Accurate Estimation of the Cost of Spatial Selections

Ashraf Aboulnaga
Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI 53706
{ashraf,naughton}@cs.wisc.edu

Abstract

Optimizing queries that involve operations on spatial data requires estimating the selectivity and cost of these operations. In this paper, we focus on estimating the cost of spatial selections, or window queries, where the query windows and data objects are general polygons. Cost estimation techniques previously proposed in the literature only handle rectangular query windows over rectangular data objects, thus ignoring the very significant cost of exact geometry comparison (the refinement step in a "filter and refine" query processing strategy). The cost of the exact geometry comparison depends on the selectivity of the filtering step and the average number of vertices in the candidate objects identified by this step. In this paper, we introduce a new type of histogram for spatial data that captures the complexity and size of the spatial objects as well as their location. Capturing these attributes makes this type of histogram useful for accurate estimation, as we experimentally demonstrate. We also investigate sampling-based estimation approaches. Sampling can yield better selectivity estimates than histograms for polygon data, but at the high cost of performing exact geometry comparisons for all the sampled objects.

June 15, 1999

1 Introduction

For a database system to fully support spatial data, it must be able to optimize queries involving this data. This requires the query optimizer to estimate the selectivity and cost of spatial operations. In this paper, we focus on estimating the selectivity and cost of spatial selections, also known as *window queries*. In a window query, a region called the *query window* is specified, and the query retrieves all objects in the data set that overlap this region. The focus of this paper is estimating the selectivity and cost of window queries where the query windows and the underlying data objects are general polygons.

Database systems process window queries and other spatial operations using a two step *filter and refine* strategy [Ore86]. The filtering step identifies a set of *candidate objects* whose *minimum bounding rectangles* (MBRs) overlap the MBR of the query window. This set of candidates is a conservative approximation (i.e., a superset) of the result. The filtering step may use an R-tree index if one exists. The refinement step tests the exact geometry of the candidate objects identified by the filtering step to determine the set of objects that actually overlap the polygonal query window.

Several cost models for window queries have been proposed in the literature [FK94, BF95, TS96, APR99]. All these cost models assume that the query windows and the data objects are rectangles. In effect, they estimate the selectivity and cost of the filtering step and ignore the refinement step.

Ignoring the refinement step makes these cost models inaccurate for two reasons. First, the estimated selectivity of the filtering step, no matter how accurate, is only an upper bound that may significantly over-estimate the actual selectivity of the query. Second, the refinement step incurs significant costs that cannot be ignored. The refinement step involves fetching the exact geometry representation of all the candidate objects, thus incurring an I/O cost. It also involves testing these candidate objects to determine the ones that actually overlap the query window using computational geometry algorithms that have a high CPU cost[1]. An important property of the costs incurred by the refinement step is that they depend not only on the selectivity of the query, but also on the number of vertices, or *complexity*, of the query window and data objects. It has been shown that, for spatial joins, the CPU cost of the refinement step dominates the query execution cost [BKSS94, PD96]. The cost of the refinement step cannot be ignored when estimating the cost of window queries, especially since typical applications of spatial databases (e.g., GIS) involve objects with high complexities (i.e., a large number of vertices).

As an example, consider the data set of the Sequoia 2000 benchmark representing land use in the state of California [SFGM93]. The polygons in this data set have between 4 and 5583 vertices, with an average of 56 vertices. We issued 100 random window queries over this data set using the Paradise object-relational database system [P+97]. The query windows were random polygons with 20 vertices generated in rectangles that cover 1% of the space containing the data objects (see Section 5.1 for a description of how these polygons are generated). Using an R-tree index, the refinement step for the 47 queries that had a selectivity of more than 1% took, on the average, 4 times as long as the filtering step. Ignoring the refinement step when estimating costs is clearly a mistake. Accurate cost estimation should also be based on some knowledge of the underlying data distribution.

In this paper, we introduce a new type of histogram for polygon data that captures all properties of a data distribution required for estimating the cost of both the filtering and the refinement steps of spatial operations. We present a simple cost model that uses our histograms to estimate the cost of window queries where the query windows and data objects are general polygons. We also investigate the use of sampling for estimating the selectivity and cost of window queries.

The rest of this paper is organized as follows. In Section 2, we present an overview of related work.

[1] The complexity of this test is $O(n \log n)$, where n is the total number of vertices in the polygons being tested.

In Section 3, we present a cost model for window queries. Section 4 introduces our novel approach to building histograms for spatial data. These histograms are used to estimate the parameters required by the cost model. Section 5 presents an experimental evaluation of the proposed techniques. Section 6 contains concluding remarks.

2 Related Work

Several techniques have been proposed for estimating the selectivity and cost of operations on traditional data types such as integers or strings. Techniques based on using histograms to approximate data distributions are widely used by current database systems [PIHS96]. Histograms for multi-dimensional data have also been proposed in the literature [MD88, PI97].

Another approach to selectivity estimation is sampling, which provides guaranteed error bounds at the cost of taking a sample of the data at query optimization time. A sequential sampling algorithm that guarantees an upper bound for both the estimation error and the number of samples taken is presented in [LNS90].

Traditional multi-dimensional histograms can be used for point data, but not for polygons or other spatial data types. Polygons have an *extent* in space, whereas these histograms only capture the *location* of the data. On the other hand, the same sampling approaches used for traditional data can be used for spatial data. However, dealing with spatial data increases the cost of sampling.

A cost model for window queries in R-trees is developed in [KF93], and independently in [PSTW93]. This cost model assumes that the data consists of uniformly distributed rectangles and estimates the number of disk I/Os needed to answer a given rectangular window query. The latter paper also suggests a framework for studying the cost of window queries based on a knowledge of the query and data distributions.

In [FK94] and [BF95], the authors suggest using the concept that all data sets are self-similar to a certain degree to represent the distribution of spatial data. The degree of self-similarity of a data set is represented by its *fractal dimension*. These papers present models developed based on this concept for estimating the selectivity of window queries over point data and the cost of these queries in R-trees. Using the fractal dimension is a significant departure from the uniformity assumption typically made by previous works.

Another cost model for window queries in R-trees is proposed in [TS96]. This cost model is based on the *density* of the dataset, which is the average number of data objects per point of the space. The authors propose using the density at several representative points of the space to capture non-uniformity in the data distribution.

Acharya, Poosala, and Ramaswamy [APR99] study different partitionings that could be used to build spatial histograms, and introduce a new partitioning scheme based on the novel notion of *spatial skew*. This work is closely related to ours, and a detailed comparison is given in Section 4.5.

As mentioned earlier, all these works assume that the query windows are rectangles and that the data objects are points or rectangles, thus ignoring the refinement step. Furthermore, with the exception of [APR99], these works do not present general solutions for accurately approximating spatial data distributions.

A different approach to estimating the selectivity of spatial selections is given in [Aok99]. This work assumes that an R-tree index for the spatial attribute exists, and proposes that each non-leaf entry store the number of tuples that it represents. Selectivity is estimated using a tree traversal augmented by sampling when necessary. Like other approaches, this approach ignores the refinement step. Furthermore, it requires I/O for selectivity estimation, making it a high-cost approach. The main appeal of this approach is that it works not only for spatial data, but also for any data type

Parameter	Description	Source
N	Number of pages in the relation	Catalog
T	Number of tuples in the relation	
m	Average number of entries per R-tree node	
h	Height of the R-tree	
c_{seqio}	Per page cost of sequential read	Calibration
c_{randio}	Per page cost of random read	
c_{polyio}	Per object cost of reading a polygon	
c_{vertio}	Per vertex cost of reading a polygon	
$c_{MBRtest}$	CPU cost of testing rectangle overlap	
$c_{polytest}$	Cost coefficient for testing polygon overlap	
v_q	Number of vertices in the query polygon	Given
s_{MBR}	MBR selectivity	Estimated
v_{cand}	Average number of vertices per candidate polygon	

Table 1: Parameters of the cost model and how they are obtained

indexed by a generalized search tree (GiST) [HNP95].

3 A Cost Model for Window Queries

In this section, we present a cost model for estimating the I/O and CPU costs of both the filtering and the refinement steps of a window query. The model assumes that the query window and the data objects are general polygons. The cost of the filtering step depends on whether a sequential scan or an R-tree index [Gut84] is used as the access method, and the cost of the refinement step is assumed to be independent of the access method used for filtering. The parameters used by the cost model are given in Table 1.

3.1 Filtering

3.1.1 Sequential Scan

If the input relation is accessed by a sequential scan, the I/O cost of the filtering step, CF_{IOseq}, is given by

$$CF_{IOseq} = N * c_{seqio}$$

where N is the number of pages in the relation, and c_{seqio} is the per page cost of a sequential read.

During the sequential scan, the MBRs of all tuples of the relation are tested to determine whether they overlap the query MBR. The CPU cost of this test, CF_{CPUseq}, is given by

$$CF_{CPUseq} = T * c_{MBRtest}$$

where T is the number of tuples in the relation, and $c_{MBRtest}$ is the CPU cost of testing whether two rectangles overlap.

3.1.2 R-tree Index

To estimate the cost of the filtering step if an R-tree index is used as the access method, we assume that the R-tree is "good", in the sense that retrieving the data objects that overlap the query window

MBR requires the minimum number of disk I/Os and rectangle overlap tests. We also assume that the buffer pool is managed in such a way that each required R-tree node is read from disk exactly once.

The filtering step retrieves $s_{MBR} * T$ tuples, where s_{MBR} is the *MBR selectivity* of the query, defined as the fraction of tuples in the relation identified as candidates by the filtering step. This is the fraction of tuples in the relation whose MBRs overlap the query window MBR. The assumption that the R-tree is "good" implies that the tuples retrieved by the filtering step will be in the minimum number of R-tree leaf nodes. This number can be estimated as $s_{MBR} * T/m$, where m is the average number of entries per R-tree node. Extending this argument, we can estimate the number of nodes that have to be read from the level above the leaves by $s_{MBR} * T/m^2$, from the next level up by $s_{MBR} * T/m^3$, and so on until we reach the root level, at which only 1 node has to be read. Thus, the I/O cost of this step, $CF_{IOrtree}$, is given by

$$\begin{aligned} CF_{IOrtree} &= \left(\frac{s_{MBR} * T}{m} + \frac{s_{MBR} * T}{m^2} + \cdots + \frac{s_{MBR} * T}{m^{h-1}} + 1\right) * c_{randio} \\ &= \left[\left(\frac{1}{m-1}\right)\left(1 - \frac{1}{m^{h-1}}\right) * s_{MBR} * T + 1\right] * c_{randio} \end{aligned}$$

where h is the height of the R-tree (number of levels including the root node), and c_{randio} is the cost per page of a random read. We assume that we will not encounter any "false hits" while searching the R-tree. This means that we do not have to read any nodes beyond those accounted for in the above formula. Notice that, for typical values of m, the number of internal R-tree nodes read will be very small.

The filtering step has to test all the entries in each R-tree node read from the disk for overlap with the query window MBR. Since each node contains, on average, m entries, the CPU cost of this step can be estimated by

$$CF_{CPUrtree} = \left[\left(\frac{1}{m-1}\right)\left(1 - \frac{1}{m^{h-1}}\right) * s_{MBR} * T + 1\right] * m * c_{MBRtest}$$

3.2 Refinement

The refinement step has to retrieve the exact representation of all the candidate polygons identified by the filtering step. We estimate the I/O cost of reading a polygon by two components. The first component is a fixed cost independent of the size of the polygon, which we call c_{polyio}. The second component is a variable cost that depends on the number of vertices of the polygon. The number of vertices of a polygon is referred to as its *complexity*. We estimate this component of the cost by $v_{cand} * c_{vertio}$, where c_{vertio} is the per vertex I/O cost of reading a polygon and v_{cand} is the average number of vertices in the candidate polygons. Thus, the I/O cost of the refinement step, CR_{IO}, can be estimated by

$$CR_{IO} = s_{MBR} * T * (c_{polyio} + v_{cand} * c_{vertio})$$

The CPU cost of the refinement step depends on the algorithm used for testing overlap. Detecting if two general polygons overlap can be done in $O(n \log n)$ using a plane sweep algorithm, where n is the total number of vertices in both polygons [dBvKOS97]. We therefore use the following formula to estimate the CPU cost of the refinement step

$$CR_{CPU} = s_{MBR} * T * (v_q + v_{cand}) \log (v_q + v_{cand}) * c_{polytest}$$

where v_q is the number of vertices in the query polygon, and $c_{polytest}$ is a proportionality constant. Database systems may use algorithms other than plane sweep to test for overlap between polygons. However, since the complexity of almost all overlap testing algorithms is a function of the number of vertices of the polygons, variations of the above formula can typically be used. Each system should replace the $n \log n$ term in the formula with the complexity of the overlap testing algorithm it uses.

3.3 Notes

- Estimating the cost of a window query does not require knowing its actual selectivity. It only requires knowing the selectivity of the filtering step, the MBR selectivity. All candidate polygons identified by the filtering step have to be tested in the refinement step, whether or not they appear in the final result of the query.

- The parameters required by the cost model are obtained from several sources (Table 1). N, T, m, and h should be available in the system catalogs. c_{seqio}, c_{randio}, c_{polyio}, c_{vertio}, $c_{MBRtest}$, and $c_{polytest}$ are calibration constants that must be provided by the system implementer at system development or installation time. These constants depend on the specific database system and its run-time environment. v_q is known at query optimization time. Finally, s_{MBR} and v_{cand} must be estimated. The next section introduces histograms that can be used to accurately estimate these two parameters.

- The cost model we have presented here is, like all estimation models used in query optimization, a simplification of reality. For example, it does not capture such things as the degree to which the system is able to overlap the CPU time of the refinement step on some polygons with the I/O time to fetch others. Certainly, variants of the equations we have given are possible, and different variants may be more accurate for different systems. Nevertheless, the key point remains that any reasonable model must involve the parameters s_{MBR} and v_{cand}.

4 SQ-histograms

In this section, we introduce a novel approach to building histograms that represent the distribution of polygon data. These histograms capture information not only about the location of the data polygons, but also about their size and complexity. We call these histograms *SQ-histograms*, for *structural quadtree histograms*, because they capture the structure of the data polygons and are based on a quadtree partitioning of the space.

In this paper, we use SQ-histograms to estimate the MBR selectivity of window queries, s_{MBR}, and the average number of vertices in candidate polygons identified by the filtering step, v_{cand}. SQ-histograms can also be used in any application that requires an approximate representation of the spatial data distribution.

SQ-histograms partition the space into possibly overlapping rectangular buckets. The partitioning is based on the object MBRs, and tries to group similar objects together in the same buckets. Each object is assigned to one histogram bucket, and each bucket stores information about the number of objects it represents, their size, and their average complexity. The data within a bucket is assumed to be uniformly distributed. SQ-histograms are built off-line as part of updating the database statistics, and they should be rebuilt periodically to ensure their accuracy in the presence of database updates.

4.1 Partitioning the Data into Buckets

The goal of partitioning the data into buckets is to have each bucket represent a "homogeneous" set of objects. This makes assuming uniformity within a bucket accurate and results in an accurate overall representation of the data distribution. Minimizing variation within a bucket is a common goal for all histogram techniques. The properties of the data that should be taken into account by the partitioning algorithm are:

- The location of the objects. A bucket should represent objects that are close to each other in the space. This minimizes the "dead space" within a bucket. Similar rules are used for histograms for traditional data.

- The size (area) of the objects. The size of the objects in a bucket determines the expected number of objects in this bucket that overlap a query window. The larger the objects in a bucket, the more likely they are to overlap a query window. Accurate estimation therefore requires that the average size of the objects in a bucket be as close as possible to the true size of these objects. This means that grouping objects with widely varying sizes in the same bucket should be avoided.

- The complexity of the objects. Estimating the cost of the refinement step requires knowing v_{cand}, the average complexity of the candidate polygons. It is therefore desirable for estimation accuracy that the average number of vertices per object in a bucket be a close approximation of the true number of vertices. Grouping objects with widely varying complexities in the same bucket should be avoided.

SQ-histograms are built using a quadtree data structure [Sam84]. A quadtree is a recursive data structure in which each node represents a rectangular region of the space. A node can have up to four children, each representing a quadrant of the region that the parent node represents. Thus, a quadtree partitions the input space into four quadrants, and allows each quadrant to be further partitioned recursively into more quadrants (Figure 1). SQ-histograms partition the space into buckets using this quadtree partitioning.

The algorithm for building an SQ-histogram starts by building a *complete quadtree* with l levels for the space containing the data, where l is a parameter of the histogram construction algorithm. The different levels of this complete quadtree represent different sized partitionings of the space. Thus, the quadtree partitions the space at several different resolutions. We use this property to separate the data polygons according to size and location. Each polygon is assigned to a quadtree node. The quadtree *level* to which a polygon is assigned is the maximum level (i.e., the furthest from the root) such that the width and height of the MBR of the polygon are less than or equal to the width and height of the quadtree nodes at this level. Informally stated, this means that the polygon "fits" in a quadtree node at this level but not at higher levels. After choosing a quadtree level for a polygon, we choose the quadtree node at this level that contains the center of the polygon MBR. The polygon is assigned to this node. Figure 1 demonstrates assigning polygons to quadtree nodes. For the purpose of illustration, the quadtree in this figure is not a complete quadtree.

After assigning all the polygons to quadtree nodes, the complete quadtree can be used as an accurate histogram. Each quadtree node represents a number of polygons with similar location and size. Polygons that are far from each other will have MBRs whose centers lie within different quadtree nodes, and polygons with widely varying sizes will be assigned to nodes at different levels of the quadtree.

The algorithm for assigning polygons to quadtree nodes does not take into account the complexity of the polygons. The algorithm assumes that polygons in the same vicinity and with similar sizes

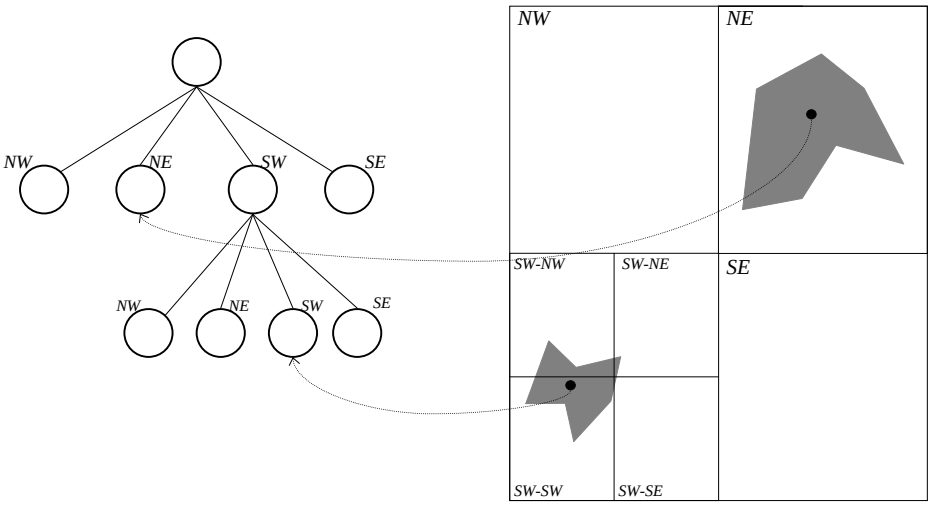


Figure 1: A quadtree partitioning and the assignment of polygons to quadtree nodes

(and therefore assigned to the same quadtree node) will have similar complexities. In the next section, we present a solution for cases in which this assumption does not hold.

The problem with the complete quadtree built by the initial phase of the SQ-histogram construction algorithm is that it may take too much memory. Database systems typically limit the amount of memory available to histograms, and an algorithm for building histograms must guarantee that they fit in the assigned memory. This constraint translates to an upper bound on the number of buckets that a histogram can have. In our case, we can reduce the number of buckets by reducing the number of levels of the complete quadtree. However, this limits the granularity at which the space is partitioned. Instead, we want to start with a complete quadtree with as many levels as we wish, but still guarantee that the final histogram will fit in the assigned memory.

To satisfy this requirement, we start with a histogram in which the buckets correspond to the non-empty nodes of the complete quadtree. We repeatedly *merge* buckets corresponding to *sibling quadtree nodes* among which the data distribution has little variation, until the number of buckets drops to the required bound. We must choose a method of measuring the *variation in data distribution* among four histogram buckets (corresponding to sibling nodes in the quadtree). For example, we could use the variance of the number of polygons represented by the buckets. We could also use the maximum difference in the number of polygons represented by any two buckets. We use this measure to compute the variation in data distribution among every set of four buckets corresponding to four sibling nodes of the quadtree. Sets of siblings nodes at all levels of the quadtree are considered in this computation. After this computation, we merge the histogram buckets corresponding to the four sibling quadtree nodes with the *least variation in data distribution.*

To merge these buckets, we replace them with one new bucket that represents all the objects that they currently represent. The new bucket represents the same region of the space that is represented by the *parent* node of the quadtree nodes corresponding to the buckets being merged. Hence, the new bucket will correspond to this parent quadtree node. If, before merging, the parent quadtree node corresponded to one histogram bucket, after merging it will correspond to *two* buckets. It is important to note that these two buckets are kept separate, even though they represent the same region of the space, because they represent objects of different sizes that were assigned to different levels of the quadtree. After an object is assigned to a quadtree level, it is only combined with other objects from the same quadtree level. This guarantees that objects in a histogram bucket always have similar sizes.

```
algorithm BuildSQ-histogram
    Build a complete quadtree of height l representing the data space;
    Scan the data and assign polygons to quadtree nodes according to size and location;
    Set the histogram buckets to correspond to the quadtree nodes that contain data;
    while Current number of buckets > Required number of buckets do
        Merge the buckets corresponding to sibling quadtree nodes
            that have the minimum variation in data distribution;
    end while;
end BuildSQ-histogram;
```

Figure 2: Algorithm for building SQ-histograms

The merging operation is repeated as many times as needed to bring the number of histogram buckets down to the required number. At each merging step, we compute the variation in data distribution among buckets that correspond to sibling quadtree nodes *and that contain objects assigned to the same quadtree level*. We always merge the buckets with the minimum variation[2]. Since we only merge buckets corresponding to sibling quadtree nodes, the partitioning of the space always remains a quadtree partitioning, and the same merging procedure can be repeated as many times as needed.

After choosing the histogram buckets, the boundaries of each bucket are set to the MBR of all the objects that it represents. This step is required because polygons can extend beyond the boundaries of the quadtree nodes to which they are assigned. It produces histogram buckets that may represent overlapping regions of the space. After this step, the regions represented by the histogram buckets no longer correspond exactly to the regions represented by the quadtree nodes. Thus, the quadtree cannot be used as an index to search for buckets that overlap a given query window. We use the quadtree only to build the SQ-histogram, not to search it at cost estimation time. At cost estimation time, a sequential search is used to determine the buckets that overlap the query window. Figure 2 presents an outline of the algorithm for building SQ-histograms. Note that this algorithm requires only one scan of the data.

4.2 Handling Objects With Varying Complexities

The above algorithm does not take into account the complexity of the polygons when creating the histogram buckets. To handle data sets in which polygons with similar sizes and locations may have widely varying complexities, we should build not one but several quadtrees, one for "low complexity" objects, one for "medium complexity" objects, and so on.

To build an SQ-histogram using this approach, we determine the minimum and maximum number of vertices of all the polygons in the data set. This requires an extra scan of the data. We also specify the number of quadtrees to build as a parameter to the algorithm. The range of vertices in the data set is divided into sub-ranges of equal width, where the number of sub-ranges is equal to the required number of quadtrees. Each quadtree represents all the objects with a number of vertices in one of these sub-ranges. Next, we build the required number of complete quadtrees, and assign the data objects to quadtree nodes according to location, size, and complexity. We decide the quadtree to which an object is assigned based on the number of vertices in this object. As before, we start with buckets corresponding to all non-empty nodes in all quadtrees, and we repeatedly merge until we get the required number of buckets. When merging, we only merge buckets corresponding to sibling

[2]Repeatedly choosing the buckets with the least variation can be done in $O(n \log n)$ using a priority queue.

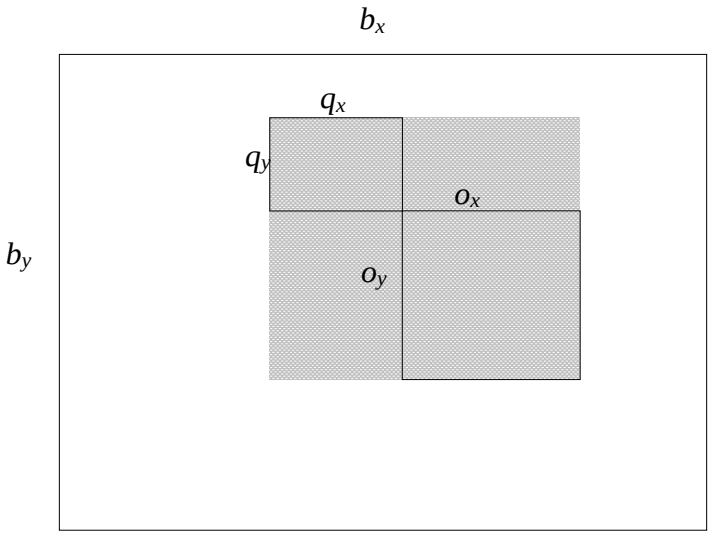


Figure 3: The MBRs of a query and an object within a bucket

nodes of the same quadtree. We merge the buckets with the least variation in data distribution among all sibling nodes of all quadtrees.

4.3 Assuming Uniformity Within a Bucket

In estimating the parameters required by the window query cost model, s_{MBR} and v_{cand}, we assume a uniform distribution within the histogram buckets. Figure 3 presents the MBR of a query window, q, and the MBR of an object, o, in a bucket, b. q overlaps o only if the upper left corner of q lies in the shaded region. Hence, under the uniformity assumption, the probability of a query window MBR, q, overlapping an object, o, in a bucket, b, is given by $\frac{(q_x+o_x)*(q_y+o_y)}{b_x*b_y}$, where q_x, q_y, o_x, o_y, b_x, and b_y are the width and height of q, o, and b, respectively [AS94]. This formula ignores the fact that the query window may extend beyond the bounds of the bucket. To partly take this possibility into account, we modify the previous formula to $\frac{\min[(qb_x+o_x),b_x]*\min[(qb_y+o_y),b_y]}{b_x*b_y}$, where qb_x and qb_y are the width and height of the rectangular overlap region between q and b.

Accordingly, we estimate f_i, the fraction of the objects in bucket i that appear in the result of the filtering step of a given window query, by the formula

$$f_i = \frac{\min[(q_{x_i} + MBR_{x_i}), b_{x_i}] * \min[(q_{y_i} + MBR_{y_i}), b_{y_i}]}{b_{x_i} * b_{y_i}} \quad (1)$$

where q_{x_i} and q_{y_i} are the width and height of the rectangular overlap region between the query window MBR and the bucket, MBR_{x_i} and MBR_{y_i} are the average width and height of the MBRs of the objects represented by this bucket, and b_{x_i} and b_{y_i} are the width and height of the bucket.

Hence, each histogram bucket must store the number of objects that it represents, the average width and height of the MBRs of these objects, and the average number of vertices in these objects. Each bucket must also store the coordinates of the lower left and upper right corners of the region of the space that it represents.

4.4 Estimation Using SQ-histograms

To estimate s_{MBR} and v_{cand}, we use a sequential search of the histogram buckets to identify the buckets that overlap the MBR of the query window. We estimate the required quantities using the

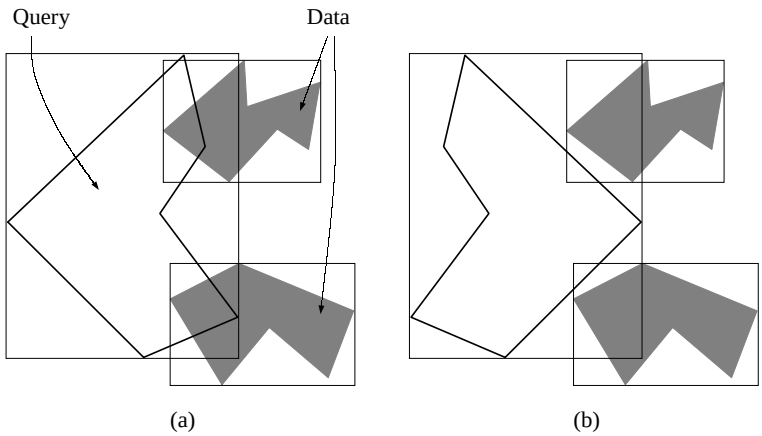


Figure 4: The difficulty of estimating actual selectivities. The query polygon overlaps the data polygons in (a) but not in (b).

following formulas:

$$s_{MBR} = \sum_{i \in B} f_i N_i$$

$$v_{cand} = \frac{\sum_{i \in B} f_i N_i V_i}{s_{MBR}}$$

where B is the set of indices in the histogram of buckets overlapping the query window MBR, N_i is the number of objects in bucket i, and V_i is the average number of vertices per object in bucket i.

SQ-histograms provide an estimate for the MBR selectivity of window queries, but not for their actual selectivity. We do not attempt to estimate the actual selectivity of window queries, as this would require information about the exact layout of the vertices of the query and data polygons. One cannot estimate whether or not two general polygons overlap based only on their MBRs, areas, or number of vertices. To demonstrate this, consider the two cases presented in Figure 4. The first case is a query polygon that overlaps two data polygons. The polygons in the second case are identical to the ones in the first case, except that the query polygon is flipped vertically. In the second case, the query polygon does not overlap either of the data polygons, despite the MBRs, areas, shapes and number of vertices being the same as in the first case.

The query optimizer can use the MBR selectivity estimated using an SQ-histogram as an upper bound on the actual selectivity of the query. This estimate may prove to be useful, especially since it is based on an accurate representation of the data distribution. Alternately, the actual selectivity of the query can be estimated using sampling (Section 5.6).

4.5 Comparison with MinSkew Partitioning

Acharya, Poosala, and Ramaswamy recently proposed a partitioning scheme for building histograms for spatial data called *MinSkew partitioning* [APR99]. Like SQ-histograms, MinSkew partitioning is based on the MBRs of the data objects. Partitioning starts by building a uniform grid that covers the input space and determining the number of objects that overlap each grid cell. This grid is an approximation of the data distribution, and is used by MinSkew partitioning to construct the histogram. During construction, the algorithm maintains a set of buckets currently in the histogram. Initially, this set contains one bucket representing the whole space. The algorithm repeatedly chooses a bucket from this set and splits it into two buckets until the number of buckets in the histogram

reaches the required number. The bucket to split and the split point are chosen to give the maximum reduction in *spatial skew*. Spatial skew is defined as the variance of the number of objects in the grid cells constituting the buckets. The algorithm considers the space at multiple resolutions by building several grids at different resolutions and generating an equal number of histogram buckets from each grid. To reduce computation time, the splitting decision is based on the marginal frequency distributions of the grid cells in the buckets.

Both MinSkew partitioning and SQ-histograms have to choose a partitioning of the space from an intractably large number of possibilities. SQ-histograms deal with this problem by considering only quadtree partitionings of the space. MinSkew partitioning restricts itself to binary space partitionings along the grid lines, which is a more general set than quadtree partitionings. However, MinSkew partitioning based on the marginal frequency distribution uses a one-dimensional measure of variation to construct the multi-dimensional partitioning, while SQ-histograms use a multi-dimensional measure of variation.

A key advantage of SQ-histograms is taking the variation in object sizes into account. MinSkew partitioning only considers the number of objects that overlap a grid cell, and not the sizes of these objects. SQ-histograms, on the other hand, assign small and large objects to different quadtree levels and thus place them in different buckets.

The most important issue in comparing SQ-histograms and MinSkew partitioning is that SQ-histograms contain information about the complexity of the objects. This information is essential for accurate cost estimation. Our experiments in the next section demonstrate that SQ-histograms are more accurate than MinSkew partitioning, even if we add the number of vertices to the information stored in the MinSkew buckets.

5 Experimental Evaluation

In this section, we present an experimental evaluation of different cost estimation techniques using polygonal window queries on real and synthetic polygon data sets. We study the performance of SQ-histograms, MinSkew partitioning, and sampling in estimating s_{MBR} and v_{cand}. Each bucket of the MinSkew partitioning stores the average complexity of the objects that it represents, in addition to the information required in [APR99]. This allows us to use MinSkew partitioning to estimate v_{cand}.

5.1 Generating Synthetic Polygons

In our experiments, we need to generate random polygons for the test queries and the synthetic data sets. To generate a polygon, we start by choosing a rectangle in the space within which the polygon is generated. This rectangle specifies the size and location of the polygon. We then choose a number of points at random inside this rectangle. These points are the vertices of the polygon. Next, we choose a random horizontal line that cuts through the rectangle, and divide the points into two groups: points that lie above this line and points that lie below it. The points in each of the groups are sorted by their x (horizontal) coordinate, and connected in the sorted order to create two "chains" of points. The leftmost and rightmost points of the two chains are moved vertically so that they lie on the splitting line. This is done to avoid generating self-intersecting polygons. Next, the two chains of points are connected at their end-points, forming a polygon. Finally, we rotate the polygon by a random angle to avoid generating polygons that are all horizontally aligned. This algorithm generates *monotone* polygons [O'R98], which are a very general class of polygons. Figure 5 gives an example of a polygon generated by this algorithm. Figure 5(a) shows the initial

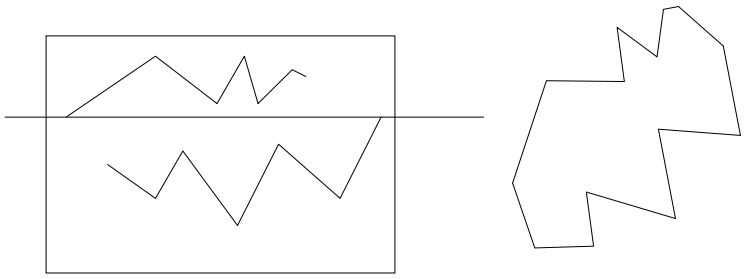


Figure 5: A synthetically generated polygon

rectangle, the split line, and the two chains of points. Figure 5(b) shows the final polygon generated by connecting the two chains and rotating.

5.2 Experimental Setup

5.2.1 Data Sets

In this paper, we present results for one real and one synthetic data set. Results on other synthetic data sets corroborate the conclusions drawn here.

The real data set we use is the set of polygons representing land use in the state of California from the Sequoia 2000 benchmark [SFGM93]. This data set consists of 58,586 polygons having between 4 and 5583 vertices, with an average of 56 vertices per polygon.

The synthetic data set we present here consists of 10,000 polygons generated using the above procedure. The polygons have between 3 and 100 vertices, with an average of 20 vertices per polygon. 30% of the polygons are distributed uniformly throughout the space, and 70% of the polygons are distributed in three clusters at different parts of the space. The rectangles in which the points of the polygons were generated have areas between 0.0025% and 0.75% of the area of the space, and aspect ratios uniformly distributed in the range 1–3.

5.2.2 Query Workloads

The number of vertices for the polygonal query windows is randomly chosen from the range 3–15. The polygons are generated inside rectangles of 9 different sizes, with areas ranging from 0.01% to 10% of the area of the space. Each workload contains 50 queries at each size, for a total of 450 queries.

When issuing a workload on some data set, we choose the centers of the rectangles in which the query polygons are generated at random from the centers of the MBRs of the data objects (i.e., the rectangles follow a distribution similar to the data [PSTW93]). We also experimented with workloads in which the queries are uniformly distributed in the space. The conclusions are the same for both types of workloads, but the query selectivities are much lower in the uniform workloads. In this paper, we use the same workload for each data set in all the experiments. For the Sequoia data set, the average selectivities of the queries of different sizes are shown in Figure 6. The figure shows both the MBR selectivity (the selectivity of the filtering step) and the actual selectivity (the selectivity of the whole query after both filtering and refinement). The selectivities are defined in the usual way as the fraction of objects in the result.

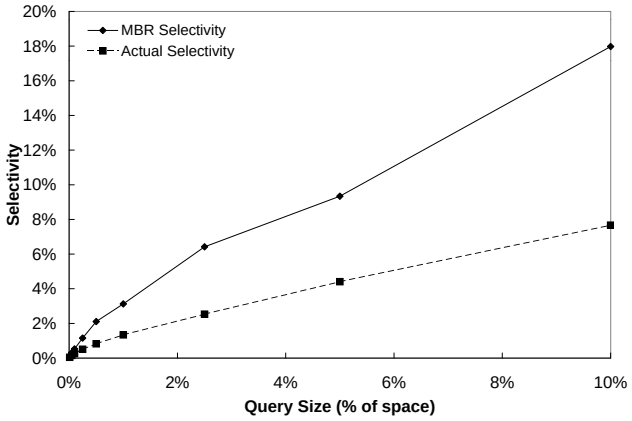


Figure 6: Query selectivity (Sequoia data set)

5.2.3 Run-time Environment

Our experiments were conducted on a Pentium Pro 200 MHz machine with 128 Mbytes of RAM running Solaris 2.6. We used one disk for the database, another disk for the log, and a third disk for the software (the database system and our test programs). Our experiments were conducted on the university version of the Paradise object-relational database system [P+97]. Both the server and the client programs were run on the same machine, and the server buffer pool size was set to 32 Mbytes.

5.2.4 Error Metric

In measuring the estimation accuracy of the various techniques, we use the *average relative estimation error* as our error metric. The relative error in estimating a quantity, x, for one query, q, is defined as

$$e_q = \frac{|\text{estimated value of } x - \text{measured value of } x|}{\text{measured value of } x}$$

For a set of M queries, the average relative estimation error is defined as

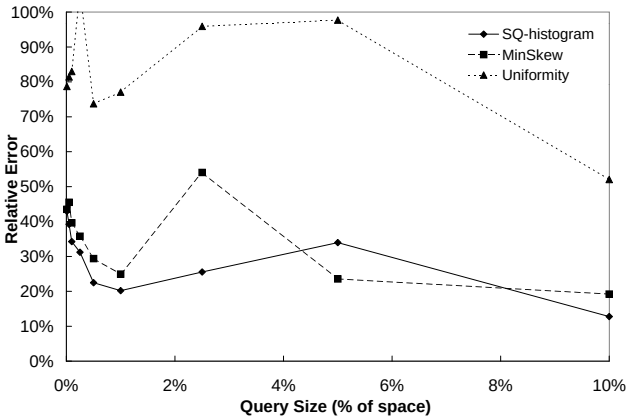
$$E = \frac{\sum_{i=1}^{M} e_i}{M}$$

Queries with a result size of zero are ignored when computing this error metric (i.e., removed from the test run). Since the query distribution is similar to the data distribution, we encounter very few queries with a result size of zero.

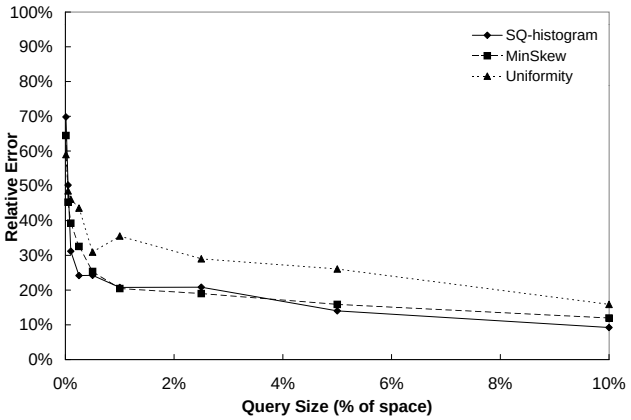
5.3 Estimation Accuracy Using SQ-histograms

In this section, we demonstrate the accuracy of SQ-histograms in estimating s_{MBR} and v_{cand} compared to MinSkew partitioning and assuming uniformity. We compare to MinSkew partitioning because it is identified as a winner among several techniques in [APR99]. We compare to assuming uniformity because it is the simplest approach in the absence of information about the data distribution.

The SQ-histograms are given 5 Kbytes of memory. They are built starting with 10 complete quadtrees of 8 levels each. We use 10 quadtrees to accommodate the varying complexities of the data objects. The histograms are built using the "maximum difference in the number of objects" to measure the variation in distribution among the quadtree nodes (Section 5.5 provides a detailed

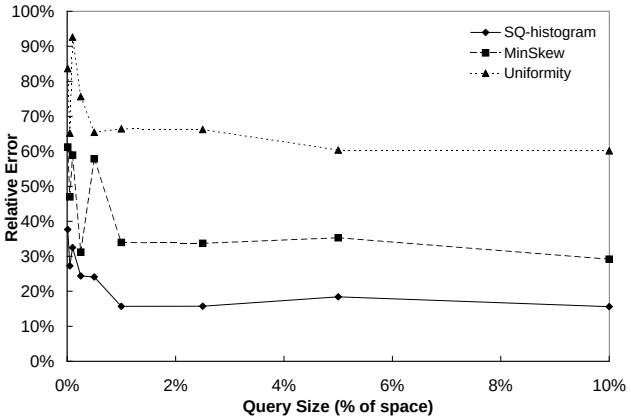


(a) s_{MBR}

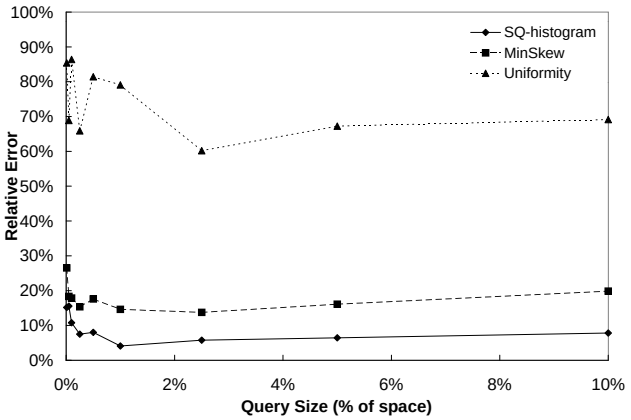


(b) v_{cand}

Figure 7: Estimation error (Sequoia data set)



(a) s_{MBR}



(b) v_{cand}

Figure 8: Estimation error (Synthetic data set)

study of the effect of the different parameters of histogram construction). MinSkew partitioning is also given 5 Kbytes of memory. We start the MinSkew partitioning with a 25×25 grid. This grid is progressively refined two times, so that the final buckets are generated from a 100×100 grid.

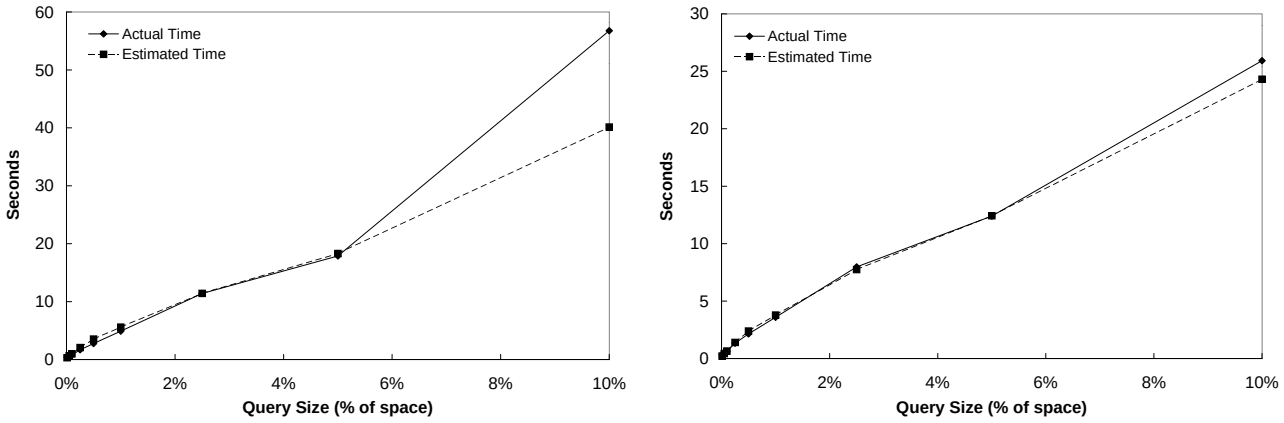
Figures 7 and 8 present the error in estimating s_{MBR} and v_{cand} for the Sequoia and synthetic data sets, respectively. Each point in the figures represents the average relative estimation error for 50 queries of a particular size. The figures show that using a histogram is always more accurate than assuming uniformity, and that SQ-histograms are generally more accurate than MinSkew partitioning. The figures also show that SQ-histograms are accurate enough in the absolute sense to be useful to a query optimizer. The irregularities in Figure 7(a) are due to a small number of queries that have estimation errors greater than 90% (fewer than five queries per test).

5.4 Accuracy of the Window Query Cost Model

Table 2 shows two sets of calibration constants for the window query cost model presented in Section 3. One set of constants is for a cold buffer pool and the other is for a warm buffer pool. These

Parameter	Cold Buffer Pool	Warm Buffer Pool
c_{seqio}	5×10^{-3}	8×10^{-4}
c_{randio}	1.5×10^{-2}	8×10^{-4}
c_{polyio}	5×10^{-7}	5×10^{-7}
c_{vertio}	2.5×10^{-5}	8×10^{-6}
$c_{MBRtest}$	0	0
$c_{polytest}$ (log base 10)	1.5×10^{-5}	1.5×10^{-5}

Table 2: Calibration constants



(a) Cold buffer pool (b) Warm buffer pool

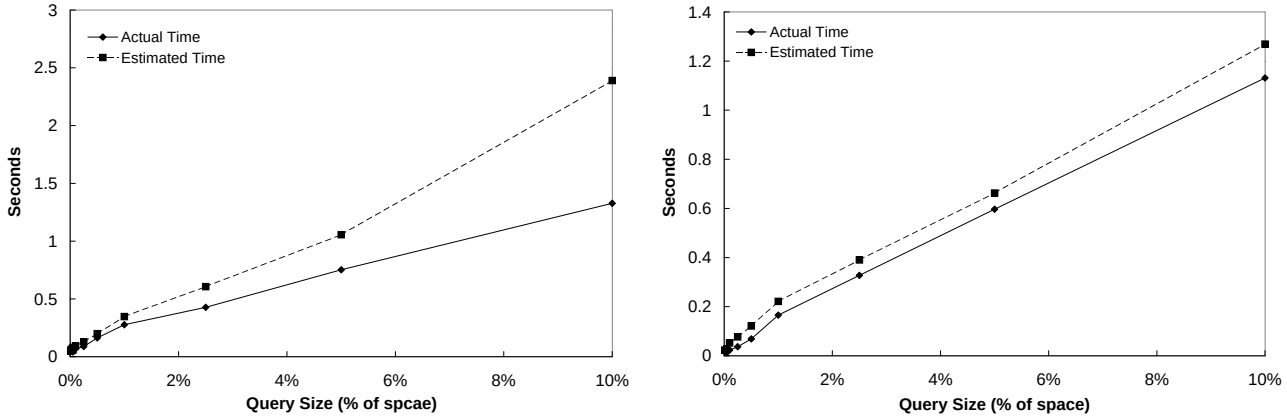
Figure 9: Execution times (Sequoia data set)

constants calibrate the cost model for use with Paradise in our run-time environment.

Figures 9 and 10 show the actual execution times of the workloads on the Sequoia and synthetic data sets, respectively. The figures show the execution times when we start with a cold buffer pool for every query (i.e., when the buffer pool is flushed between queries), and when the buffer pool is kept warm (i.e., not flushed between queries). An R-tree index is available, but the query optimizer may choose not to use it for queries with large areas and, hence, a large expected selectivity. The figures also show the estimated execution times using the calibration constants in Table 2 and with s_{MBR} and v_{cand} estimated using SQ-histograms built using the parameters described in the previous section. Each point in the figures is the average execution time for 50 queries of a particular size.

The figures show that, even with the variability in execution time, with the simplifying assumptions made by the cost model, and with the estimation errors introduced by histograms, the cost model still estimates the overall execution times of the window queries relatively accurately. While the estimated time does not, in general, match the actual time exactly, it is likely to be good enough for query optimization.

The cost model is more accurate for a warm buffer pool than it is for a cold buffer pool. A warm buffer pool reduces the variability in query execution time, making cost estimation easier. The cost model is also more accurate for the Sequoia data set than it is for the synthetic data set. Queries on the Sequoia data set have longer execution times, so estimation accuracy is more important for this data set. On the other hand, the short execution times of the queries on the synthetic data set make small estimation errors appear more pronounced.



(a) Cold buffer pool (b) Warm buffer pool

Figure 10: Execution times (Synthetic data set)

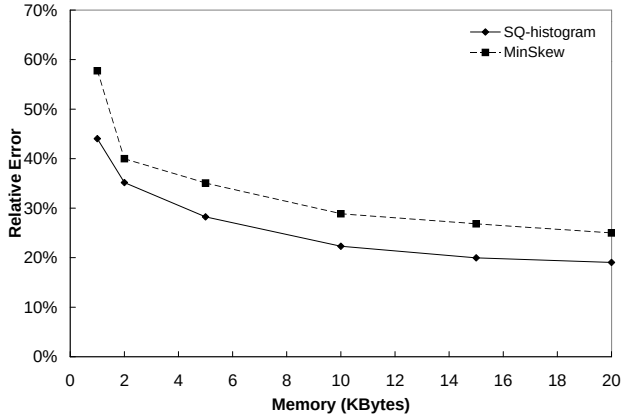


Figure 11: Effect of available memory on the accuracy of estimating s_{MBR} (Sequoia data set)

5.5 Parameters of Histogram Construction

Next, we turn our attention to the effect of the different parameters of the SQ-histogram construction algorithm. The default histogram for this experiment uses 5 Kbytes of memory, and is built starting with one 10-level quadtree using "maximum difference in the number of objects" to measure the variation in data distribution. We vary each of the histogram construction parameters in turn and show that the histogram is robust under all these variations. The errors shown in this section are average errors for all the queries of the workload.

Figure 11 shows the effect of the amount of memory available to a histogram on its accuracy. The figure shows the error in estimating s_{MBR} for the Sequoia data set using SQ-histograms and MinSkew partitioning occupying the same amount of memory. SQ-histograms are more accurate than MinSkew partitioning for the whole range of available memory. As expected, more available memory results in more estimation accuracy. Notice, though, that the error at 5 Kbytes is already reasonable, and that the slope of the error beyond this point is small.

Figure 12 shows the effect of the number of levels in the initial complete quadtree on the accuracy of SQ-histograms in estimating s_{MBR} for the Sequoia and synthetic data sets. Starting with

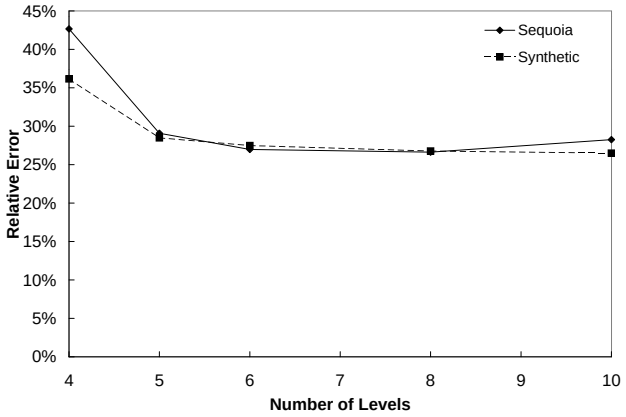


Figure 12: Effect of the number of levels in the initial quadtree on estimating s_{MBR}

more quadtree levels is generally better, as it allows the histogram to consider the space at a finer granularity. Furthermore, using more levels allows for better separation of objects according to size. However, having too many levels may actually increase the error by creating a histogram with an unnecessarily large number of small buckets. The most important observation, though, is that the error is relatively flat for a wide range of initial quadtree levels. The histogram construction algorithm is not overly sensitive to this parameter.

Next, we compare SQ-histograms constructed using different measures of variation in the data distribution. We experiment with three different measures of variation. The first is the maximum difference between the number of objects in the different buckets. The second is the maximum difference between the number of objects in the different buckets *relative to* the maximum number of objects any of the buckets. The third is the variance of the number of objects in the buckets. We also try choosing the buckets to merge based on the total number of objects in these buckets. Under this scheme, we merge the buckets in which the total number of objects is minimum. This scheme tries to construct histograms where the buckets all have the same number of objects, similar to equi-depth histograms for traditional data [PIHS96]. Figure 13 presents the error in estimating s_{MBR} using SQ-histograms constructed using the different measures of variation. Maximum difference is the winner by a tiny margin. More importantly, we notice that the histogram is robust across three of the four methods.

In the interest of space, we do not present the results for starting with different numbers of quadtrees for different object complexities. The number of quadtrees does affect histogram accuracy, but the effect is small.

5.6 Sampling

In this section, we consider sampling for selectivity estimation. Figure 14 presents the accuracy of using sampling to estimate the MBR selectivity and the actual selectivity for the Sequoia and synthetic data sets. The figure presents the errors for sample sizes of 100 and 200. Sampling is very inaccurate for queries with low selectivity because most of the samples taken are negative samples (i.e., do not satisfy the selection predicate). Thus, the figure present the average errors for all queries in the workloads with actual selectivities $> 1\%$.

Sampling is less accurate than SQ-histograms for estimating MBR selectivities. The key advantage of sampling is that, since it accesses and tests the actual data objects, it can be used to

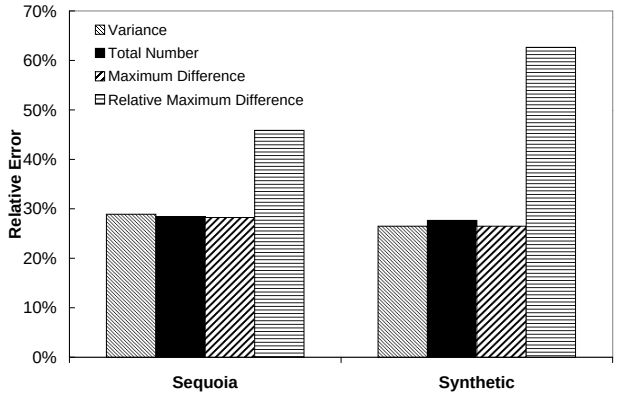
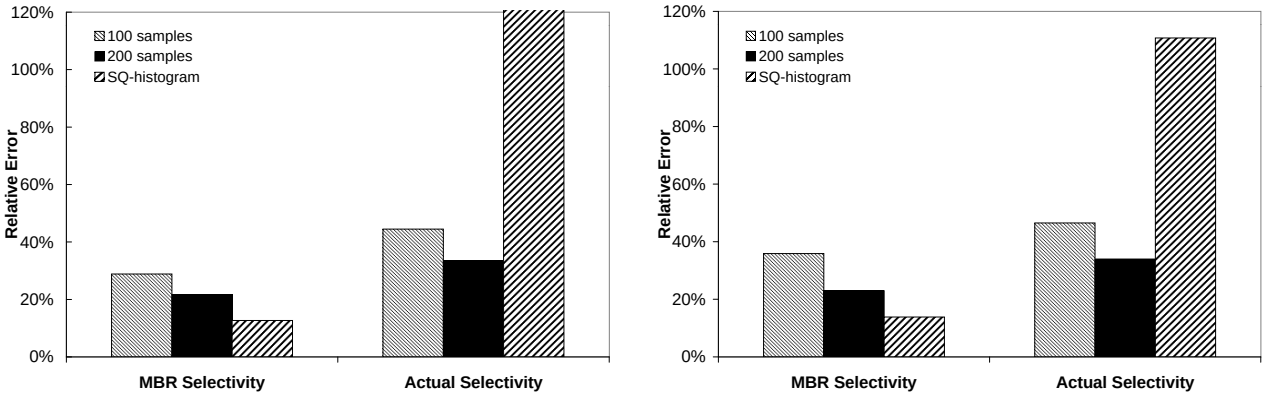


Figure 13: Effect of the measure of variation on estimating s_{MBR}



(a) Sequoia (b) Synthetic

Figure 14: Estimation using sampling

accurately estimate actual selectivities. Histograms provide only summary information that does not reflect the exact layout of the data objects, and hence cannot be used to estimate actual selectivities. Using the MBR selectivities estimated using the histograms as estimates of the actual selectivities leads to large errors (shown in the figures).

The disadvantage of sampling is its cost. Sampling involves the I/O cost of fetching the sampled tuples, as well as the high CPU cost of the exact geometry test for all the objects in the sample. In our experiments, we found that taking a positive sample of one polygon (i.e., a sample where the polygon does overlap the query window) takes up to 25 ms when all the required indexes are buffered. A negative sample can often be detected by testing the MBRs of the query and polygon. In this case, the sample usually takes less than 1 ms if the indexes are in the buffer pool. Thus, the argument that sampling is expensive, which is often made in the context of traditional data, is more pronounced in the context of spatial data.

As expected, estimation accuracy increases with increasing the number of samples. Hence, one can reduce the error as desired by increasing the number of samples.

6 Conclusions

Accurate estimation of the cost of spatial selections requires taking into account the I/O and CPU costs of the refinement step. This requires estimating the MBR selectivity of the query and the average number of vertices in the candidate polygons identified by the filtering step.

SQ-histograms effectively estimate these two quantities and can be used to provide reasonably accurate cost estimates. SQ-histograms are also robust for a wide range of histogram parameters.

Sampling can also be used to estimate these two quantities. Sampling does not work well for very selective queries. For other queries, sampling offers the additional benefit of accurately estimating the actual selectivity of the query in addition to its MBR selectivity. However, sampling from spatial databases is expensive because each sample requires an expensive polygon overlap test.

Estimating the cost of spatial operations, in general, requires information about the location, size, and complexity of the data objects. In this paper, we demonstrated how to effectively capture these properties using SQ-histograms, and how to use them for accurate estimation of the cost of spatial selections.

References

[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.

[AS94] Walid G. Aref and Hanan Samet. A cost model for query optimization using R-trees. *ACM Workshop on Advances in Geographic Information Systems*, December 1994.

[BF95] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proc. Int. Conf. on Very Large Data Bases*, pages 299–310, Zurich, Switzerland, September 1995.

[BKSS94] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 197–208, May 1994.

[dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer-Verlag, 1997.

[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.

[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.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proc. Int. Conf. on Very Large Data Bases*, pages 562–573, Zurich, Switzerland, September 1995.

[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.

[LNS90] R.J. Lipton, J.F. Naughton, and D.A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–11, May 1990.

[MD88] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 28–36, June 1988.

[O'R98] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition, 1998.

[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.

[P+97] Jignesh Patel et al. Building a scalable geo-spatial database system: Technology, implementation, and evaluation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, May 1997.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 259–270, June 1996.

[PI97] Viswanath Poosala and Yannis Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. Int. Conf. on Very Large Data Bases*, pages 486–495, Athens, Greece, August 1997.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 294–305, May 1996.

[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.

[SFGM93] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The SEQUOIA 2000 storage benchmark. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 2–11, May 1993.

[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.