Window Query Processing in Linear Quadtrees

Ashraf Aboulnaga
Computer Sciences Department
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI 53706
ashraf@cs.wisc.edu
Phone: (608) 262-6623
Fax: (608) 262-9777

Walid G. Aref
Department of Computer Sciences
Purdue University
1398 Computer Science Building
West Lafayette, IN 47907
aref@cs.purdue.edu
Phone: (765) 494-1997
Fax: (765) 494-0739

Abstract

The linear quadtree is a spatial access method that is built by decomposing the spatial objects in a database into quadtree blocks and storing these quadtree blocks in a B-tree. It is very useful for geographic information systems because it provides good query performance while using existing B-tree implementations. In this paper, we present an algorithm for processing window queries in linear quadtrees that can handle query windows of any shape in the general case of spatial databases with overlapping objects. The algorithm recursively decomposes the space into quadtree blocks, and uses the quadtree blocks overlapping the query window to search the B-tree. We also present a cost model that estimates the I/O cost of processing window queries using our algorithm. The cost model is also based on a recursive decomposition of the space, and it uses very simple parameters that can easily be maintained in the database catalog. Experiments with real and synthetic data sets verify the accuracy of the cost model.

Keywords: spatial databases, spatial access methods, quadtrees, window queries, GIS

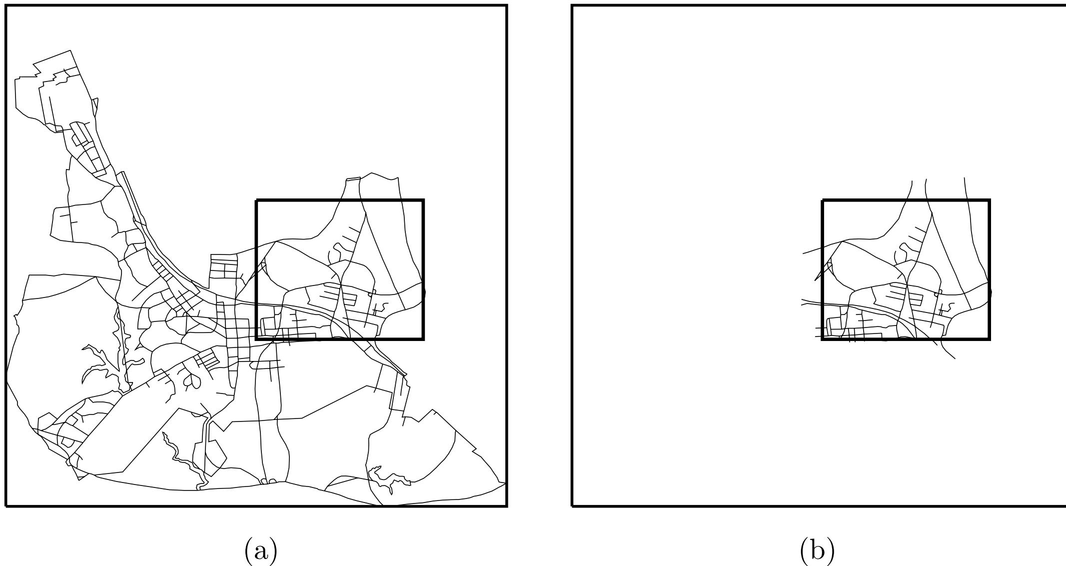


Figure 1: (a) a road map, and (b) the result of a window query operation on it

1 Introduction

Database systems that support spatial data for GIS applications use spatial indexes, also known as *spatial access methods*, to speed up access and queries. The *linear quadtree* is one such spatial access method, offering good query performance while relying on the efficient and robust B-tree index.

One operation for which a spatial access method is very useful is the spatial selection, also known as the *window query*. The window query operation retrieves all spatial objects overlapping a given region of the space known as the *query window*. The query window is usually a rectangle, but it can also be any arbitrarily shaped region. Figure 1 shows an example of the window query operation.

To fully support a spatial access method like the linear quadtree, a database system must be able to estimate the cost of using it for different operations, such as the window query operation, for example. Cost estimates are required by the query optimizer to compare different alternative query execution plans and choose the cheapest one. For example, a selection would use an index only if the cost of using it is lower than the cost of a full file scan.

In this paper, we present an algorithm for processing window queries in linear quadtrees and a cost model that estimates the cost of using this algorithm. The algorithm assumes a database in which the spatial objects are allowed to overlap and can handle arbitrarily shaped query windows. It is based on recursively decomposing the space into quadtree blocks and searching for these quadtree blocks in the linear quadtree index. The cost model is also based on a recursive decomposition of the space into quadtree blocks. The focus of the cost model is estimating the *I/O cost* of processing a window query, and it uses two very simple parameters of the linear quadtree index that can easily be maintained in the database catalog. Experiments using real and synthetic data sets verify the accuracy of the cost model.

The rest of this paper is organized as follows. In Section 2, we present an overview of related work. Section 3 describes the linear quadtree spatial access method. In Section 4, we present the general window query algorithm. In Section 5, we present the cost model. Section 6 presents an experimental evaluation of the cost model. Section 7 contains concluding remarks.

2 Related Work

Several spatial access methods have been proposed in the literature. The focus of this paper is the linear quadtree, which is described in detail in the next section. Another spatial access method that is very popular and widely used is the R-tree [Gut84], a height balanced tree storing the *minimum bounding rectangles* (MBRs) of the objects being indexed. More information about these and other spatial access methods can be found in [GG98] and [Sam90a, Sam90b].

Algorithms for the window query operation in linear quadtrees are presented in [OM88] and [AS97]. In [OM88], the window query operation is treated as a special case of the "spatial filter" join operation. A query window is decomposed into quadtree blocks and the window query is treated as a join between these quadtree blocks and the quadtree blocks representing the database. The join algorithm used is a modified merge join. In [AS97], a query window is decomposed into quadtree blocks. The linear quadtree index is used to retrieve the quadtree blocks of the database that overlap the window quadtree blocks. An approach based on *active borders* [ST85] is used to ensure that every quadtree block of the database that overlaps the query window is retrieved exactly once, thereby minimizing I/O. This algorithm only works for databases in which the spatial objects do not overlap. Our algorithm is compared to these two algorithms in Section 4.

Several cost models for window queries in *R-trees* have been proposed [PSTW93, KF93, FK94, TS96, Aok99], but we are not aware of any published cost models for window queries in linear quadtrees, despite the importance of linear quadtrees as a spatial access method. Like [PSTW93] and [KF93], our cost model assumes that the data in the underlying spatial database is uniformly distributed. In particular, we assume that the *quadtree blocks* representing the data are uniformly distributed in the space. More accurate approximations of the spatial data distribution include assuming that the data is self-similar and using the concept of fractal dimension as in [FK94], using the average number of objects per point for some representative points in the space as in [TS96], or using a histogram to represent the spatial data distribution as in [APR99] and [AN00]. As we shall see later in the paper, for our approach based on a top down decomposition of the space, assuming uniformity is quite adequate for estimating the I/O cost of window queries in linear quadtrees. The more accurate (and more expensive) approximations of the data distribution are useful for estimating the selectivity of window queries [APR99] or their total CPU and I/O cost on polygonal data sets [AN00].

3 The Linear Quadtree

The quadtree is a hierarchical data structure based on recursively decomposing the space. Each quadtree node represents a rectangular region of the space and can have up to four children, each representing a quadrant of the region represented by the parent quadtree node. The root node of the quadtree represents the entire space. Variants of the quadtree can be used to store points, lines, or regions [Sam84, Sam90b].

The quadtree is a very useful data structure for many spatial database applications. For example, it is used in the Sloan Digital Sky Survey to build indexes for different views of the sky (different "catalogs") [SKT+00]. Figure 2 shows a region in an 8×8 space represented as a quadtree. The rectangular (or in this case square) regions 1–5 are known as *quadtree blocks*.

The linear quadtree spatial access method is based on representing the spatial data as a quadtree (i.e., decomposing the spatial objects into quadtree blocks), encoding the non-empty quadtree blocks (i.e., the quadtree blocks containing data) as integers, and storing these integers in a B-tree index [Gar82]. The quadtree blocks are encoded as integers known as *Morton blocks*, which we describe

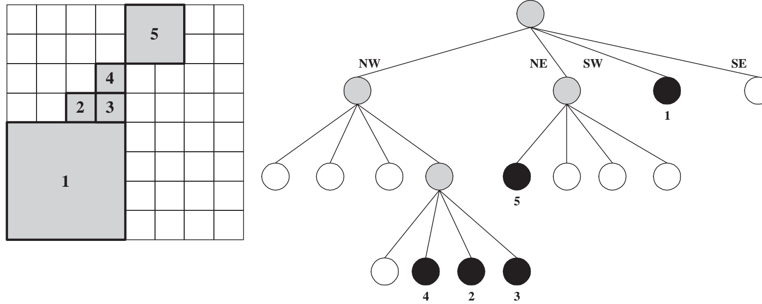


Figure 2: A region represented as a quadtree

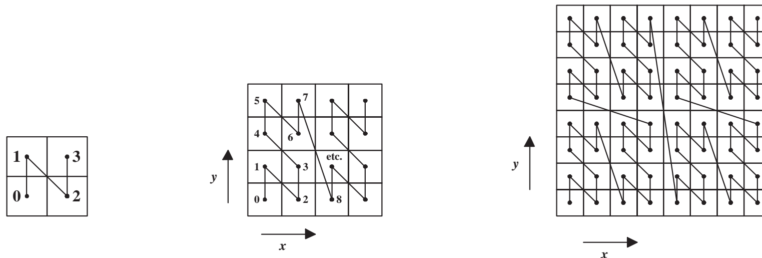


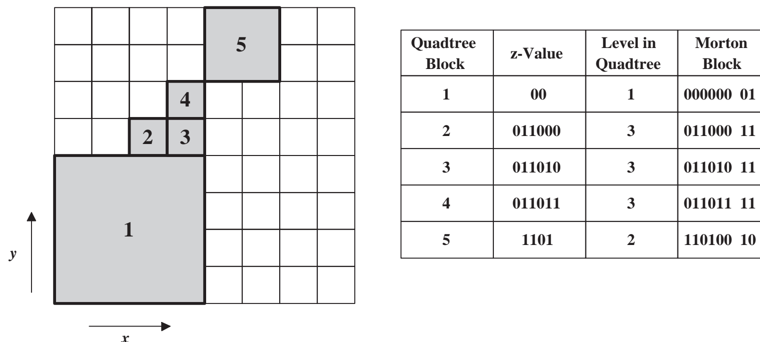
Figure 3: The Morton order for 2×2, 4×4, and 8×8 spaces

next.

A *Morton block* is an integer representing a quadtree block. Morton blocks are based on the Morton order [Mor66], also known as the z-order [Ore86]. The Morton order is a *space filling curve*, a line that visits every point in a two-dimensional space exactly once in a predetermined order. The order in which the space filling curve visits the points of the two-dimensional space maps the two-dimensional space to the one-dimensional space of integers. Figure 3 shows the recursive definition of the Morton order. If the figure is turned sideways, the Morton order resembles the letter Z, which is why it is referred to as the z-order in [Ore86]. The Morton order of a point can be very efficiently computed by interleaving the bits of its x and y coordinates. Other space filling curves include the Gray order and the Hilbert order [Fal88, FR89, Jag90, AM90].

The Morton order encodes the points of a two-dimensional space as integers. If n-bit integers are used for the x and y coordinates of the points, the Morton order representing a point would be a $2n$-bit integer. To encode entire quadtree blocks, and not just points, as integers, we use $(2n + l)$-bit integers. In the $2n$ most significant bits of the integer representing a quadtree block, we store the Morton order of its lower left corner. In the l least significant bits of this integer, we store the level in the quadtree of the quadtree block. The quadtree is a hierarchical data structure, and the level in the quadtree of a quadtree block is the number of edges between the quadtree node representing this block and the root of the quadtree. Alternately, we can view the level in the quadtree of a quadtree block as the number of times the space has to be decomposed to get this quadtree block. The root of the quadtree – the quadtree node representing the entire space – is at level 0. As such, l is the number of bits required to represent a level in the quadtree. $l = \log_2(\text{height of the quadtree})$. The $(2n + l)$-bit integer representing a quadtree block is called the *Morton block* corresponding to this quadtree block.

The Morton order of the lower left corner of a quadtree block identifies the location of the block in the two-dimensional space, but it does not identify its size. The same point can be the lower left corner of many quadtree blocks at different levels of the quadtree. Including the quadtree level in



Quadtree Block	z-Value	Level in Quadtree	Morton Block
1	00	1	000000 01
2	011000	3	011000 11
3	011010	3	011010 11
4	011011	3	011011 11
5	1101	2	110100 10

Figure 4: The region in Figure 2 and the Morton blocks representing it

the representation uniquely identifies the size of the quadtree block and removes any ambiguity.

In [OM88], quadtree blocks are represented as bit strings known as z-values. To get the z-value corresponding to a quadtree block, we start with the entire space and repeatedly apply a quadtree decomposition until we get the required quadtree block. At each stage of the decomposition, we append the bits 00, 01, 10, or 11 to the z-value depending on whether the decomposition takes us into the SW, NW, SE, or NE quadrant, respectively (the same sequence used to define the Morton order). To get the Morton block from a z-value, we store the z-value left-justified in the most significant $2n$ bits and the level in the quadtree of the quadtree block in the least significant l bits.

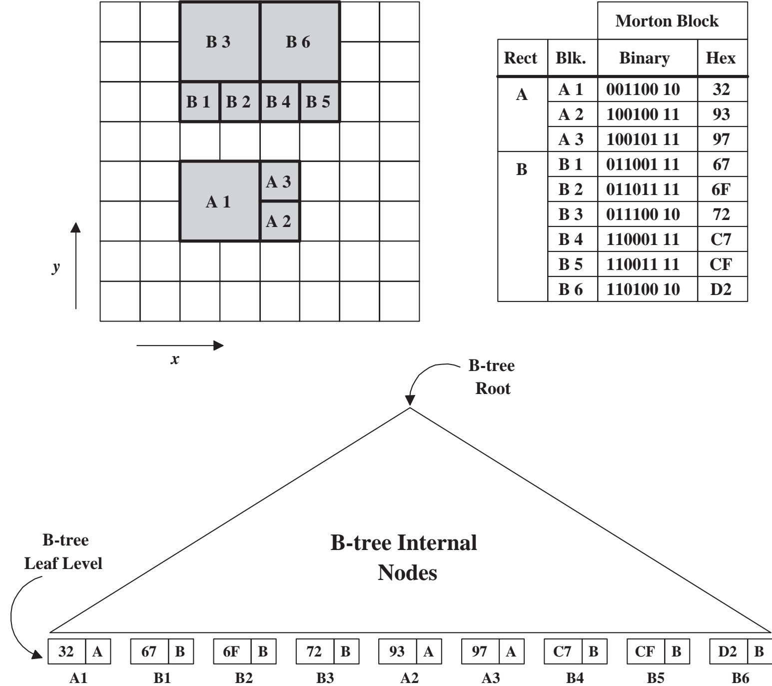
In this paper, we use the term "Morton block" and not "z-value" to emphasize the fact that we are using integers and not strings to represent quadtree blocks. Furthermore, z-values as presented in [OM88] can represent regions that are twice as high as they are wide, not just quadtree blocks. Figure 4 shows the z-values and Morton blocks for the quadtree blocks in Figure 2.

In the linear quadtree spatial access method [Gar82], the non-empty quadtree blocks are represented as Morton blocks, and these Morton blocks are stored in a B-tree (a B^+-tree, to be more specific) [Com79]. In addition to the Morton block, each B-tree entry contains the id of the spatial object to which the block belongs. Figure 5 shows a linear quadtree representing two rectangles A and B.

To represent a spatial object in a linear quadtree, we can decompose it into the maximal quadtree blocks overlapping it and store these quadtree blocks as Morton blocks in the B-tree. However, such an exact decomposition may require a large number of quadtree blocks to represent each object, which means that the B-tree may have many more entries than there are spatial objects. This can degrade the performance of the linear quadtree index. Instead, we may want to limit the number of quadtree blocks to which an object is decomposed by storing an *approximate* representation of the object in the linear quadtree. Since the linear quadtree is used in the filtering step of a *filter and refine* query processing strategy, it is acceptable to have it store a conservative approximation of the spatial objects, because the objects retrieved from the linear quadtree are further tested in the refinement step [Ore89].

In a linear quadtree, each spatial object is decomposed into multiple quadtree blocks so the number of entries in the index is greater than the number of spatial objects. Furthermore, each two-dimensional query is converted to a series of one-dimensional searches in the B-tree. Despite these apparent drawbacks, the linear quadtree does offer very good query performance. Moreover, the linear quadtree can be used to represent any variant of the quadtree data structure, so it can be used for points, lines, or regions [Sam90b].

A very important advantage of the linear quadtree is that it uses the B-tree index. The B-tree is the workhorse index for current commercial database systems, offering excellent performance



		Morton Block	
Rect	Blk.	Binary	Hex
A	A 1	001100 10	32
	A 2	100100 11	93
	A 3	100101 11	97
B	B 1	011001 11	67
	B 2	011011 11	6F
	B 3	011100 10	72
	B 4	110001 11	C7
	B 5	110011 11	CF
	B 6	110100 10	D2

Figure 5: A linear quadtree representing two rectangles

for both queries and updates. There are specialized algorithms for concurrency control and recovery in B-trees, and all commercial database systems have well-established highly tuned B-tree implementations that scale well with the data size and the number of concurrent users. The linear quadtree leverages all these desirable properties of the B-tree index, so it is especially useful for adding support for spatial data on top of a relational infrastructure in object-relational database systems. For example, the linear quadtree is used in the recently released Oracle Spatial product (http://technet.oracle.com/products/spatial/) [Ora99].

4 The Window Query Algorithm

In this section, we present an algorithm for processing window queries in linear quadtrees. The algorithm works for query windows of any shape and assumes the general case of a database with overlapping spatial objects. If the spatial features represented in the database do overlap (e.g., roads), then we must obviously assume a database with overlapping objects. Furthermore, even if the spatial features represented in the database do *not* overlap (e.g., land use), assuming a database with overlapping objects is still the correct assumption when the linear quadtree represents a conservative approximation of the objects and not an exact decomposition. As we saw in the previous section, this is the common case for linear quadtree indexes. In this case, the approximate representations of the objects in the linear quadtree may overlap even though the spatial features corresponding to these objects do not overlap.

The window query algorithm is given in Figure 6. The algorithm is based on recursively decomposing the space into quadtree blocks. This recursive decomposition terminates when it reaches the maximal quadtree blocks totally enclosed in the query window. The initial call to the algorithm is WindowQuery (w, r), where w is the query window and r is the quadtree block representing the entire space (the root of the quadtree). With each recursive call, the algorithm searches for the input

```
algorithm WindowQuery (window w, quadtree block b)
    M_b = Morton block representing b;
    if b is totally enclosed in w then
            /* Get all quadtree blocks of database enclosed in b */
        Compute M_bmax;
        Perform a B-tree scan for all Morton blocks M_b ≤ M ≤ M_bmax;
        Add object ids corresponding to the retrieved Morton blocks to result set;
    else    /* w overlaps b without containing it */
            /* Check if b is in database */
        Perform a B-tree scan for all Morton blocks M = M_b;
        Add object ids corresponding to the retrieved Morton blocks to result set;
            /* Recursive application of algorithm */
        Decompose b into its four children, sw, nw, se, and ne;
        for child in {sw, nw, se, ne} do
            if child overlaps w then
                WindowQuery (w, child);
            end if;
        end for;
    end if;
end WindowQuery;
```

Figure 6: The window query algorithm

quadtree block, b, in the B-tree. b is guaranteed to overlap the query window, w.

If b is totally enclosed in the query window, w, then all quadtree blocks of the database enclosed in b qualify as results of the window query operation. We retrieve all these qualifying quadtree blocks of the database in one B-tree scan. An important feature of the linear quadtree is that any quadtree blocks of the database contained in b will appear as consecutive Morton blocks in the B-tree immediately after M_b, the Morton block representing b. Thus, we compute M_{bmax}, the maximum possible Morton block totally enclosed in b. This is the Morton block representing the point in the upper right corner of b, and can easily be computed by interleaving the bits of the x and y coordinates of this point. A B-tree scan using the range predicate $M_b \leq M \leq M_{bmax}$ retrieves all quadtree blocks of the database enclosed in b, which are all results of the window query operation.

If b overlaps the query window, w, without being totally enclosed in it, we perform a B-tree scan using the equality predicate $M = M_b$ to check if b itself is in the database. However, we cannot retrieve the quadtree blocks of the database contained in b because they do not necessarily overlap the query window. Instead, we decompose b into its four children (quadtree sub-blocks), and recursively call the algorithm for the children that overlap the query window.

The order of the recursive calls to the algorithm should be the same as the Morton order. This causes the algorithm to generate quadtree blocks in increasing order of Morton blocks. The B-tree scans for these Morton blocks will traverse the B-tree in a left-to-right order. Contrary to commonly used buffer management strategies, the best buffer management strategy for traversing a B-tree left-to-right is *last-in first-out* [CD85].

While the algorithm in [AS97] minimizes the number of I/O's required to process a window query, it only works for databases where the spatial objects do not overlap. The algorithm presented here is more general because it can handle databases where the spatial objects *do* overlap.

Our algorithm is similar to the "spatial filter" algorithm presented in [OM88] in that we make

use of the ability of a B-tree to support range predicates in addition to equality predicates. A B-tree scan can return all entries satisfying a range predicate by doing a search in the B-tree followed by a sequential scan of the leaf level. This ability is used in [OM88] to skip ahead in the B-tree leaf level past quadtree blocks of the database that are guaranteed not to overlap the query window. In our algorithm, we use this ability to retrieve all quadtree blocks of the database that are guaranteed to overlap the query window in one shot. Our algorithm is much simpler than the spatial filter algorithm in [OM88], and it accesses the B-tree using equality and range predicates, which perfectly match the interface offered by B-trees. Unlike the spatial filter algorithm, our algorithm traverses the B-tree left-to-right, which is very useful for minimizing I/O's by using the last-in first-out buffer management strategy.

5 A Cost Model for the Window Query Algorithm

For our window query algorithm to be fully integrated in a database system, the query optimizer must be able to estimate its cost. In this section, we present a cost model that estimates the I/O cost of processing a window query using our algorithm.

The cost model computes two measures of the I/O cost of a window query in a linear quadtree: the number of B-tree scans, S, and the number of visits to B-tree nodes, V. The parameters required to estimate these two measures are the number of leaf nodes of the B-tree, N_{leaf}, and the height (number of levels) of the B-tree, h_B. These two parameters can easily be maintained in the database catalog.

The cost model is based on an algorithm that recursively decomposes the space into quadtree blocks in exactly the same way as the window query algorithm. This cost estimation algorithm is given in Figure 7. The algorithm has three parameters: the query window, w, the current quadtree block in the decomposition, b, and the level in the quadtree of b, l. Recall that l is the number of times the space has to be decomposed to get b. The initial call to the cost estimation algorithm is WindowQueryCostEstimate (w, r, 0), where w is the query window and r is the root of the quadtree. The number of B-tree scans, S, and the number of visits to B-tree nodes, V, are global to the cost estimation algorithm and are initialized to 0 before the first call to the algorithm.

The window query processing algorithm recursively decomposes the space into quadtree blocks until it encounters quadtree blocks that are totally enclosed in the query window. The cost estimation algorithm emulates the window query algorithm by decomposing the space into quadtree blocks in exactly the same manner using a similar sequence of recursive calls.

The cost estimation algorithm increments the number of B-tree scans, S, by 1 every time it is recursively called. Since the sequence of recursive calls to the cost estimation algorithm matches the sequence of recursive calls to the window query algorithm, and since every recursive call to the window query algorithm results in exactly one B-tree scan, the cost estimation algorithm computes an *exact* value for the number of B-tree scans, S.

The cost estimation algorithm also adds h_B, the height of the B-tree, to the number of B-tree nodes visited, V, each time it is recursively called. Every B-tree scan descends down the tree from the root to the leaf level visiting exactly one node from every level, for a total of h_B nodes. A B-tree scan may also visit some *additional* nodes at the leaf level in order to return *all* B-tree entries satisfying the scan predicate.

If the current quadtree block, b, overlaps the query window without being totally enclosed in it, the window query algorithm performs a B-tree scan using an equality predicate, $M = M_b$. In this case, the cost model assumes that all B-tree entries satisfying the scan predicate are in the same leaf node. This means that no additional nodes are visited at the B-tree leaf level, so no more nodes are

```
algorithm WindowQueryCostEstimate (window w, quadtree block b, integer l)
    S = S + 1;
    V = V + h_B;
    if b is totally enclosed in w then
            /* Estimate the number of additional B-tree leaf nodes visited */
        V = V + ⌊N_leaf / 4^l⌋ ;
    else    /* w overlaps b without containing it */
            /* Recursive application of algorithm */
        Decompose b into its four children, sw, nw, se, and ne;
        for child in {sw, nw, se, ne} do
            if child overlaps w then
                WindowQueryCostEstimate (w, child, l + 1);
            end if;
        end for;
    end if;
end WindowQueryCostEstimate;
```

Figure 7: A cost model for window queries

added to V. The cost estimation algorithm decomposes b into its four children (quadtree sub-blocks) and calls itself recursively for the children overlapping the query window. Note that the children of quadtree block b are at quadtree level $l + 1$.

If the current quadtree block, b, is totally enclosed in the query window, the window query algorithm performs a B-tree scan using a range predicate, $M_b \leq M \leq M_{bmax}$. In this case, the scan is very likely to visit additional nodes at the B-tree leaf level.

To estimate the number of additional B-tree leaf nodes visited, recall that the Morton blocks stored in the B-tree represent quadtree blocks in a two-dimensional space. The scan predicate, $M_b \leq M \leq M_{bmax}$, retrieves all quadtree blocks in the B-tree contained in the area of this two-dimensional space covered by quadtree block b. Since b is obtained by repeatedly decomposing the space into four quadrants, its area is $\frac{1}{4^l}$ of the area of the two-dimensional space, where l is the quadtree level of b. We assume that the quadtree blocks of the database, which are stored in the B-tree, are uniformly distributed in the two-dimensional space. Therefore, the scan will retrieve $\frac{1}{4^l}$ of these quadtree blocks. Retrieving these quadtree blocks requires visiting $\left\lceil \frac{N_{leaf}}{4^l} \right\rceil$ B-tree leaf nodes. The number of *additional* leaf nodes visited by the B-tree scan is therefore

$$\left\lceil \frac{N_{leaf}}{4^l} \right\rceil - 1 = \left\lfloor \frac{N_{leaf}}{4^l} \right\rfloor$$

This assumes that $\frac{N_{leaf}}{4^l}$ has a fractional part, which we can generally expect to be true.

Thus, when the current quadtree block, b, is totally enclosed in the query window, the cost estimation algorithm adds $\left\lfloor \frac{N_{leaf}}{4^l} \right\rfloor$ to the number of B-tree nodes visited, V, to account for the additional leaf nodes visited to retrieve all entries satisfying the scan predicate.

Our cost model focuses on the I/O cost of using a linear quadtree and ignores the CPU cost. For window queries on databases containing complex spatial objects, the CPU cost of the exact geometry overlap test may not be negligible [AN00]. However, this CPU cost is not related to the cost of using a linear quadtree. A linear quadtree is used for the filtering step in a *filter and refine* query processing strategy. The linear quadtree identifies a set of candidate objects that *potentially* qualify as result

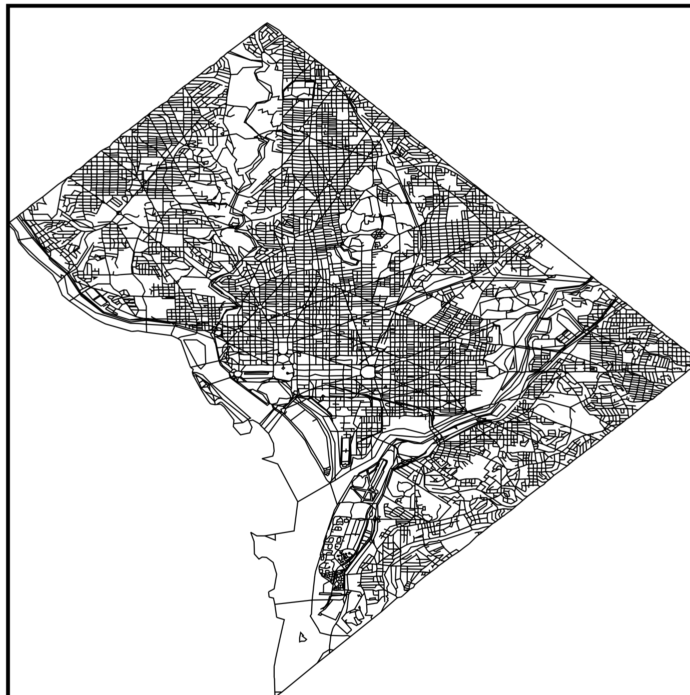


Figure 8: The DC data set

objects. The CPU cost of this step is very low. In the refinement step, the exact geometry of these candidate objects is tested to see which of them actually overlap the query window [Ore86]. This CPU intensive exact geometry overlap test is completely independent of the linear quadtree. Thus, while CPU cost is important in estimating the overall cost of a window query, it can safely be ignored when estimating the cost of using a linear quadtree.

6 Experimental Evaluation

In this section, we experimentally verify the accuracy of our cost model. A more comprehensive experimental evaluation can be found in [Abo96]. Recall that the number of B-tree scans, S, is computed exactly by our cost model. The goal of this section is therefore to verify the accuracy of the cost model only in estimating the number of B-tree nodes visited, V.

The experiments in this section use two real data sets and one synthetic data set. The real data sets consist of road maps of Washington, DC and Baltimore obtained from the U.S. Bureau of Census TIGER/Line database (http://www.census.gov/ftp/pub/geo/www/tiger/). Figure 8 shows the DC data set. We construct the minimum bounding rectangle of each road and normalize these rectangles to a 4096×4096 space. This results in $19,896$ and $11,165$ rectangles for the Baltimore and DC data sets, respectively. The synthetic data set consists of $15,000$ rectangles of area 1024 uniformly distributed in a 4096×4096 space.

The linear quadtrees for these data sets are built by decomposing each data rectangle into at most 50 quadtree blocks. The quadtree blocks obtained by this decomposition are an approximate representation of the data rectangles. These quadtree blocks are stored as Morton blocks in B-trees with 50 elements per node. The number of elements per B-tree node is set artificially low to

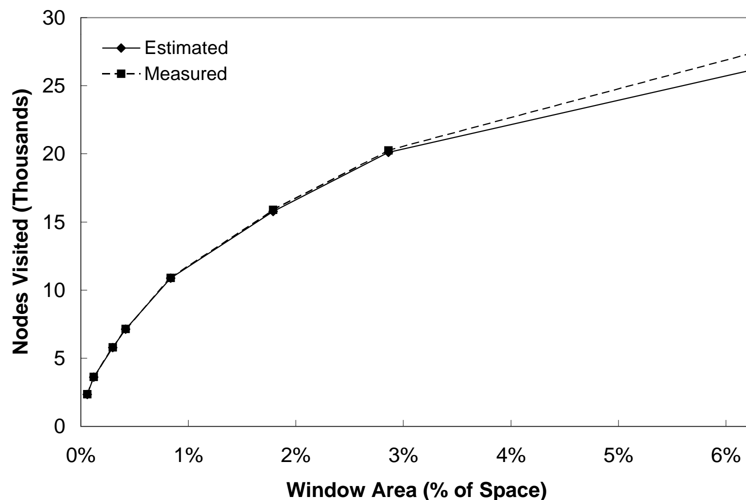


Figure 9: Accuracy of the cost model for the DC data set

correspond to the relatively small number of objects in our data sets.

To verify the accuracy of the cost model, we generate rectangular query windows of 8 different sizes (areas), ranging from 0.06% to 6.25% of the area of the space. We generate 20 different query windows of each size and use each window to query the linear quadtree using our window query algorithm. We count the number of B-tree nodes visited by the algorithm for each window query. For each window query, we also estimate the number of B-tree nodes visited using our cost model.

Figures 9–11 show the average estimated and measured number of B-tree nodes visited at each query window size for the three data sets used. The figures clearly show that the number of nodes visited is very accurately estimated by our cost model.

7 Conclusions

In this paper, we presented an algorithm for window queries in linear quadtrees. The algorithm is based on recursively decomposing the space into quadtree blocks and searching for these quadtree blocks in the B-tree. The algorithm is much simpler than previously proposed algorithms, and it works for query windows of arbitrary shape and databases with overlapping spatial objects. It uses equality and range predicates to access the B-tree, which perfectly matches the interfaces B-trees usually provide, and it traverses the B-tree left-to-right allowing us to minimize the number of I/O's by using the appropriate last-in first-out buffer management strategy.

We also present a cost model for estimating the I/O cost of our window query algorithm. The cost model accurately estimates the number of B-tree scans and the number of visits to B-tree nodes required to process a window query. It is based on a recursive decomposition of the space just like the window query algorithm, and it uses two parameters that are easily maintained in the database catalog. The accuracy and simplicity of the cost model make it very useful for query optimization.

References

[Abo96] Ashraf Aboulnaga. An algorithm and a cost model for window queries in spatial databases. Master's thesis, Faculty of Engineering, Alexandria University, Alexandria,

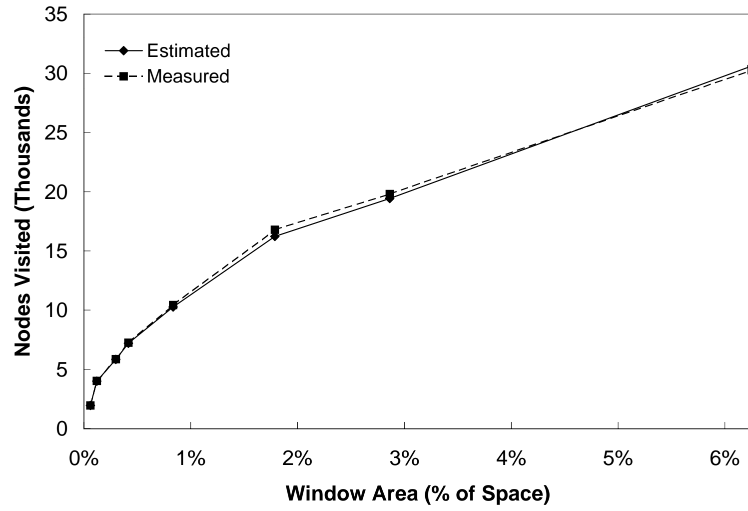


Figure 10: Accuracy of the cost model for the Baltimore data set

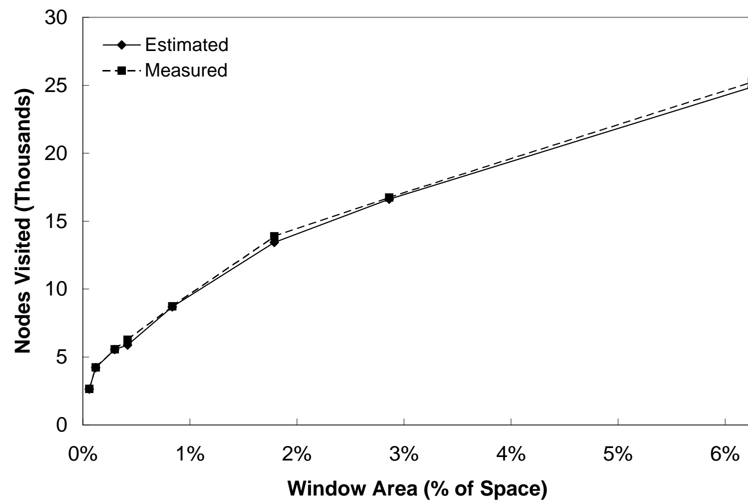


Figure 11: Accuracy of the cost model for the synthetic data set

Egypt, 1996.

[AM90] David J. Abel and David M. Mark. A comparative analysis of some two-dimensional orderings. *Int. Journal of Geographical Information Systems*, 4(1):21–31, 1990.

[AN00] Ashraf Aboulnaga and Jeffrey F. Naughton. Accurate estimation of the cost of spatial selections. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 123–134, San Diego, California, March 2000.

[Aok99] Paul Aoki. How to avoid building DataBlades that know the value of everything and the cost of nothing. In *Proc. Int. Conf. on Scientific and Statistical Database Management*, Cleveland, Ohio, July 1999.

[APR99] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, June 1999.

[AS97] Walid G. Aref and Hanan Samet. Efficient window block retrieval in quadtree-based spatial databases. *GeoInformatica*, 1(1):59–91, April 1997.

[CD85] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. Int. Conf. on Very Large Data Bases*, pages 127–141, Stockholm, Sweden, August 1985.

[Com79] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[Fal88] Christos Faloutsos. Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, October 1988.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 4–13, Minneapolis, Minnesota, May 1994.

[FR89] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 247–252, Philadelphia, Pennsylvania, March 1989.

[Gar82] Irene Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[GG98] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–54, June 1984.

[Jag90] H.V. Jagadish. Linear clustering of objects with multiple attributes. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 332–342, 1990.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *Proc. 2nd Int. Conference on Information and Knowledge Management*, pages 490–499, Washington, DC, November 1993.

[Mor66] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[OM88] Jack A. Orenstein and Frank A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14(5):611–629, May 1988.

[Ora99] Oracle Corporation. *Oracle Spatial User's Guide and Reference*, 1999. Available from http://technet.oracle.com/products/spatial/.

[Ore86] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 326–336, May 1986.

[Ore89] Jack A. Orenstein. Redundancy in spatial databases. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 294–305, June 1989.

[PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 214–221, Washington, DC, May 1993.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[Sam90a] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, Massachusetts, 1990.

[Sam90b] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[SKT+00] Alexander S. Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, Don Slutz, and Robert J. Brunner. Designing and mining multi-terabyte astronomy archives: The Sloan Digital Sky Survey. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 451–462, Dallas, Texas, May 2000.

[ST85] Hanan Samet and Markku Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(2):229–240, March 1985.

[TS96] Yannis Theodoridis and Timos Sellis. A model for the prediction of R-tree performance. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 161–171, Montreal, Canada, June 1996.