The Impact of Buffering on Closest Pairs Queries Using R-Trees

Antonio Corral[1], Michael Vassilakopoulos[2], and Yannis Manolopoulos[3]⋆

[1] Department of Languages and Computation
University of Almeria, 04120 Almeria, Spain
`acorral@ual.es`

[2] Lab of Data Engineering, Department of Informatics
Aristotle University, 54006 Thessaloniki, Greece
`mvass@computer.org`

[3] Department of Computer Science
University of Cyprus, 1678 Nicosia, Cyprus
`manolopo@ucy.ac.cy`

Abstract. In this paper, the most appropriate buffer structure, page replacement policy and buffering scheme for closest pairs queries, where both spatial datasets are stored in R-trees, are investigated. Three buffer structures (i.e. single, hybrid and by levels) over two buffering schemes (i.e. local to each R-tree, and global to the query) using several page replacement algorithms (e.g. FIFO, LRU, 2Q, etc.) are studied. In order to answer K closest pair queries (K-CPQs, with $K \geq 1$) we employ recursive and non-recursive (iterative) branch-and-bound algorithms. The outcome of this study is the derivation of the outperforming configuration (in terms of buffer structure, page replacement algorithm and buffering scheme) for CPQs. In all cases, the savings in disk accesses is larger for a recursive algorithm than for a non-recursive one, in the presence of buffer space. Also, the global buffering scheme is more appropriate for small or medium buffer sizes for recursive algorithms, whereas the local scheme is the best choice for large buffers. If we use non-recursive algorithms, the global buffering scheme is the best choice in all cases. Moreover, LRU is the most appropriate page replacement algorithm for small or medium buffer sizes for both types of branch-and-bound algorithms. FIFO and LRU are the best choices for recursive algorithms and 2Q for the non-recursive ones, when the buffer is large enough.

1 Introduction

The use of buffers is very important in DBMSs, since it can improve the performance substantially (reading data from the disk is significantly more expensive than reading from a main memory buffer). There exist two basic research directions that aim at reducing the disk I/O activity and enhancing the system

⋆ On sabbatical leave from the Department of Informatics, Aristotle University, 54006 Thessaloniki, Greece. `manolopo@csd.auth.gr` throughput during query processing using buffers. The first one focuses on the availability of buffer pages at runtime by adapting memory management techniques for buffer managers used in operating systems to database systems [1,9, 13,15]. The second one focuses on query access patterns, where the query optimizer dictates the query execution plan to the buffer manager, so that the latter can allocate and manage its buffer accordingly [4,6,20].

The spatial selections, nearest neighbor searches and joins are considered the most important queries in spatial databases that are based on R-trees. R-trees [10] are multi-dimensional, height balanced tree structures for secondary storage, that handle objects by means of their Minimum Bounding Rectangles (MBRs). In [5] a new kind of spatial query, called K closest pairs query (K-CPQ), is presented. It combines join and nearest neighbor queries for discovering the K pairs ($K \geq 1$) of spatial objects from two datasets that have the K smallest distances between them (1-CPQ is treated as special case). Like a join query, all pairs of objects are candidates for the result. Like a nearest neighbor query, proximity metrics are the basis for pruning strategies and the final ordering.

The main objective of this work is to find the most appropriate buffer structure, page replacement policy and buffering scheme for CPQs, where both spatial datasets are indexed with R-trees. Based on experimental results, we draw conclusions about the importance of using an appropriate buffer management for the I/O performance of this kind of query. We present a comparative study, where several parameters (such as the buffer structure, page replacement algorithms, buffering schemes, buffer size in pages, number of pairs in the result K and the nature of indexed datasets) and corresponding values are considered.

The rest of this paper is organized as follows. In Sect- 2, we review the literature (CPQs using R-trees and buffering) and motivate the research topic under consideration. In Sect. 3, a brief description of the spatial access method (i.e. R-tree) and the branch-and-bound algorithms (i.e. recursive and non-recursive) for satisfying CPQs are presented. In Sect. 4, in order to study the effect of buffering in the performance of this kind of query, we examine combinations of buffer structures, page replacement algorithms and buffering schemes. Moreover, in Sect. 5, an extensive comparative performance study of CPQ algorithms over these alternative combinations is presented. Finally, in the last section, the conclusions on the contribution of this paper and future research plans are summarized.

2 Related Work and Motivation

In DBMSs, the buffer manager is responsible for operations in the buffer pool, including buffer space assignment to queries, replacement decisions and buffer reads and writes in the event of page faults. When buffer space is available, the manager decides about the number of pages that are allocated to an activated query. This decision may depend on the availability of pages at runtime (page replacement algorithms), or the access pattern of queries (nature of the query). A number of studies focus on adapting memory management techniques used in operating systems to database systems, such as FIFO, LRU, LFU, Gclock, etc. [1,9,13,15]. Other research efforts aim at determining the buffer requirements of queries based on their access patterns (the nature of the query) without considering the availability of buffer pages at runtime [4,6,20].

Since this paper is related to the research directions based on the nature of the query, we focus in the most representative papers about the buffer management on indices. In [18], an LRU buffer structure for indices was presented (OLRU), where the addressing space is logically partitioned into L independent regions, each managed by a local LRU chain. In [6] an extensible and dynamic priority-based hint mechanism was proposed to design an optimal replacement strategy by exploiting the predictable access pattern of indexing methods. An application on their hint mechanism was to design a hybrid replacement strategy, combining the LRU and MRU page replacement policies. There are several studies on spatial queries involving more than one R-tree, and most of them examine the use of buffering to reduce the I/O activity [3,5,7,11,12,17].

All the previous papers involved more than one R-tree for the query and used a buffer pool with LRU or FIFO replacement policy, but they did not justify the use of these policies. In other words, they did not examine several alternatives for the buffer structure, or for the page replacement strategies in order to reduce the disk activity. In this paper, our objective is to find the most appropriate buffer pool structure (i.e. single, hybrid and by levels) over two buffering schemes (i.e. local and global) and the best page replacement policy (e.g. FIFO, LRU, Gclock, etc.) for CPQs, where both spatial datasets are indexed by R-trees.

3 R-Trees and Algorithms for Closest Pairs Queries

3.1 R-Trees

R-trees [10] are hierarchical, height balanced data structures based on B$^+$-trees, used for the dynamic organization of k-dimensional geometric objects that are represented by k-dimensional MBRs. R-trees obey the following rules. Leaves reside on the same level and contain pairs of the form (R, O), where R is the MBR containing the object determined by the identifier O, spatially. Internal nodes contain pairs of the form (R, P), where P is a pointer to a child of the node and R is the MBR containing (spatially) the rectangles stored in this child. Also, internal nodes correspond to MBRs containing (spatially) the MBR of their children. An R-tree of class (m, M) has the characteristic that every node, except possibly for the root, contains between m and M pairs, where $m \leq \lceil M/2 \rceil$. If the root is not a leaf, it contains at least two pairs. Figure 1 depicts some rectangles on the right and the corresponding R-tree on the left. Dotted lines denote the bounding rectangles of the subtrees that are rooted in inner nodes.

Many R-tree variants have appeared in the literature. One of the most popular variations is the R*-tree [2], which follows a sophisticated node split technique and is considered to be the most efficient variant of the R-tree family. In this paper, we have chosen R*-trees to perform our experimental study, although in the sequel, the terms R-tree and R*-tree will be used interchangeably.

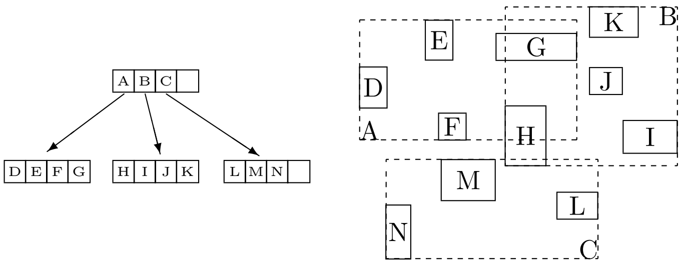


Fig. 1. An example of an R-tree

3.2 Algorithms for Closest Pairs Queries

A new spatial query was presented in [5], called K closest pairs query (K-CPQ). It combines join and nearest neighbor queries for discovering the K pairs ($K \geq 1$) of spatial objects from two datasets that have the K smallest distances between them. These queries are defines as follows.

1-CPQ. Assume two object datasets P and Q (where $P \neq \emptyset, Q \neq \emptyset$), stored in two R-trees, R_P and R_Q, respectively. Find the pair of objects p, $p \in P \times Q$, such that: $dist(p) \leq dist(p'), \forall p' \in (P \times Q - \{p\})$, where $dist$ is a Minkowski distance of the pairs of $P \times Q$.

***K*-CPQ.** Assume two object datasets P and Q (where $P \neq \emptyset$, $Q \neq \emptyset$), stored in two R-trees, R_P and R_Q, respectively. Find the K ordered pairs of objects $p_1, p_2, \ldots, p_K, p_i \in P \times Q$, such that: $dist(p_1) \leq dist(p_2) \leq \ldots \leq dist(p_K) \leq dist(p'), \forall p' \in (P \times Q - \{p_1, p_2, \ldots, p_K\})$.

Metrics (MINIMINDIST, MINMAXDIST and MAXMAXDIST) and properties between two MBRs in the k-dimensional Euclidean space were proposed for the 1-CPQ and K-CPQ in [5] as bounds for the branch-and-bound (recursive and non-recursive) algorithms. The recursive branch-and-bound algorithm (with a synchronous traversal, following a depth-first search strategy) for processing the 1-CPQ between two sets of points stored in two R-trees with the same height can be described by the following steps:

CPQ1. Start from the roots of the two R-trees and set the minimum distance found so far, T, to ∞.

CPQ2. If you access a pair of internal nodes, then calculate the minimum of MINMAXDIST for all possible pairs of MBRs. If this minimum is smaller than T, then update T. Calculate MINMINDIST for each possible pair of MBRs. Propagate downwards recursively only for those pairs having MINMINDIST$\leq T$.

CPQ3. If you access two leaves, then calculate the distance of each possible pair of points. If this distance is smaller than T, then update T.

The non-recursive branch-and-bound algorithm (with a synchronous traversal, following a best-first search strategy using a minimum heap) for processing the 1-CPQ between two sets of points stored in two R-trees with the same height can be described by the following steps:

CPQ1. Start from the roots of the two R-trees, set T to ∞ and initialize the minimum heap.

CPQ2. If you access a pair of internal nodes, then calculate the minimum of MINMAXDIST for all possible pairs of MBRs. If this minimum is smaller than T, then update T. Calculate MINMINDIST for each possible pair of MBRs. Insert into the minimum heap those pairs having MINMINDIST$\leq T$.

CPQ3. If you access two leaves, then calculate the distance of each possible pair of points. If this distance is smaller that T, then update T.

CPQ4. If the minimum heap is empty, then stop.

CPQ5. Get the pair on top of the minimum heap. If this pair has MINMINDIST$>T$, then stop. Else, repeat the algorithm from CPQ2 for this pair.

The pseudo-code of the recursive and non-recursive algorithms can be found in the technical report [8]. Moreover, in order to process the K-CPQ, an extra structure that holds the K closest pairs is necessary. More details can be found in [5].

4 Buffer Management

DBMSs use indices to speed up query processing (e.g. various spatial databases use R-trees). Indices may partly reside in main memory buffers. This reduces response times. The buffering effect should be studied, since even a small number of buffer pages can substantially improve the global database performance. Our objective is to find the best structure of the buffer pool, the best page replacement algorithm and the best buffering scheme for the buffer manager in order to reduce the number of disk accesses for K-CPQs. We propose three structures of the buffer pool (i.e. single, hybrid and by levels) managed by a variety of page replacement algorithms (e.g. FIFO, LRU, etc.).

The buffer pool structure will be organized adopting two buffering schemes as depicted in Fig. 2. In the first scheme, the buffer pool is split in two parts, each one allocated locally to an R-tree (left part of Fig. 2). We call it, thus, a Local buffering scheme. In the second one, the buffer pool is allocated globally to the query (right part of Fig. 2), giving rise to a Global buffering scheme.

In [9] a systematic description of replacement algorithms was presented for a single buffer structure. The FIFO (First-In First-Out) algorithm replaces the oldest page, even if its reference frequency gives the priority to the youngest page. The LFU schema (Least Frequently Used) replaces the page with the lowest reference frequency. Gclock consists of a circular decrementing of the reference counters until 0 is reached. When a buffer fault occurs, the first page having a counter equal to 0 is replaced. The LRU (Least Recently Used) algorithm gives the priority to the most recently used page, replacing the page that was the least recently used. MRU (Most Recently Used) is the opposite of LRU and replaces the page that was the most recently used. The LRU/2 is a particular case of LRU/K, proposed in [15] for $K = 2$, replacing the page whose penultimate (second-to-last) access is the least recent among all penultimate accesses. LRD

(Least Reference Density) is not a page replacement algorithm based on page ages, but on its reference density (reference probability) from the first time that a page was accessed. The page replacement algorithm LRD rejects from the buffer the page with the minimum reference density. Finally, in [16] a page replacement algorithm for spatial databases, called LRD-Manhattan, was proposed as a variation of LRD.

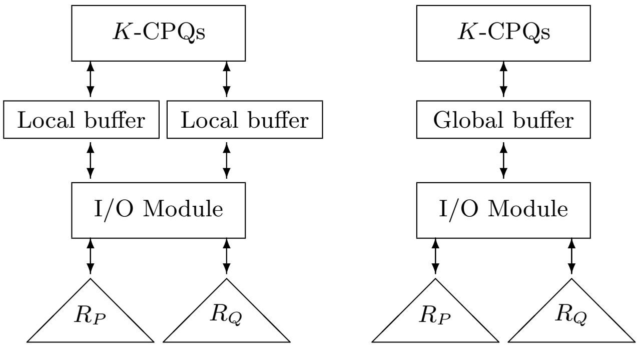


Fig. 2. Local and Global buffering schemes

The most representative methods for the hybrid buffer structure are the techniques called 2Q and FIFO_LRU. The 2Q algorithm divides the buffer pool in two areas: the hot area managed as an LRU queue and the cold area maintained as a FIFO queue [13]. On the first reference of a page, 2Q places it in the cold area (FIFO). If the page is re-referenced while in the cold area, then it is moved to the hot area (LRU). Evidently, if a page is not re-referenced while in the cold area, it is rejected from the buffer. In order to solve the "correlated references" problem, 2Q divides the cold area in two parts, one for pages and another for page identifiers. The FIFO_LRU technique works in the same way as 2Q, but the hot area is implemented as a FIFO replacement algorithm and the cold area is managed with an LRU policy [1].

Here, we present a buffer structure linked to each R-tree based on its height, h, for solving K-CPQs. This means that the buffer pool is split in h independent areas. For each R-tree level we allocate a number of pages according to its minimum fan-out factor m and its height, with the exception of the root, for which we allocate only one page. We create this buffer structure in a bottom-up way, trying to set a distribution of pages per level as fair as possible (root level=level $h-1 : m^0$, level $h-2 : m^1$, level $h-3 : m^2$, ..., level $1 : m^{h-2}$, leaf level=level $0 : m^{h-1}$). In the case of K-CPQs, pages at lower levels are very important for the branch-and-bound algorithms. Besides, we manage these h independent areas using a specific page replacement algorithm, for example LRU (LRU_L=LRU by levels), or FIFO (FIFO_L=FIFO by levels).

5 Experimentation

This section summarizes the results of an extensive experimentation that aims at measuring and evaluating the behavior of the recursive and non-recursive branch-and-bound algorithms for K-CPQs using different structures, schemes, policies and buffer sizes. We ignore the effect of path-buffer [5], since it offers more advantages to the recursive algorithms, regardless of the page replacement policy.

For our experiments, we have built several R*-trees [2] using the following datasets: (a) a real dataset from the Sequoia project [19] consisting of 62.536 points that represent specific country sites of California (Real), (b) a point dataset produced from the real one by moving randomly every point (Real$'$) and (c) two datasets of cardinality 62.536 points, which completely overlap and follow uniform and skewed distributions [5]. All experiments have run on a Linux workstation with 128 Mb of main memory and several Gb of secondary storage, making use of GNU C++ compiler. The page size was 1 Kb, resulting to a maximum R*-tree node capacity $M = 21$ (minimum capacity was set to $m = M/3 = 7$, a reasonable choice according to [2]). The quantity counted in all experiments was the number of disk accesses required to perform the K-CPQs.

5.1 K-CPQ Algorithms Using a Local Buffering Scheme

We now proceed to the performance comparison of the recursive and non-recursive branch-and-bound algorithms for K-CPQs using a Local buffering scheme in order to investigate the best page replacement policy and buffer structure. We used a buffer pool, B, with varying size from 0 to 512 pages, dedicating different portions of B to each R*-tree. The datasets joined were Real/Real$'$ and Uniform/Skewed. However, in the sequel we focus on Real/Real$'$ data sets, since both cases gave very similar trends.

First of all, for the hybrid structure in the Local or Global buffering scheme, we have performed several experiments with different B values ($B/2$ for each R*-tree) using recursive and non-recursive algorithms to derive the best page distribution for the hot and cold regions in the buffer. If B_P is the number of pages in the local buffer of the R*-tree R_P, the best configuration was <Hot, Cold> = $< B_P/2,\ B_P/2 >$. Moreover, for the Local buffering scheme, we have assigned a varying number of pages to each R*-tree, and the best distribution of the buffer was to assign more pages to the largest R*-tree, whatever the type (recursive, or non-recursive) of algorithm used. Since in our experimentation we have point datasets with identical cardinalities, $(B/2, B/2)$ was the best configuration [8].

We have run experiments using different page replacement policies over the three buffer pool structures. The best policies for the recursive algorithms were FIFO and LRU in case of small buffers (e.g. $B \leq 64$), but in case of large buffers (e.g. $B \geq 128$) LRU_L was slightly better than FIFO and LRU. FIFO and LRU were better than LFU, Gclock, MRU, LRU/2 and LRD, because recursion favors the youngest and most recently used pages in the backtracking phase and this behavior is slightly improved in case of large buffers organized by levels (FIFO_L and LRU_L). On the other hand, for the non-recursive algorithms and small buffers (e.g. $B \leq 64$), FIFO and LRU were again the best policies, whereas for large buffers (e.g. $B \geq 128$) 2Q was slightly better than FIFO and LRU. In this case, we did not use recursion and the organization of the buffer pool in two regions (i.e. hot and cold) provided a good performance, when the search strategy was best-first implemented through a heap of minimums and the buffer was large enough. For instance, for the recursive 1-CPQ, using a single buffer structure, MRU was 35% worse with respect to the LRU. Under these conditions, Gclock was 4% worse with respect to LRU, LFU 35% worse than FIFO, LRU/2 20% worse than LRU, and LRD 32% worse than FIFO. These behaviors are depicted in Fig. 3, where different page replacement policies are compared, using the recursive algorithm for 1-CPQ in a single buffer structure. Besides, if we include a large buffer (e.g. $B = 512$) with the single structure and the LRU policy, the savings in I/O operations were 73% for the recursive algorithm and 68% for the non-recursive one with respect to the absence of buffer space ($B = 0$). For the non-recursive algorithm the results were very similar.

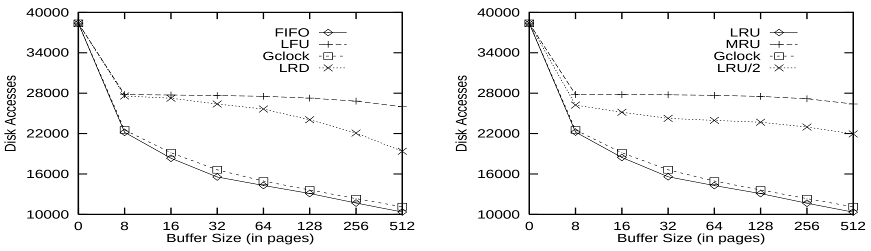
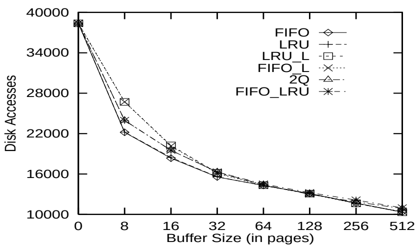


Fig. 3. The performance of the 1-CPQ recursive algorithm for various page replacement policies and a single buffer structure, as a function of the buffer size

For the recursive and non-recursive algorithms, in Fig. 4 we illustrate the performance of the 1-CPQ recursive (left) and non-recursive (right) algorithms for various page replacement policies, as a function of the buffer size. It can be seen that the two charts follow the same trend. When the buffer size is small (e.g. $B \leq 64$), the single structure with LRU policy is the best (with 6% and 5% savings for LRU in comparison with 2Q, for recursive and non-recursive algorithms, respectively), the second is the hybrid and the third one is by levels. However, in case of large buffers (e.g. $B \geq 128$) the difference is almost negligible for all page replacement policies, although LRU_L and 2Q are slightly better that the other for the recursive and non-recursive algorithms, respectively.

The results of the recursive K-CPQ algorithm for a given buffer size (e.g. $B = 512$) showed that the best behavior was for LRU_L with a 0.5% improvement over LRU (for all K values), whereas the worst results appeared in the case of the hybrid structure (2Q and FIFO_LRU). For the non-recursive algorithm with the same number of buffer pages ($B = 512$), the best behavior was for 2Q with a



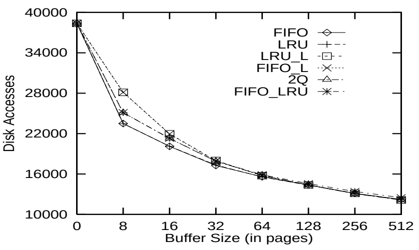
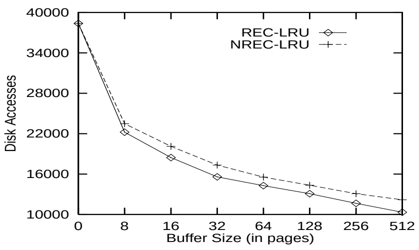


Fig. 4. The performance of the of 1-CPQ recursive (left) and non-recursive (right) algorithms for various page replacement policies, as a function of the buffer size

0.6% improvement over LRU (for all K values), whereas the worst results were for FIFO_LRU [8].

In the case of 1-CPQ, the recursive algorithm presents 10% excess of I/O activity in comparison to the non-recursive one with the same page replacement policy (LRU), as can be noticed by the gap between the two lines in the left part in the Fig. 5. The gap for K-CPQ is bigger when the K value is incremented; it is 25% bigger for $K \leq 10000$, but it reaches 45% when $K = 100000$ (see the right part of Fig. 5). Besides, by increasing K values (1..100000), the performance of the recursive algorithm is not significantly affected; with a buffer of 512 pages and the best page replacement algorithm there is an extra cost of 2%. On the other hand, this extra cost is about 39% for the non-recursive algorithm using the same buffer characteristics. If we do not have any buffer space ($B = 0$), then increasing K implies an additional cost of 33% for the recursive algorithm and 16% for the non-recursive one. Moreover, the recursive variant demonstrates savings in the range 73%-82%, when K increases (1..100000) and a buffer of 512 pages is used, in comparison to the no buffer case ($B = 0$). The non-recursive algorithm under the same buffer setup results in savings from 68% to 57%.



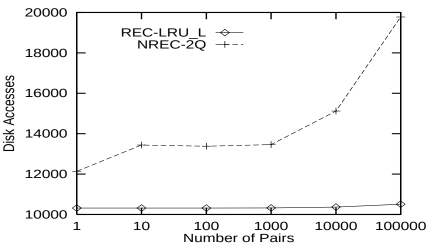


Fig. 5. The performance of the 1-CPQ (left) and the K-CPQ (right) recursive (REC) and non-recursive (NREC) algorithms using the best page replacement policies and $B = 512$, as a function of the buffer size

In Fig. 6, the percentage of I/O cost savings (induced by the use of buffer size $B > 0$ in contrast to not using any buffer) of the K-CPQ recursive algorithm with LRU_L policy (left) and non-recursive algorithm with 2Q policy (right) is depicted. For the recursive algorithm, the percentage of savings grows as buffer sizes increase, for all K values, although it is bigger for $K = 100000$. The behavior of non-recursive algorithm is slightly different. When the buffer becomes larger, the percentage of savings also increases, but when we fix the buffer size, the increase of K causes a decrease in the percentage of savings. From all these results, we notice that the influence of buffering for a Local scheme is more important for the recursive algorithm than for the non-recursive one.

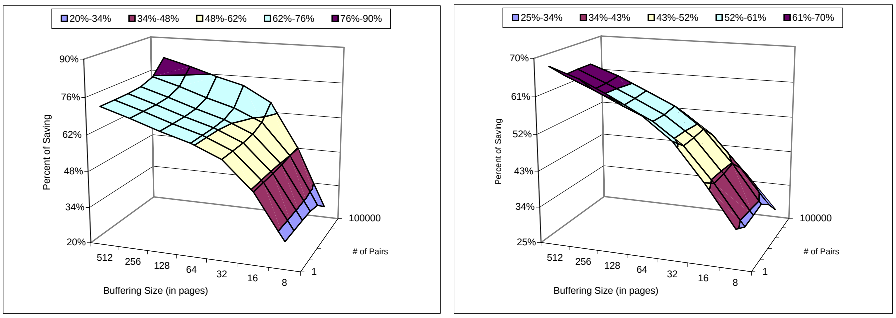


Fig. 6. The I/O cost savings of the K-CPQ recursive algorithm with LRU_L policy (left) and non-recursive algorithm with 2Q policy (right), as a function of B and the cardinalities of the data sets

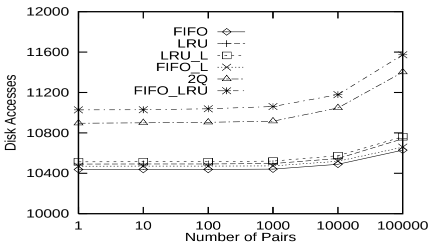
5.2 *K*-CPQ Algorithms Using a Global Buffering Scheme

For the Global buffering scheme, we have used the same parameters as for the Local one in order to investigate the best page replacement policy and buffer structure. In particular we used: (a) several replacement algorithms (FIFO, LRU, LRU_L, FIFO_L, 2Q and FIFO_LRU) for the three buffer structures, (b) the same number of pages for the buffer (B varying from 0 to 512 pages), and (c) the recursive and non-recursive algorithms for K-CPQ with K varying from 1 to 100000.

We have performed experiments with 1-CPQ using several replacement algorithms in the Global buffering scheme. When the buffer size was small or medium (e.g. $B \leq 128$), the single structure with LRU policy was the best (with 3% savings with respect to 2Q, for recursive and non-recursive algorithms), the second was the hybrid and the third one was by levels. Again, when the buffer was large (e.g. $B \geq 256$) the difference was almost negligible for all page replacement policies, although FIFO and 2Q were slightly better than the other ones for the recursive and non-recursive algorithms, respectively [8].

In the left part of Fig. 7, we depict the performance of the recursive K-CPQ algorithm for a given buffer size (e.g. $B = 512$). The best behavior is for FIFO with savings of 0.6% in relation to the LRU (for all K values), and the worst results are again for the hybrid structure (2Q and FIFO_LRU). On the other hand, the results of the non-recursive K-CPQ algorithm are illustrated in the right part of Fig. 7 for the same buffer size ($B = 512$). The best behavior arises for 2Q with savings of 0.6% in relation to LRU (for all K values), and the worst results are for FIFO_LRU.

For 1-CPQ, the buffering increased the performance of the recursive algorithm by 9% in comparison to the non-recursive one with the same page replacement policy (LRU). For K-CPQ, when the K value was incremented, this improvement was 26% approximately for $K \leq 10000$ and 47% for $K = 100000$. Besides, for increasing K values, the I/O cost of the recursive algorithm was not significantly affected, when we had a buffer of 512 pages and the best page replacement algorithm had only an extra cost of 2%. On the other hand, this extra cost was about 39% for the non-recursive algorithm using the same buffer characteristics. Moreover, the recursive variants demonstrated savings in the range 73%-81% as K increased, between the case of a 512 pages buffer and the case no buffer at all ($B = 0$). The non-recursive algorithm, under the same buffer setup, resulted in 68%-57% savings. In general, these results were very similar to the Local buffering scheme ones [8].



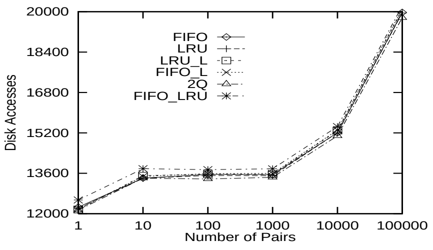
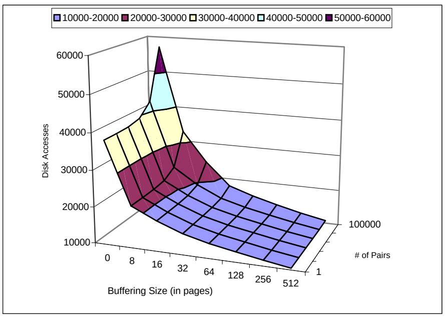


Fig. 7. The performance of the K-CPQ algorithm for different page replacement policies as a function of the buffer size for recursive (left) and non-recursive (right) algorithms and $B = 512$ pages

In Fig. 8, we see the performance of K-CPQ recursive and non-recursive algorithms as a function of buffer size ($B \geq 0$) with LRU policy. For the recursive algorithm, when $B \geq 32$, the savings in terms of disk accesses are large and almost the same for all K values. However, the savings are considerably less when $B \leq 16$, whereas for $K = 100000$ and $B = 0$ we can notice a characteristic peak. For the non-recursive algorithm, the savings trend is similar to the recursive one, but for high K values these savings become considerably less than the recursive one. For instance, if we have available enough buffer space, the recursive algorithm is the best alternative, because it provides an average I/O savings of 20% in respect to the non-recursive one for K-CPQ using LRU. For all these results, we notice that the influence of buffering for a Global scheme is more important for the recursive algorithm than for the non-recursive one in the K-CPQs, when we have enough buffer space. It is the same conclusion to that for the Local buffering scheme.



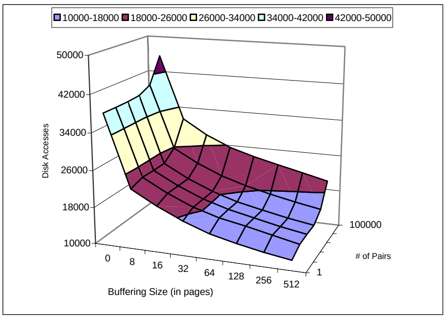


Fig. 8. The performance of K-CPQ recursive (left) and non recursive (right) algorithms with LRU policy, as a function of the buffer size and the cardinality of the data sets

5.3 Comparison of the Buffering Schemes for K-CPQ

Table 1 contains the results of an exhaustive comparison of the Local and Global buffering schemes, using the best buffer structure and page replacement algorithms for each of them. These results concern the performance of K-CPQs ($K \geq 1$) using the recursive (REC) and non-recursive (NREC) algorithms.

Table 1. Comparison of the Local and Global buffering schemes

Buffer Size	8	16	32	64	128	256	512
REC	**G** (LRU)	**G** (LRU)	**L** (LRU)	**G** (LRU)	**G** (LRU)	**G** (FIFO)	**L** (LRU_L)
NREC	**G** (LRU)	**G** (LRU)	**G** (LRU)	**G** (LRU)	**G** (2Q)	**G** (2Q)	**G** (2Q)

From this table (where L and G stand for Local and Global, respectively), we deduce that the Global buffering scheme is the best alternative in most cases, except for $B = 32$ and $B = 512$ for the recursive algorithm where the Local scheme prevails. The difference between the Global and Local schemes is around 1%-2% in terms of disk accesses for all cases. Since the difference is small, we suggest to use the Global buffering scheme, because, in this case, the buffer manager may: (a) include and handle more than two R-trees in the same buffer area, (b) give priority to a specific R-tree, (c) manage and assign dynamically more pages to one R-tree and (d) introduce global optimization techniques.

Besides, LRU is the most appropriate page replacement algorithm with a single buffer structure when the buffer size is small or medium. On the other hand, when the buffer is large the best alternatives are FIFO (single structure) and LRU_L (structure by level) for the recursive algorithm and 2Q (hybrid structure) for the non-recursive one. Since the difference between LRU and the other winner page replacement algorithms (FIFO, LRU_L and 2Q) is in the range 1%-2%, we suggest to use LRU as the policy with the best overall stable performance.

6 Conclusions and Future Work

Efficient processing of closest pairs queries (K-CPQs with $K \geq 1$) is of great importance in a wide area of applications like spatial databases, GIS, image databases, etc. Buffering is very important in DBMSs, because it improves the performance considerably (since reading from disk is orders of magnitude more expensive than reading from a buffer). In this paper we have examined the most important factors that affect the performance in the presence of a buffer. These are: the buffer structure, the page replacement algorithm, and the buffering scheme. From the experimentation we deduce the following conclusions:

- The I/O savings for the recursive algorithm are larger than that of the non-recursive one for K-CPQ when we have enough buffer space. The reason is that the use of recursion in a depth-first way is affected by the buffering scheme more than the case of a best-first search strategy implemented through a heap of minimums.
- With a fixed buffer size, increasing the number K of pairs in a CPQ for the recursive algorithm results in a negligible extra cost with respect to the additional cost for the non-recursive one.
- The Global buffering scheme is more appropriate when the buffer size is small or medium for the recursive algorithm, while the Local scheme is the best choice for large buffers. On the other hand, if we use the non-recursive algorithm, the Global buffering scheme is the best alternative for all cases.
- LRU is the most appropriate page replacement algorithm with a single buffer structure when the buffer size is small or medium, whatever the type (recursive, or non-recursive) of algorithm for K-CPQs. On the other hand, when the buffer is large, then the best alternatives are FIFO (single structure) and LRU_L (structure by levels) for the recursive algorithm and 2Q (hybrid structure) for the non-recursive one.

Future research may include:

- Study of alternative choices for the buffer structure, page replacement algorithm and buffering scheme in the Self-CPQ and Semi-CPQ [5], which are extensions of 1-CPQ and K-CPQ.
- Consideration of other spatial data structures and multi-dimensional data.
- Development of a cost model, taking into account the effect of buffering to analyze the number of disk accesses required for K-CPQs for R*-trees (along the same lines as in [14], where a cost model for range queries in R-trees has been developed).

References

1. W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza and N. MacNaughton: “The Oracle Universal Server Buffer”, *Proc. 23rd VLDB Conf.*, pp.590-594, Athens, Greece, 1997.

2. N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger: “The R*-tree: and Efficient and Robust Access Method for Points and Rectangles”, *Proc. 1990 ACM SIGMOD Conf.*, pp.322-331, Atlantic City, NJ, 1990.
3. T. Brinkhoff, H.P. Kriegel and B. Seeger: “Efficient Processing of Spatial Joins Using R-Trees”. *Proc. 1993 ACM SIGMOD Conf.*, pp.237-246, Washington, DC, 1993.
4. H.T. Chou and D.J. DeWitt: “An Evaluation of Buffer Management Strategies for Relational Database Systems”, *Proc. 11th VLDB Conf.*, pp.127-141, Stockholm, Sweden, 1985.
5. A. Corral, Y. Manolopoulos, Y. Theodoridis and M. Vassilakopoulos: “Closest Pair Queries in Spatial Databases”, *Proc. 2000 ACM SIGMOD Conf.*, pp.189-200, Dallas, TX, 2000.
6. C.Y. Chan, B.C. Ooi and H. Lu: “Extensible Buffer Management of Indexes”, *Proc. 18th VLDB Conf.*, pp.444-454, Vancouver, Canada, 1992.
7. A. Corral, M. Vassilakopoulos and Y. Manolopoulos: “Algorithms for Joining R-Trees and Linear Region Quadtrees”, *Proc. 6th SSD Conf.*, pp.251-269, Hong Kong, China, 1999.
8. A. Corral, M. Vassilakopoulos and Y. Manolopoulos: “The Impact of Buffering on Closest Pairs Queries using R-trees”, *Technical Report*, Dept. of Informatics, Aristotle University of Thessaloniki, February 2001.
9. W. Effelsberg and T. Harder: “Principles of Database Buffer Management”, *ACM Transactions on Database Systems, Vol.9, No.4*, pp.560-595, 1984.
10. A. Guttman: “R-trees: A Dynamic Index Structure for Spatial Searching”, *Proc. 1984 ACM SIGMOD Conf.*, pp.47-57, Boston, MA, 1984.
11. Y.W. Huang, N. Jing and E.A. Rundensteiner: “Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations”, *Proc. 23rd VLDB Conf.*, pp.396-405, Athens, Greece, 1997.
12. G.R. Hjaltason and H. Samet: “Incremental Distance Join Algorithms for Spatial Databases”, *Proc. 1998 ACM SIGMOD Conf.*, pp.237-248, Seattle, WA, 1998.
13. T. Johnson and D. Shasha: “2Q: a Low Overhead High Performance Buffer Management Replacement Algorithm”, *Proc. 20th VLDB Conf.*, pp.439-450, Santiago, Chile, 1994.
14. S.T. Leutenegger and M.A. Lopez: “The Effect of Buffering on the Performance of R-Trees”. *Proc. ICDE Conf.*, pp.164-171, Orlando, FL, 1998.
15. E.J. O’Neil, P.E. O’Neil and G. Weikum: “The LRU-K Page Replacement Algorithm for Database Disk Buffering”, *Proc. 1993 ACM SIGMOD Conf.*, pp.297-306, Washington, DC, 1993.
16. A. Papadopoulos and Y. Manolopoulos: “Global Page Replacement in Spatial Databases”, *Proc. DEXA’96, Conf.*, pp.855-864, Zurich, Switzerland, 1996.
17. A. Papadopoulos, P. Rigaux and M. Scholl: “A Performance Evaluation of Spatial Join Processing Strategies”, *Proc. 6th SSD Conf.*, pp.286-307, Hong Kong, China, 1999.
18. G.M. Sacco: “Index Access with a Finite Buffer”, *Proc. 13th VLDB Conf.*, pp.301-309, Brighton, England, 1987.
19. M. Stonebraker, J. Frew, K. Gardels and J. Meredith: “The Sequoia 2000 Benchmark”, *Proc. 1993 ACM SIGMOD Conf.*, pp.2-11, Washington, DC, 1993.
20. G.M. Sacco and M. Schkolnick: “A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model”, *Proc. 8th VLDB Conf.*, pp.257-262, 1982.