Global Page Replacement in Spatial Databases

Apostolos Papadopoulos Yannis Manolopoulos

Department of Informatics, Aristotle University, Thessaloniki 54006, Greece
email : apapadop,manolopo@athena.auth.gr

Abstract. In this paper we show how we can utilize a database buffer, when the underlying dataspace is organized by means of a spatial data structure and particularly R-trees. In such a case the Least Recently Used (**LRU**) page replacement policy is not powerful enough to guarantee efficiency, because it does not take into consideration important parameters of the structure, such as the portion (area) of the dataspace that each page occupies. Of course, the time period passed since the last reference of a page is still an important factor. Therefore, we combine several factors, in order to derive a more powerful caching algorithm. Experiments based on real and synthetic datasets, show that the new replacement algorithm performs better than **LRU** in many cases and also supports tuning.

1 Introduction

Spatial data management is an active area of research over the past ten years [Guti94, Laur92, Same90]. The basic direction we could follow in order to improve the efficiency of a Spatial DBMS, involves the design of robust spatial data structures and algorithms, aiming at efficiently satisfying user requests (inserts, deletes, updates and queries). Among these structures we highlight: the quadtree family [Same90], the Cell-tree [Gunt89], the LSD-tree [Henr89], the R-tree [Gutt84], packed R-trees [Rous85, Kame93], R^+-tree [Sell87], the R^*-tree [Beck90], and the Hilbert R-tree [Kame94].

Another direction involves the improvement of query processing with additional helpful techniques. One such technique is buffering. The use of buffers is obligatory in a DBMS [Effe84, O'Neil93], since the performance improvements can be substantial, if we keep resident frequently requested pages.

There are two basic factors that affect the performance of a buffer cache. First, the allocation of buffer pages to individual transactions must guarantee reasonable response times and second, the page replacement policy must be efficient and effective. In this paper we focus on the second factor only. One page replacement algorithm that is widely acceptable is the **LRU** (Least Recently Used) policy. If there is no empty buffer slot and a new page must be retrieved, a page already into the buffer must be replaced (victim page). The **LRU** policy replaces the page that has not been referenced for the longest time period.

Although the **LRU** policy has been proven a very successful method for cache manipulation [Effe84], modifications such as the **LRU-K** policy [O'Neil93] and

2Q policy [John94] perform better in a database environment. In this paper we focus on spatial database systems that organize the dataspace by means of a spatial data structure (i.e. R-trees). We show that taking into consideration criteria such as the area occupied by each page, a more sophisticated and tunable cache maintenance algorithm can be derived.

A new page replacement algorithm, **LRD-Manhattan**, is proposed. Each page into the buffer is considered as a point in the 2-dimensional Euclidean space. The first dimension is the reference density of the page [Effe84] and the second is the area of the page's MBR. When a victim must be chosen, we choose the page that has the minimum distance from the coordinate's system origin. This page is selected by means of the Manhattan (city block) metric. Experiments based on real and synthetic datasets, show that this can improve the cache utilization.

We consider the original R-tree [Gutt84] as the underlying data structure. However, the method is applicable to any R-tree variant or any other spatial data structure. Also, although the discussion is based on the 2-d space, the generalization to spaces with higher dimensionality is straightforward.

The rest of the paper is organized as follows. In Section 2 we give the appropriate background on R-trees and distance metrics. In Section 3 we present the motivation behind the proposed method and we describe the method in detail. Section 4 contains the experimental results and performance comparisons. Finally, in Section 5 we conclude the paper and give some directions for future research in the area.

2 Background

2.1 R-trees

The R-tree [Gutt84] is a hierarchical, height balanced data structure (all leaf nodes appear at the same level), designed for use in secondary storage, and it is a generalization of the B-tree for multidimensional spaces. For clarity and simplicity purposes we focus on 2-d space. A sample dataspace with a corresponding R-tree is presented in Figure 1 below. The structure handles objects by means of

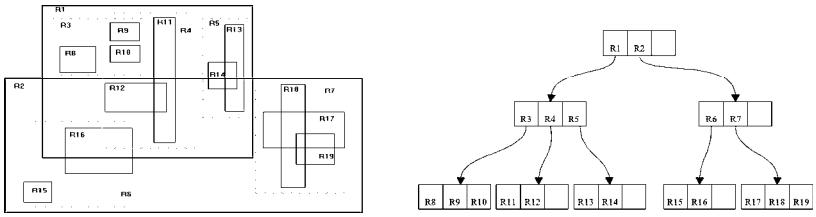

Fig. 1. R-tree example.

their conservative approximation. The most simple conservative approximation of an object's shape is the Minimum Bounding Rectangle (MBR). Each node of the tree corresponds to exactly one disk page. Internal nodes contain entries of the form *(R,child-ptr)*, where R is the MBR that encloses all the MBRs of its

descendants and *child-ptr* is the pointer to the specific child node. Leaf nodes contain entries of the form *(R,object-ptr)* where R is the MBR of the object and *object-ptr* is the pointer to the objects detailed description. Since MBRs of internal nodes are allowed to overlap, we may have to follow multiple paths from root to leaves when answering a range query. This inefficiency triggered the design of the R^+-tree [Sell87] which does not permit overlapping MBRs of internal nodes and performs better especially for small range queries.

One of the most important factors that affects the overall structure performance is the node split strategy used. In [Gutt84] three split policies are reported, namely exponential, quadratic and linear split policies. However, more sophisticated policies that reduce the overlap of MBRs are reported in [Beck90] and in [Kame94] that lead to R^*-trees and Hilbert R-trees respectively. These two structures are the most robust and efficient, to the best of the authors' knowledge. In this paper, we base our work on the original R-tree structure [Gutt84] for simplicity and because we want to give emphasis not on the underlying spatial data structure but on the technique to reduce the processing cost.

Finally, we note that some R-tree variants have been reported to support a static or a nearly static database. If the objects composing the dataspace are known in advance, we can apply several packing techniques, [Rous85, Kame93] with respect to the spatial proximity of the objects, in order to design a more efficient data structure.

2.2 Distance Metrics

Assume that the dataspace is the unit square and that we have two points p and q with coordinates (p_x, p_y) and (q_x, q_y) respectively. In many applications (GIS, LIS, etc) it is required to have a measure such as to identify how far a point is located with respect to a reference point. The most common metric is the L_k metric, which is defined as:

$$L_k(p,q) = \sqrt[k]{|q_x - p_x|^k + |q_y - p_y|^k}$$

Setting $k = 2$ we get the Euclidean metric L_2 and setting $k = 1$ we get the Manhattan (city block) metric L_1.

The applicability of a distance metric depends heavily on the kind of application. For example, measuring distances in a city, the Manhattan metric is more appropriate, since in order to travel from one place to another, we use the paths between building walls. On the other hand, if we need to know the closest forest to a lake, the Euclidean distance is more appropriate. For the rest of the paper we consider only the Manhattan metric. We justify its usefulness in the next section, where we present the new global page replacement algorithm.

3 The LRD-Manhattan Replacement Policy

3.1 Motivation

Assume we have a set $\mathcal{O}$ of spatial objects in the 2d space, organized in a R-tree. Assume also that a set $\mathcal{Q}$ of queries must be processed. If the queries are executed sequentially, then the processing of each query q_i corresponds to a depth first search (DFS) traversal of a portion of the structure. In such a case we have a priori knowledge of the access patterns and we can take advantage of this. Indeed, in [Chan92] a hint passing algorithm is presented that utilizes the access pattern knowledge very well. The buffer pages are labeled as useful and useless with respect to the time of the next reference. However, in multi-user environments such a hint passing algorithm can not be applied. The reason is that when many users are querying the database concurrently, such a characterization of the buffer pages is difficult. Moreover, a useful page for a query q_i may be useless for another query q_j and therefore, the pages present in the buffer may not follow a DFS traversal of the R-tree. Also, during the processing of other complex queries, such as spatial join queries, the DFS approach is not followed. Instead, the addresses of the related pages are first obtained, the join graph is constructed and finally the access path is derived. What we need is more general criteria for characterizing a page useful or useless and more sophisticated "hints", in order to eject a page from the buffer pool.

3.2 The Algorithm

The basic idea behind the proposed replacement algorithm is to select as the victim page, the page that has the minimum probability of reference. We assign to each buffer resident page p_i a reference probability $Prob_i$ that serves as a priority measure. When the buffer is full, the page with the minimum value of $Prob_i$ is ejected from the buffer. The $Prob_i$ probabilities are derived by combining two different values:

- the probability of reference due to buffer usage PRB_i, and
- the probability of reference due to data structure characteristics, $PRDS_i$.

The value PRB_i can be derived using the concept of reference density of a page p_i [Effe84]. Assume that a page p_i is just brought into the buffer. We store the reference number for the page entrance, $First\_Reference_i$, and the number of references, $Total\_References_i$, of the page. Let $Current\_Reference$ denotes the current reference number. This number is increased with every page reference. The reference density RD_i for the page p_i is calculated as follows:

$$RD_i = \frac{Total\_References_i}{Current\_Reference - First\_Reference_i}$$

Clearly $0 \leq RD_i \leq 1$ and we can interpreted RD_i as the reference probability for page p_i. In fact, the page replacement algorithm **LRD** (Least Reference

Density) [Effe84] ejects from the buffer the page with the minimum value of RD_i. Therefore, setting $PRB_i = LRD_i$ we have defined the probability of reference due to buffer usage.

Let us know define the probability of reference due to data structure characteristics, $PRDS_i$. Each R-tree page (node) can be characterized by the area that occupies due to its MBR. The area of the MBR is highly related to the probability that this page will be fetched during the processing of a range query [Kame93, Page93]. Assume that range queries are generated uniformly over the dataspace, with respect to the query window centroid. For point queries (degenerated range queries) the probability that an R-tree page will be fetched is simply the area of the page MBR (the dataspace is assumed to be the unit square). For general range queries, the probability that a page will be fetched is the area of the page MBR augmented by the x and y extends of the query window [Kame93, Page93]. However, the extends of range queries posed by users are arbitrary in general. Therefore, we assume that the fetching probability is given by just the area of the page (ignoring the extends of range queries). Thus, we can define the reference probability of a page p_i due to the data structures characteristics as $PRDS_i = area_i$, where $area_i$ is the area of the page's MBR.

Each buffer page p_i is assigned the two values PRB_i and $PRDS_i$ where clearly $0 \leq PRB_i \leq 1$ and $0 \leq PRDS_i \leq 1$. The next step involves the assignment of a value to every $Prob_i$, given the values PRB_i and $PRDS_i$. We can use the following function:

$$Prob_i = \frac{PRB_i + PRDS_i}{2}, \qquad 0 \leq Prob_i \leq 1$$

Assume now, that a page fault has occurred and a new page must be retrieved into the buffer. If the buffer is full, a victim must be chosen. We calculate the $Prob_i$ values for every buffer page p_i and we select to replace the page with the minimum value of $Prob_i$. Each page p_i can be viewed as a point in the 2-d space (unit square) with coordinates PRB_i (x-axis coordinate) and $PRDS_i$ (y-axis coordinate). Clearly, the selected victim page will have the smaller value for $PRB_i + PRDS_i$ and therefore, the smaller Manhattan distance with respect to the origin of the coordinate system. In Figure 2, we present an example. Also,

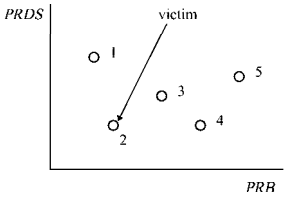

Fig. 2. Example of selecting a victim page from a 5 page buffer.

with respect to the least recently used criterion, we apply the above calculations only to a portion of the buffer pages. Therefore, we consider the F least recently used buffer pages (F is a parameter) and among them we select the victim according to the minimum value of $Prob_i$. The whole algorithm is described below:

Algorithm **LRD-Manhattan**
begin
  if (a page fault occurs) **and** (buffer full)
  then
    isolate the F least recently used pages.
    for each page p_i calculate PRB_i and $PRDS_i$.
    set $Prob_i = \frac{PRB_i + PRDS_i}{2}$.
    from the F pages select p_{victim} with the minimum value of $Prob_i$.
    read the new page p_{new} into the buffer.
  endif
end

We note that the value $PRDS_i$ for each page p_i, needs only to be calculated once, when the page is fetched into the buffer. On the other hand the value PRB_i must be calculated every time a victim is selected.

4 Experimental Results

4.1 Preliminaries

We implemented the R-tree data structure with the quadratic split policy, and the page replacement policies **LRU**, **LRD** and **LRD-Manhattan**, in the C programming language under UNIX. The simulation experiments were performed on DEC 3000 and SUN SPARC workstations. The page size was set to 2048 bytes (2Kbytes). The dataspace dimensions were set to the unit square $[0,1) \times [0,1)$ and all benchmark datasets were normalized to fall inside the dataspace area. The different datasets used throughout the evaluation of the methods are presented in Figure 3. The first dataset contains real-life objects describing the roads of Long Beach (TIGER project) as MBRs. The second dataset is synthetic and contains uniformly distributed rectangles in the 2-d space. The maximum x and y extends of the objects were set to 0.01. In order to test the new algorithm under

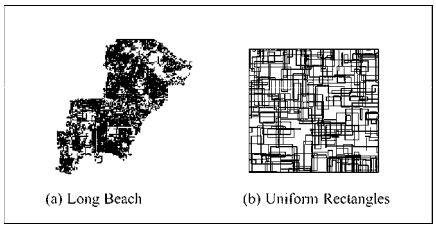

Fig. 3. Data sets used throughout the experimentation.

different access patterns, we used two query sets. The first set contains uniform generated range queries with respect to the query window centroid, whereas the second contains zipfian (80-20) range queries with respect to the query window

centroid. The zipfian queries were produced by means of the following formula [Gray94], applied to both the x and y coordinates of the query window centroid:

$$skew\_value = E \cdot R^{\frac{ln(h)}{ln(1-h)}}$$

where R is a random number between 0 and 1, E is the dataspace extend (x or y accordingly) and h is the skewness factor. In order to produce a 80-20 zipfian query distribution, h was set to 0.8. We used small range queries with maximum window 0.01×0.01, and large range queries with maximum window 0.1×0.1.

A very important factor that affects the performance of an R-tree index is the height of the tree. Query processing performance is very sensitive to the height of the tree, since it is related to the number of disk accesses. It is reported in the literature [Lin94] that R-trees performance deteriorates for dimensionalities over 20. In the experiments performed we used 2Kbytes pages. Assuming that each number or pointer occupies 4 bytes of storage, for 2-d space we can achieve a maximum fanout of 100. On the other hand, for 20-d space the maximum fanout drops to 12. We used both values in order to observe the performance of the proposed algorithm. It is also assumed that there are 20 users posing queries concurrently. We simulated the concurrent processing of queries as follows:

- 20 reference strings (one for each user) were generated,
- the reference strings were merged into one, taking one page reference from each user in a round-robin manner.

4.2 Performance Comparisons

We use the number of page faults occurred during the processing of all queries, in order to compare the various replacement policies. The buffer initially was empty. In Figure 4 we present the relative performance of the policies with respect to the buffer size (in pages). Let PF_{LRU}, PF_{LRD} and $PF_{LRD-Manhattan}$ denote the number of page faults for each policy. The relative performance of **LRD** and **LRD-Manhattan** with respect to **LRU** is measured as follows:

$$RP_{LRD} = \frac{PF_{LRD}}{PF_{LRU}} \times 100, \qquad RP_{LRD-Manhattan} = \frac{PF_{LRD-Manhattan}}{PF_{LRU}} \times 100$$

Graphs (a), (b), (c) and (d) show the results when the dataspace is composed of the Long Beach dataset and when the distribution of the queries is uniform. Graphs (e) and (f) show the results when the data objects are uniformly distributed synthetic rectangles, whereas the queries are skewly (80-20) distributed over the dataspace. The size of the buffer ranges between 10 and 100 pages. The R-tree index maximum fanout was set to 12 (small) and 100 (large). We note that we performed our experiments setting the parameter F to $\frac{N}{3}$, where N is the number of buffer space in pages.

4.3 Interpretation of Results

By studying Figure 4, some very interesting observations can be derived. For uniform range queries the new algorithm outperforms both **LRU** and **LRD**. The performance improvements over the **LRU** reach 20%. However, in graph (b) we observe that **LRD-Manhattan** performs poorly when the buffer space ranges between 30 and 50 pages. In this range, **LRU** outperforms **LRD** and **LRD-Manhattan**. For buffer space grater than 60 pages, the proposed algorithm is consistently better. When the buffer space ranges between 30 and 50 pages, each query possesses about 2 pages (since there are 20 users executing queries concurrently). In a situation like this **LRU** performs better. We performed experiments setting the number of users to 10 and the same peak was observed, but the inefficiency range was between 10 and 30 pages (again about 2 pages per query). The fact is that 2 pages is a very limited buffer space, in order to begin the processing of a query. Moreover, the buffer allocator would never permit the processing of a query with such limited buffer space. Therefore, we believe that this partial inefficiency of the proposed algorithm would cause no serious performance deterioration in a real environment. In the case of zipfian distribution of the queries, **LRD** is the best choice since the access patterns are quite steady. The performance of **LRD-Manhattan** is very close to that of **LRU**. In conclusion, the experimental results show that the proposed algorithm is generally promising. However, more careful tuning is required, in order to achieve even better performance in all query distributions.

5 Conclusions

In this paper we studied the concept of page replacement in spatial data structures and particularly in R-trees. The buffering techniques are well studied in the literature and several approaches have been followed in order to increase the performance. A new global page replacement algorithm, **LRD-Manhattan**, is introduced and compared to **LRU** and **LRD**. Experimental results based on real-life and synthetic datasets show that the new algorithm outperforms **LRU** and **LRD** in many cases. However, when queries follow a highly skew distribution (zipfian 80-20) the best choice is **LRD**. Based on these results, we conclude that further research in the area is required in order to derive more powerful replacement techniques. Future research may include:

- Modification of the proposed algorithm in order to take into consideration other useful criteria such as the reference frequency of each buffer page. Therefore, each page could be considered as a point in n-d space, where n is the number of parameters used to describe a single page.
- Tuning the algorithm with respect to the parameter F (the number of least recently used buffer pages).
- Modification of the $Prob_i$ values in order to obtain a more general and tunable function. For example, we could assign $Prob_i = \frac{a*PRB_i+b*PRDS_i}{a+b}$ where a and b are tunable constants.
- Consideration of other spatial data structures.

– Modification of **LRU-K** and **2-Q** algorithms in order to obtain more powerful techniques suitable for spatial databases.

References

[Beck90] N. Beckmann, H.P. Kriegel and B. Seeger: "The R*-tree: an efficient and robust method for points and rectangles", *Proceedings of the 1990 ACM SIGMOD Conference*, pp.322-331, Atlantic City, NJ, 1990.

[Effe84] W. Effelsberg: "Principles of database buffer management", *ACM Transactions on Database Systems*, vol.9, no.4, pp.560-595, 1984.

[Gray94] J. Gray, S. Englert, K. Baclawski and P. Weinberger: "Quickly generating billion-record synthetic databases", *Proceedings of the 1994 ACM SIGMOD Conference*, pp.243-252, Minneapolis, MN, 1994.

[Gunt89] O. Gunther: "The design of the Cell tree: an object-oriented index structure for geometric databases", *Proceedings of the 5th IEEE Conference on Data Engineering*, pp.598-615, Los Angeles, CA, 1989.

[Guti94] R.H. Guting: "An introduction to spatial database systems", *The VLDB Journal*, vol.3, no.4, pp.357-399, 1994.

[Gutt84] A. Guttman: "R-trees: a dynamic index structure for spatial searching", *Proceedings of the 1984 ACM SIGMOD Conference*, pp.47-57, Boston, MA, 1984.

[Henr89] A. Henrich, H.W. Six and P. Widmayer: "The LSD-tree: spatial access to multidimensional point and non-point objects", *Proceedings of the 15th VLDB Conference*, pp.45-53, Amsterdam, Netherlands, 1989.

[John94] T. Johnson and D. Shasha: "2Q: a low overhead high performance buffer management replacement algorithm", *Proceedings of the 20th VLDB Conference*, pp.439-450, Santiago, Chile, 1994

[Kame93] I. Kamel and C. Faloutsos: "On packing R-trees", *Proceedings of the 2nd Conference on Information and Knowledge Management (CIKM)*, Washington DC, 1993.

[Kame94] I. Kamel and C. Faloutsos: "Hilbert R-tree: an improved R-tree using fractals", *Proceedings of the 20th VLDB Conference*, pp.500-509, Santiago, Chile, 1994.

[Laur92] R. Laurini and D. Thompson: "*Fundamentals of spatial information systems*", Academic Press, London, 1992.

[Lin94] K. I. Lin, H. V. Jagadish and C. Faloutsos: "The TV-tree: an index structure for high-dimensional data", *The VLDB Journal*, vol.3, pp.517-542, 1994.

[O'neil93] E. J. O'Neil, P. E. O'Neil and Gerhard Weikum: "The LRU-K page replacement algorithm for database disk buffering", *Proceedings of the 1993 ACM SIGMOD Conference*, pp.297-306, Washington DC, 1993.

[Page93] B.U. Pagel, H.W. Six, H. Toben and P. Widmayer: "Towards an analysis of range query performance in spatial data structures", *Proceedings of the 1993 ACM PODS Conference*, pp.214-221, Washington DC, 1993.

[Rous85] N. Roussopoulos and D. Leifker: "Direct spatial search on pictorial databases using packed R-trees", *Proceedings of the 1985 ACM SIGMOD Conference*, pp.17-31, Austin, TX, 1985.

[Same90] H. Samet: "*The design and analysis of spatial data structures*", Addison-Wesley, Reading, MA, 1990.

[Sell87] T. Sellis, N. Roussopoulos and C. Faloutsos: "The R+-tree: a dynamic index for multidimensional objects", *Proceedings of the 13th VLDB Conference*, pp.507-518, Brighton, UK, 1987.

This article was processed using the LaTeX macro package with LLNCS style

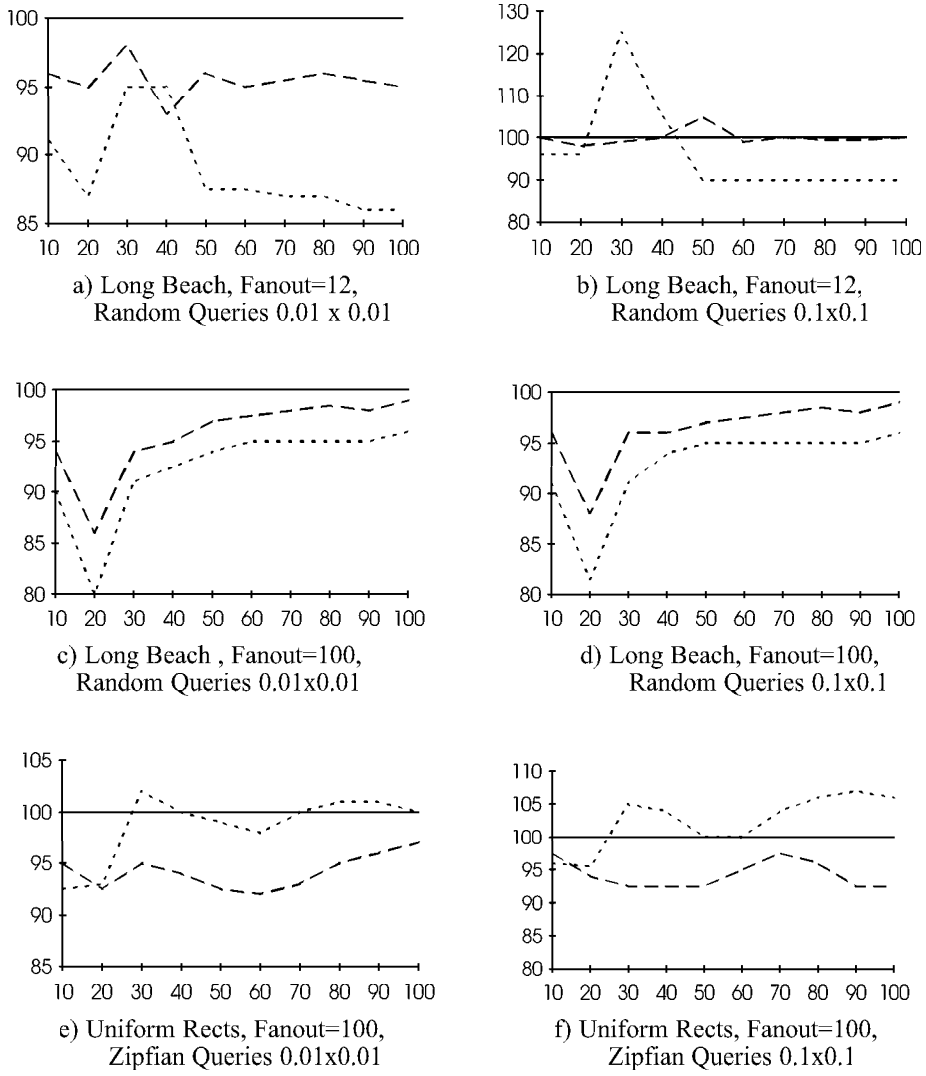

Fig. 4. Experimental results.