object contains a higher number of attributes. The attributes are frequently text strings, and a large fraction of the queries in such systems will be for perfect match of one or more words. In order to be able to use signatures for such queries, the object signatures are generated by superimposing the individual text word signatures. As a result, the value of $D$ will be large.

The default parameter values for the two cases are given in Table 1. Note that with the default parameters, we keep the total database size constant, the studied database has a size of 2 GB for both case I and II. This is not a large database, but the interesting

| Parameter | | Case I | Case II |
|---|---|---|---|
| Object size | $S_{obj}$ | 128 bytes | 2048 bytes |
| Number of objects | $N_{obj}$ | 16 mill. | 1 mill. |
| Attributes | $D$ | 4 attributes | 128 attributes |
| Fraction of perfect match accesses that are actual drops (selectivity) | $P_A$ | 0.001 | 0.001 |
| Fraction of the read accesses that can benefit from signatures | $P_{PMA}$ | 0.4 | 0.8 |
| Signature size | $F$ | 32 bits | 1024 bits |

**Table 1.** Default workload parameters.

point is performance versus the relative memory size (memory size relative to database size). The results scale for larger database sizes, so that given a memory size of a fraction $f$ of the total database size, the gain from using signatures is the same *independent of the database size*. In all figures, the memory size is given as memory size relative to the database size, i.e. as $\frac{M}{N_{obj}S_{obj}}$ (note that when signatures are used, the memory needed to keep all pages in memory is larger).

### *5.2 Performance*

Fig. 2 shows the gain from using signatures, with 4 different access patterns, 70/30, 80/20, 90/10, and 95/05 (where 70/30 means that 70% of accesses are done to 30% of the objects, etc.). The performance gain (for example, the increase in transactions per time unit) from using signatures is calculated as

$$Gain = 100\left(\frac{T^{nosig}_{readobj} - T^{sig}_{readobj}}{T^{sig}_{readobj}}\right)$$

where $T^{nosig}_{readobj}$ is the average object access objects when not using a SigCache, and $T^{sig}_{readobj}$ is the average object access objects when a SigCache is employed.

*Case I:* We see that using signatures is especially beneficial for access patterns with a narrow hot spot area. There are two cases when signatures are not beneficial:

1. When the memory size is large enough to keep most of the hot spot object pages. In this case, it is more beneficial to use all the memory for page buffering, so that the hot spot objects can be kept in main memory. It should be noted that when the memory size is smaller, so that only some of these pages fits in the page buffer, using signatures is beneficial.
2. The number of object pages necessary to store a database will be larger if signatures are stored as well. If the main memory is large enough to keep most of the object pages in main memory (large values of $M$) when signatures are not used, using signatures will result in decreased or negative gain because of a lower page buffer hit rate.

*Case II:* In this case, using signatures is very beneficial for all access patterns, except when most of the object pages fits in main memory, as explained above. The gain is high, up to 300% when we have a narrow hot spot area and medium amounts of memory.

A more detailed analysis of the performance and gain from using the SigCache approach is given in [7].

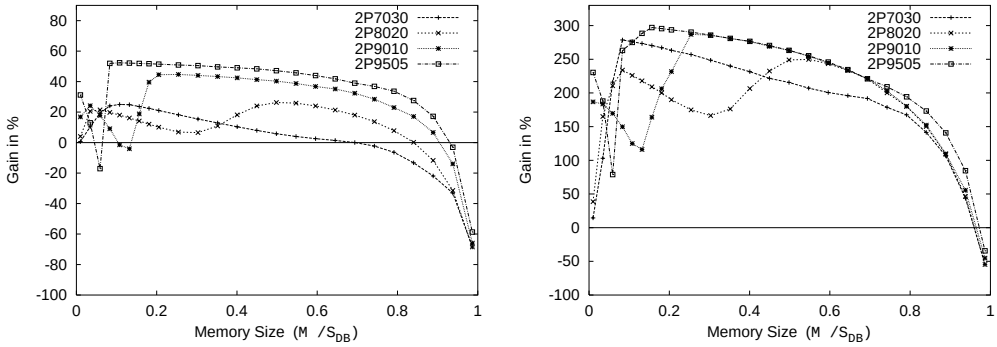


**Fig. 2.** Gain from using signatures with different access patterns. Case I to the left, and case II to the right.

## 6 Conclusions

In this paper we have described how object signatures can be cached in main memory in a signature cache, and how the use of the signatures in perfect match object accesses can be used to reduce the average object access cost in an ODB.

When storing signatures together with their objects instead of in separate signature files, the signature maintenance cost in terms of disk space as well as I/O is only marginal. This makes the SigCache technique an interesting supplement to traditional signature files as well as traditional indexes, which have a higher maintenance cost, and in the case of indexes, a higher storage cost as well.

The discussion here is done in the context of ODBs, but the SigCache approach is also applicable for relational and object-relational database systems experiencing a navigational access pattern. It can also be employed in parallel and distributed object database systems, in order to reduce communication costs.

## References

1. V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML documents in Xyleme. Technical Report 182, Verso/INRIA, 2000.
2. E. Bertino and F. Marinaro. An evaluation of text access methods. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, 1989. Vol.II: Software Track*, 1989.
3. C. Faloutsos. Access methods for text. *ACM Computer Surveys*, 17(1), 1985.
4. C. Faloutsos and R. Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *Proceedings of the 14th VLDB Conference*, 1988.
5. Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in OODBs. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.
6. D. L. Lee, Y. M. Kim, and G. Patel. Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 7(3), 1995.
7. K. Nørvåg. Fine-granularity signature caching in object database systems. Technical Report IDI 6/2000, Norwegian University of Science and Technology, 2000.
8. H. Schöning and J. Wäsch. Tamino – an internet database system. In *Proceedings of EDBT'2000*, 2000.
9. F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies (submitted for publication), 2000.