# Multiple Range Query Optimization in Spatial Databases *

Apostolos N. Papadopoulos and Yannis Manolopoulos

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

**Abstract.** In order to answer efficiently range queries in 2-d R-trees, first we sort queries by means of a space filling curve, then we group them together, and finally pass them for processing. Initially, we consider grouping of pairs of requests only, and give two algorithms with exponential and linear complexity. Then, we generalize the linear method, grouping more than two requests per group. We evaluate these methods under different LRU buffer sizes, measuring the cache misses per query. We present experimental results based on real and synthetic data. The results show that careful query scheduling can improve substantially the overall performance of multiple range query processing.

## 1 Introduction

Two basic research directions exist aiming at improving efficiency in a Spatial DBMS [4,8,13]. The first direction involves the design of robust spatial data structures and algorithms [2], the second one focuses on the design of clever query optimizers. Most of the work in the latter area deals with the optimization of a single (possibly complex) spatial query [1].

Here, we concentrate on range/window queries defined by a rectilinear rectangle, where the answer is composed of all objects overlapping the query rectangle. We examine methods to combine many range queries (posed by one or many users) in order to reduce the total execution time, based on the reasoning of [14]. We quote from the latter work: *the main motivation for performing such an interquery optimization lies in the fact that queries may share common data.*

There are real-life cases where many requests can be present simultaneously:

- in complex disjunctive/conjunctive queries which can be decomposed in simpler subqueries,
- in spatial join processing, where if one relation participates with only a few objects, then it is more efficient to perform lookups in the second relation,
- in spatial client/server environment, where at any given time instance more than one users may request for service.

* Work supported by the European Union's TMR program ("Chorochronos" project, contract number ERBFMRX-CT96-0056), and the national PENED program.

W. Litwin et al. (Eds.): ADBIS'98, LNCS 1475, pp. 71–??, 1998.

– in benchmarking/simulation environments, where submitted queries are generated with analytical techniques and therefore are known in advance.

For this purpose, we use space filling curves to sort query windows and apply a simple criterion in order to group queries efficiently. We consider the original R-tree [5] as the underlying access method. However, the method is applicable to any R-tree variant or any other spatial access methods with minor modifications. Although, the discussion is based on 2-d space, the generalization to higher dimensional spaces is straightforward.

The use of buffers is very important in database systems [3], since the performance improvement can be substantial. One of the policies that is widely acceptable is the LRU (Least Recently Used) policy, which replaces the page that has not been referenced for the longest time period. The performance of the proposed methods using LRU buffers of various sizes is evaluated and results show that different methods have different performance under different buffer sizes. However, it is emphasized that a careful preprocessing of the queries can improve substantially the overall performance of range query processing.

The rest of the paper is organized as follows. In Sections 2 and 3 we give the appropriate background on R-trees and space filling curves, and analytic considerations respectively. In Section 4 we describe the various techniques in detail. Section 5 contains the experimental results and performance comparisons. Finally, Section 6 concludes the paper and gives some future research directions.

## 2 Background

### 2.1 R-trees

The R-tree [5] is a multi-dimensional, height balanced tree structure for use in secondary storage. The structure handles objects by means of their Minimum Bounding Rectangles (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 enclosing all the MBRs of its descendants and *child-ptr* is the pointer to the specific child node. Leaves 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. One of the most important factors that affects the overall structure performance is the node split strategy. In [5] three split policies are reported, namely exponential, quadratic and linear split policies. More sophisticated R-tree variants have been proposed [2], however here we adopt the original R-tree structure because we mainly want to emphasize on the technique to reduce the processing cost.

### 2.2 Space Filling Curves

A Space Filling Curve is a special fractal curve which has the following basic characteristics:

- it covers completely an area, a volume or a hyper-volume in a 2-d, 3-d or $n$-d space respectively,
- each point is visited once and only once (the curve does not cross itself), and
- neighbor points in the native space are likely to be neighbors in the space filling curve.

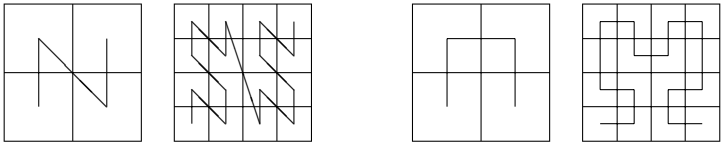

**Fig. 1.** Peano and Hilbert space filling curves.

In Figure 1 we present two of most important Space Filling Curves: Peano and Hilbert. We can easily observe the self-similarity property of the curves. A Peano curve can be constructed by interleaving the bits of coordinates $x$ and $y$. The generation of the Hilbert curve is more complex, i.e. it is constructed by means of a process that uses rotation and mirroring. Algorithms for the generation of the 2-d Hilbert curve can be found in [6,8]. The goodness of a space filling curve is measured with respect to its ability to preserve proximity. Although there are no analytical results to demonstrate the superiority of the Hilbert curve, experiments [6] show that it is the best distance preserving curve. Therefore, in the rest of the paper we focus on the Hilbert curve.

## 3 Analytical Considerations

In this section, we derive an estimate for the expected number of page references, when processing a set $\mathcal{S}$ of $N$ window queries $q_1, .., q_N$. The notations used along with their description are presented in Table 1.

Assume that the query rectangle centroids obey a uniform distribution and that the dataspace dimensions are normalized to the unit square. The expected number of page references to satisfy the query $q_i$ is [7]:

$$EPR(q_{ix}, q_{iy}) = TA + q_{ix} \cdot E_y + q_{iy} \cdot E_x + TN \cdot q_{ix} \cdot q_{iy} \tag{1}$$

where $q_{ix}$ and $q_{iy}$ are the $x$ and $y$ extends of the window query $q_i$. Equation (1) is independent of the R-tree construction method as well as independent of the data object distribution. Also, the parameters used can be calculated and maintained with negligible cost.

To simplify the analysis we focus on the R-tree leaf level. However, the analysis can be applied to all the levels. We also assume that each query window is a square with side $q$, and each data page has a square MBR with area $L_a$. Setting $q_{ix}$=$q_{iy}$=$q$, $TN$=$LN$, $E_x$=$LN \cdot \sqrt{L_a}$, $E_y$=$LN \cdot \sqrt{L_a}$, $TA$=$LN \cdot L_a$, we get:

$$EPR(q) = LN \cdot (q^2 + 2 \cdot \sqrt{L_a} \cdot q + L_a)$$

| Symbol | Description |
|---|---|
| $N$ | number of pending range queries |
| $q$ | query window side |
| $Q$ | super window side |
| $TN$ | total number of R-tree nodes |
| $TA$ | sum of areas of all nodes |
| $LN$ | number of R-tree leaves |
| $L_a$ | area of the MBR of a leaf |
| $E_x$ | sum of $x$ extends of all R-tree nodes |
| $E_y$ | sum of $y$ extends of all R-tree nodes |
| $EPR(q)$ | expected number of page references for query $q$ |
| $TEPR(\mathcal{S})$ | total expected number of page references for set $\mathcal{S}$ |
| $DPR(\mathcal{S})$ | number of distinct page references for set $\mathcal{S}$ |
| $ERPP(q,Q)$ | expected number of references per page |

**Table 1.** Notations used throughout the analysis.

Since there are $N$ requests, the total number of page references (including the redundant ones) is:

$$TERP(\mathcal{S}) \;=\; \sum_{k=1}^{N} EPR(q) \;=\; N \cdot LN \cdot (q^2 + 2 \cdot \sqrt{L_a} \cdot q + L_a) \qquad (2)$$

We can associate to the $N$ window queries, a super-window $SW$ which corresponds to the MBR of all query windows. For simplicity let $SW$ be a square with side $Q$. We would like to have an estimate for the number of distinct page references (i.e. excluding redundant references). We can approximate this number by the number of page references introduced when processing $SW$. However, this approximation is not accurate if $N$ is small and $q \ll Q$. In this case the number of distinct page references includes a large number of pages referenced by $SW$, but not referenced by any $q_i$. Ignoring this effect we get:

$$DPR(\mathcal{S}) = DPR(Q) = LN \cdot (Q^2 + 2 \cdot \sqrt{L_a} \cdot Q + L_a) \qquad (3)$$

We can use Formulae (2) and (3) to derive the expected number of references per page:

$$ERPP(q,Q) \;=\; \frac{TEPR(\mathcal{S})}{DPR(\mathcal{S})} \;=\; N \cdot \left( \frac{q + \sqrt{L_a}}{Q + \sqrt{L_a}} \right)^2 \qquad (4)$$

From Equation (4) we observe that as $q$ approaches $Q$, $ERPP(q,Q)$ approaches $N$. Obviously, in the extreme case where $q$=$Q$, we get $ERPP(q,Q)$=$N$. This is exactly the case where all $N$ query windows represent the same portion of the dataspace. Figure 2 depicts $ERPP(q,Q)$ as a function of $Q$ for $N$=100, and $q$=0 (point queries), $q$=0.1 (small window queries), $q$=0.3 (large window queries). The value of $\sqrt{L_a}$ was set to 0.01, i.e. the area of the MBR of each leaf covers 1% of the dataspace area.

The graphs show that a single page may be references many times during the processing of the $N$ window queries. If a page reference causes a disk access,

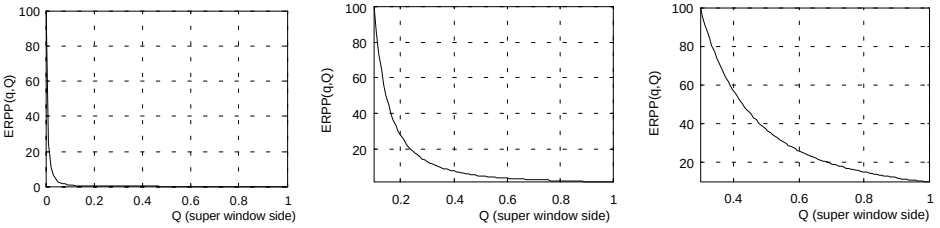


**Fig. 2.** Plots of $ERPP(q, Q)$ vs. $Q$ for $N$=100, $\sqrt{L_a}$=0.01 and $q$=0, 0.1, 0.3.

the number of I/O operations increases substantially. However, with adequate cache buffers, the total processing cost may be reduced. In the following section we discuss several techniques to solve the problem.

## 4 Processing Multiple Requests

In this section we study several alternatives to service a number of window queries. The processing cost of a window query $q_i$ is mainly affected by the I/O time to fetch the appropriate disk pages and the CPU time to process them. For the rest of the paper we focus on the I/O activity ignoring the CPU time as a negligible cost.

A common approach to service a number of requests is to process them in a First-Come-First-Served (**FCFS**) manner. Clearly, in case of low rate of query arrivals (e.g. one query per minute), **FCFS** is a reasonable service strategy. However, there is a major problem with this approach in other cases (see Section 1). If the order of processing follows the arrival order, then the probability to have a cache hit is very small, leading to poor cache utilization. However, we can take advantage of the fact that we have knowledge of all pending requests and improve the performance.

Our first attempt is to perform a quick preprocessing of all pending query windows, in order to increase the probability that a page required is residing into the cache buffer. The first algorithm **HS** (Hilbert Sorting) has as follows:

**Algorithm HS**

**[HS1] For** each $q_i \in \mathcal{S}$ calculate the Hilbert value of the window's centroid.

**[HS2]** Sort the Hilbert values in increasing order to obtain a total order of the query windows, $q_{i1}, .., q_{iN}$.

**[HS3] For all** $q_{ij}$, $j$=1..$N$ execute query $q_{ij}$.

The method guarantees (up to a point) that nearby requests will be executed sequentially, thus enhancing the locality of references. The main observation is that the method depends heavily on the size of the cache buffer. Moreover, if there is no buffer space, the algorithm has the same performance with the **FCFS** method. This drawback motivates us to go one step further.

Consider two requests $q_i$ and $q_j$. If these requests share common pages, we could execute them as one. What we need is a criterion to decide when to group these queries and when to execute them individually. We can use Equation (1)

to determine if the grouping of queries $q_i$ and $q_j$ is advantageous or not. Let $Q$ denote the MBR of the query windows $q_i$ and $q_j$ and $Q_x$, $Q_y$ its $x$ and $y$ extend respectively. If we execute $Q$ instead of both $q_i$ and $q_j$, there will be a reduction in disk accesses if and only if:

$$\begin{aligned} EPR(Q_x, Q_y) \leq EPR(q_{ix}, q_{iy}) + EPR(q_{jx}, q_{jy}) \Leftrightarrow \\ E_y \cdot (Q_x - q_{ix} - q_{jx}) + E_x \cdot (Q_y - q_{iy} - q_{jy}) \ + \\ TN \cdot (Q_x \cdot Q_y - q_{ix} \cdot q_{iy} - q_{jx} \cdot q_{jy}) - TA \ \leq \ 0 \end{aligned} \tag{5}$$

In Inequality (5) above we observe that:

- the factors $E_y \cdot (Q_x - q_{ix} - q_{jx})$ and $E_x \cdot (Q_y - q_{iy} - q_{jy})$ are negative if the two query rectangles overlap in space.
- the value of the factor $TN \cdot (Q_x \cdot Q_y - q_{ix} \cdot q_{iy} - q_{jx} \cdot q_{jy})$ decreases as the overlap increases.

However, two queries may share common pages even if they do not intersect. Consequently, as the distance between the two query rectangles in the native space decreases, the gain in disk accesses increases. If two range queries satisfy Inequality (5), this is a clear criterion that with high probability there will be a reduction in the number of disk accesses.

Based on this simple grouping criterion, let us proceed with the construction of efficient algorithms where this criterion can be valuable. First we present an exhaustive method with exponential complexity, and next we give a simple greedy method along with its extension, with linear complexity. Beforehand, we emphasize that these algorithms are based on the following assumptions:

1. the window queries have been ordered according to the Hilbert value of their rectangle centroids (with $O(N \cdot \log N)$ cost).
2. grouping is allowed only to neighbor queries (with respect to Hilbert order).

Later we will generalize to more queries and discuss the pros and cons of such an approach.

### 4.1 The Exponential Algorithm (Algorithm E)

Consider $N$ unserviced range query requests. For a query $q_j$, $j$=1..$N$ there are three alternatives:

1. the query will be executed individually,
2. the query will be combined with its left sibling,
3. the query will be combined with its right sibling.

Therefore, to determine a promising processing schedule, we could consider all possible schedules and select the one with the minimum total execution cost.
**Proposition**. The number of all possible schedules derived for $N$ range queries under the constraint that each request can be executed either alone or together with its left or right sibling is: $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^{N+1}\right)$. □

It is evident that the execution cost of the algorithm is very high, and therefore unacceptable for practical use in real applications.

### 4.2 The Linear Algorithm (Algorithm L)

We may reduce the algorithmic complexity by using a simple greedy method. The idea is: given two requests $q_1$ and $q_2$, just check if Inequality (5) is satisfied or not. If yes, we will execute the queries as one. If not, we will execute $q_1$ alone and proceed with $q_2$ and $q_3$ until we consider all pending requests.

**Algorithm L**

**[L1] For** each $q_i \in \mathcal{S}$ calculate the Hilbert values of the window's centroid. Sort the Hilbert values in increasing order to obtain a total order of the query windows.

**[L2]** Let *pos* denote the current query index. Initialize *pos*=1.

**[L3] while** ($pos < N$) **do**

**begin**

Test Inequality (5) for query rectangles $q_{pos}$ and $q_{pos+1}$;

**if** Inequality (5) is satisfied

**then** process the two queries as one **and** set $pos$=$pos$+2

**else** process query $q_{pos}$ **and** set $pos$=$pos$+1.

**end**

**if** ($pos$==$N$) **then** service $q_{pos}$.

Clearly, the complexity of the algorithm is $O(N \cdot \log N)$, since in step **[L1]** we sort the rectangles. Step **[L3]** takes only $O(N)$, because the queries are scanned only once.

### 4.3 The Extended Linear Algorithm (Algorithm ExL)

In this subsection we relax the constraint that at most two queries can be executed as one. Algorithm **ExL** derived is an extension of the **L** algorithm, enabling the grouping of more than two window queries.

Consider the queries $q_1, .., q_N$, in increasing order with respect to the Hilbert value of the rectangle centroid. The algorithm tries to pack requests into disjoint sets. We begin with request $q_1$. Initially the first group $G_1$ contains only $q_1$ ($G_1$=$\{q_1\}$). If the processing of $q_1$ and $q_2$ together retrieves less pages than the processing of $q_1$ plus $q_2$ (according to Inequality (5)), then $G_1$=$\{q_1, q_2\}$. If the processing of $q_3$ and $q_2$ and $q_1$ together retrieves less pages than the processing of $q_3$ plus $q_2$ plus $q_1$, then $G_1$=$\{q_1, q_2, q_3\}$. We continue the same process, until we reach a request $q_k$ such that $EPR(G_1 + q_k) > EPR(G_1) + EPR(q_k)$. When this happens we set $G_1$=$\{q_1, .., q_{k-1}\}$ and $G_2$=$\{q_k\}$. Therefore, a second group $G_2$ is considered. This process is continued until all requests are examined.

**Algorithm ExL**

**[ExL1] For** each $q_i \in \mathcal{S}$ calculate the Hilbert values of the window's centroid. Sort the Hilbert values in increasing order to obtain a total order of the query windows.

**[ExL2]** Let *pos* denote the current query index. Initialize *pos*=1.
Let $GroupId$ denote the current group. Initialize $GroupId$=1.

**[ExL3] while** ($pos < N$) **do**

**begin**

Initialize $EndOfGroup$=$FALSE$ **and** $G_{GroupId}$=$\{q_{pos}\}$;
**while** (!$EndOfGroup$) **do**
**begin**
**if** ($EPR(G_{GroupId} + q_{pos}) < EPR(G_{GroupId}) + EPR(q_{pos})$)
**then** assign $q_{pos}$ to $G_{GroupId}$ **and** set $pos$=$pos$+1;
**else** set $EndOfGroup$=$TRUE$ **and** set $GroupId$=$GroupId$+1;
process as one all $q_j$'s $\in G_{GroupId}$;
**end**
**end**
**if** ($pos$==$N$) **then** service $q_{pos}$.

Provided that the query windows have been already sorted with respect to the Hilbert value of their centroid, the time complexity of step **[ExL3]** is linear to the number of requests ($O(N)$).

Finally, another major issue is the separation of the results. After the processing of a multiple range query, we must determine which objects correspond to specific range queries. This operation is CPU-bound and can be performed using computational geometry [12] techniques in order to find the queries that a specific object geometry satisfies.

## 5 Experimental Results

### 5.1 Preliminaries

We implemented the R-tree access method with the quadratic split policy, and the algorithms **HS**, **L** and **ExL** in the C programming language under UNIX. The experimentation was performed on DEC 3000 workstation. The page size was set to 2Kbytes. The dataspace dimensions were set to the unit square $[0, 1) \times [0, 1)$ and all datasets were normalized to fall inside the dataspace area. The buffer sizes (in Kbytes) considered in this paper are: 0, 8, 32, 128, 512 and 1024 (i.e. 1 Mbyte). The different datasets used throughout the evaluation of the methods are presented in the next table.

| Dataset | Representation of | Object | Population | Source |
|---|---|---|---|---|
| CP | California places | points | 65,252 | s2k-ftp.cs.berkeley.edu |
| CUA | California Urban and Agricultural | rectangles | 12,361 | s2k-ftp.cs.berkeley.edu |
| MG | Montgomery County | rectangles | 39,323 | http://www.census.gov |
| LB | Long Beach County | rectangles | 53,146 | http://www.census.gov |

**Table 2.** Datasets used for experimentation.

The major factors that affect the performance of the algorithms are:

- the number of pending window queries,
- the size of the LRU buffer,
- the characteristics of the query windows (i.e. area, perimeter) and

– the characteristics of the dataset (i.e. distribution, coverage, geometry).

Let us investigate the impact of these factors. Figure 3 presents the results for the MG+LB dataset and Figure 4 the results for the CP dataset. Each one of these figures comprises of two parts:

1. The left part ((a) to (c)) presents the cache misses per query when the varying quantities are the LRU buffer size and the number of pending range queries, whereas the query window side is fixed at 0.05.
2. The right part ((d) to (f)) presents the cache misses per query when the varying quantities are the LRU buffer size and the query window side. The number of pending range queries is fixed at 100.

### 5.2 Interpretation of Results

From Figures 3 and 4 some very interesting observations can be derived. It is easily understood that:

– as the buffer size increases the performance of all methods is improved,
– the more the pending range queries, the more efficient is the derived processing plan,
– as the side of the query window increases, the performance improvement is more significant,
– when the R-tree stores points (CP dataset), the R-tree nodes have (generally) less area and perimeter (in comparison to other datasets) and thus the probability that a page will be referenced by more than one queries decreases. Therefore, the performance improvement of the proposed method in comparison to **FCFS** is less significant (but still present).

By inspecting closer Figures 3 and 4 we derive that:

– The **HS** algorithm, for 0Kbyte buffer has identical performance with the **FCFS** method, since the locality of references is not utilized at all. For LRU buffer sizes ranging between 8Kbytes and 32Kbytes, the performance improvement of **HS**, is around 5% over the **FCFS** method (in some cases reaches 20% for 32Kbytes buffer). However, for large buffer sizes (128Kbytes and above) **HS** is the best choice. In such cases the improvement over the **FCFS** method ranges from 20% to 60%. As stated in a previous, the performance of this algorithm is highly related to the LRU buffer size. The only thing that this algorithm can guarantee, is the locality of references. However, it is not certain that all $N$ requests will be processed without other requests interfering.
– Algorithm **ExL** is the best choice, when no LRU buffer is available. We observe that in cases where the queries cover a large portion of the dataspace (Figures 3f and 4f) **ExL** can achieve up to 80% improvement over **FCFS**. As the buffer size increases, **HS** and **L** are clearly better than **ExL**. This is due to the fact that grouping more than two queries can lead to a large number of unnecessary page accesses.

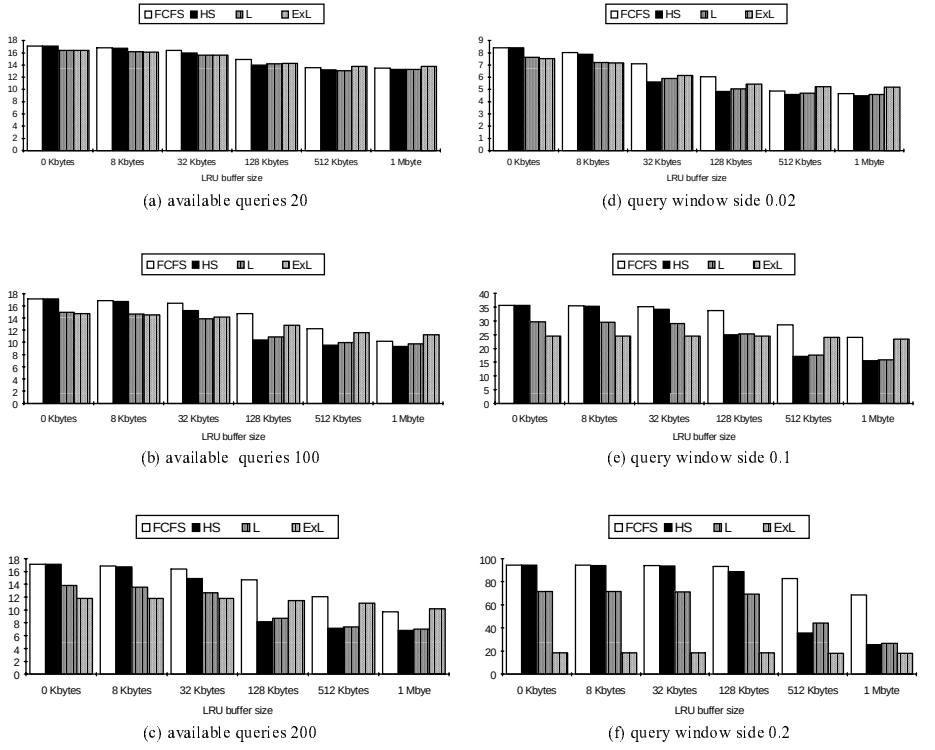


**Fig. 3.** MG+LB dataset. Left Part: Cache misses per query versus LRU buffer size and number of pending queries (for $q$=0.05). Right Part: Cache misses per query versus LRU buffer size and query window side (for $N$=100).

- Algorithm **L** keeps the balance between algorithms **HS** and **ExL**. For no (or small) LRU buffers, its performance is very close to that of **ExL**, whereas for large LRU buffers, its performance is very close to that of **HS**. In general, **L** achieves a 30% performance improvement over **FCFS**.
- When the LRU buffer size is large (e.g. > 1Mbyte), and the number of pending queries is small (e.g. 20), the performance of **FCFS** method is very close to that of algorithm **L** and slightly better than that of algorithm bf ExL. The reason for this is the number of extra pages fetched by **L** and **ExL**.

## 6 Conclusions

We proposed a global query optimization technique to improve the performance of a Spatial DBMS when answering multiple range queries. The main result is that a careful preprocessing of the queries can lead to substantial reduction of the number of disk accesses and better cache utilization, in comparison to the **FCFS** method. This goal has been achieved by satisfying two main needs:

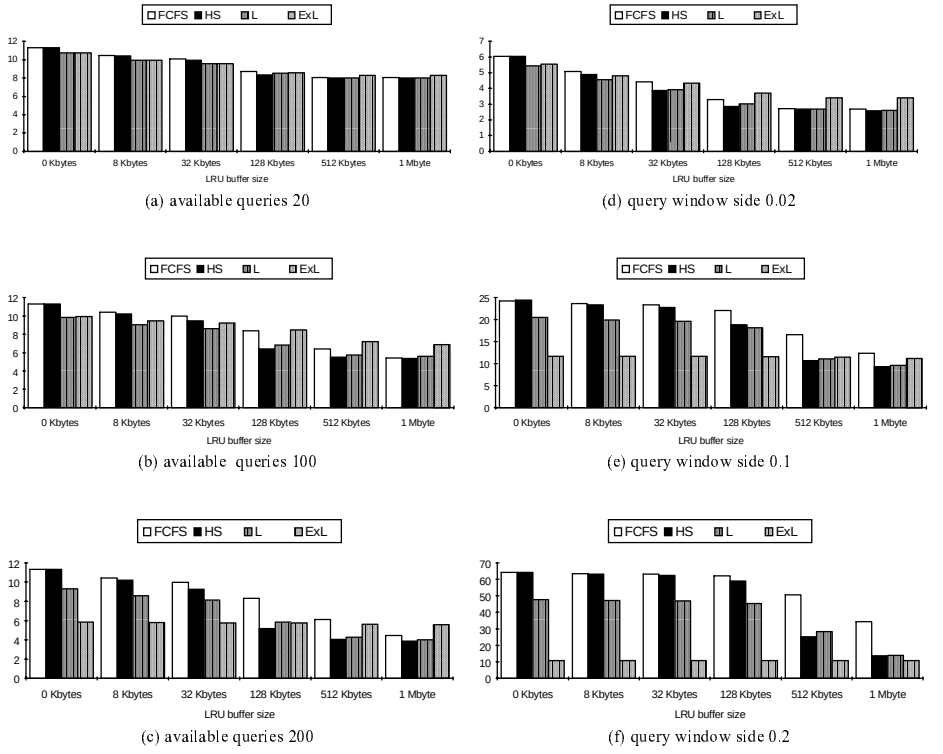


**Fig. 4.** CP dataset. Left Part: Cache misses per query versus LRU buffer size and number of pending queries (for $q$=0.05). Right Part: Cache misses per query versus LRU buffer size and query window side. (for $N$=100).

1. bring "*similar*" query rectangles close to each other, and
2. provide special algorithms to aid the reduction of disk accesses.

To satisfy need 1, we have used the Hilbert space filling curve, and Algorithm **HS** is based only on this sorting according to Hilbert values. To satisfy need 2, we provide three algorithms (**E**, **L** and **ExL**) to combine neighbor (according to Hilbert order) query rectangles. Algorithms **L** and **ExL** are linear and exhibit considerable gain when compared to the conventional **FCFS** approach. We tested our method under different data sets and different LRU buffer sizes. We suggest using algorithm **HS** for large buffers and algorithm **L** in all other cases. Although algorithm **ExL** introduces a substantial improvement in some cases, if the requests represent queries of different users, may impose a large waiting time, as opposed to algorithm **L** which does not introduce much overhead.

Future research may include:

– Application of the method to other R-tree variants and other spatial data structures, modifying accordingly the formula deriving the expected number of disk accesses for a range query (Equation (1)),

- Evaluation of the method when the query distribution follows the object distribution, i.e. each object has the same probability to be retrieved [11],
- Global optimization by considering other more complex spatial queries (e.g. spatial join).

## References


1. W. Aref: "Query processing and optimization in spatial databases", Technical Report CS-TR-3097, Department of Computer Science, University of Maryland at College Park, MD, 1993.
2. V. Gaede and O. Guenther: "Multidimensional access methods", *ACM Computer Surveys*, to appear. Address for downloading: http://www.wiwi.huberlin.de/gaede/survey.rev.ps.Z.
3. J. Gray and A. Rueter: *Transaction processing - concepts and techniques*, Morgan Kaufmann, San Francisco, CA, 1993.
4. R.H. Gutting: "An introduction to spatial database systems", *The VLDB Journal*, vol.3, no.4, pp.357-399, 1994.
5. A. Guttman: "R-trees: a dynamic index structure for spatial searching", *Proceedings of the 1984 ACM SIGMOD Conference*, pp.47-57, Boston, MA, 1984.
6. H.V. Jagadish: "Linear clustering of objects with multiple attributes", *Proceedings of the 1990 ACM SIGMOD Conference*, pp.332-342, Atlantic City, NJ, 1990.
7. I. Kamel and C. Faloutsos: "On packing R-trees", *Proceedings of the 2nd Conference on Information and Knowledge Management (CIKM)*, Washington DC, 1993.
8. R. Laurini and D. Thompson: *"Fundamentals of spatial information systems"*, Academic Press, London, 1992.
9. V. Ng and T. Kameda: "Concurrent accesses to R-trees", *Proceedings of 3rd International Symposium on Large Spatial Databases (SSD '93)*, pp.142-161, Singapore, 1993.
10. J. Orenstein: "Spatial query processing in an object-oriented database system", *Proceedings of the 1986 ACM SIGMOD Conference*, pp.326-336, Washington DC, 1986.
11. 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.
12. F.P. Preparata and M.I. Shamos: *Computational geometry: an introduction*, Springer-Verlag, New York, 1985.
13. H. Samet: *The design and analysis of spatial data structures*, Addison-Wesley, Reading, MA, 1990.
14. T. Sellis: "Multiple query optimization", *ACM Transactions on Database Systems*, vol.13, no.1, pp.23-52, 1988.