Statistics Collection in Oracle Spatial and Graph: Fast Histogram Construction for Complex Geometry Objects

Bhuvan Bamba, Siva Ravada, Ying Hu and Richard Anderson
Oracle Spatial and Graph
Oracle America Inc.
{bhuvan.bamba, siva.ravada, ying.hu, richard.anderson}@oracle.com

ABSTRACT

Oracle Spatial and Graph is a geographic information system (GIS) which provides users the ability to store spatial data alongside conventional data in Oracle. As a result of the coexistence of spatial and other data, we observe a trend towards users performing increasingly complex queries which involve spatial as well as non-spatial predicates. Accurate selectivity values, especially for queries with multiple predicates requiring joins among numerous tables, are essential for the database optimizer to determine a good execution plan. For queries involving spatial predicates, this requires that reasonably accurate statistics collection has been performed on the spatial data. For *extensible data cartridges* such as Oracle Spatial and Graph, the optimizer expects to receive accurate predicate selectivity and cost values from functions implemented within the data cartridge. Although statistics collection for spatial data has been researched in academia for a few years; to the best of our knowledge, this is the first work to present spatial statistics collection implementation details for a commercial GIS database. In this paper, we describe our experiences with implementation of statistics collection methods for complex geometry objects within Oracle Spatial and Graph. Firstly, we exemplify issues with previous partitioning-based algorithms in presence of complex geometry objects and suggest enhancements which resolve the issues. Secondly, we propose a main memory implementation which not only speeds up the disk-based partitioning algorithms but also utilizes existing R-tree indexes to provide surprisingly accurate selectivity estimates. Last but not the least, we provide extensive experimental results and an example study which displays the efficacy of our approach on Oracle query performance.

1. INTRODUCTION

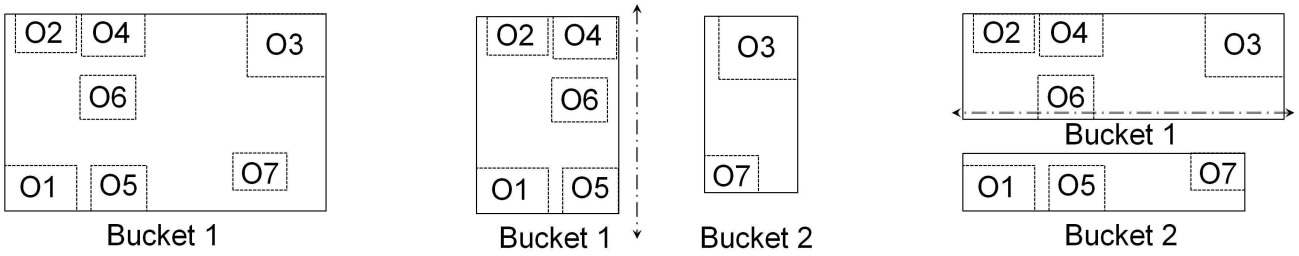
Selectivity estimation in databases is an important problem considering the impact it can have on the ability of the query optimizer to select the correct execution plan. Incorrect execution plans can adversely affect the query execution time. In the past few years, we have witnessed a trend towards storage of spatial data in databases along with other conventional data allowing users to perform complex spatial analysis. Databases such as Oracle Spatial and Graph [4], Informix Spatial Datablade [1], PostGIS [5], Microsoft SQL Server [2] allow users to leverage their spatial data in such a manner. This has also led to increasingly complex SQL queries involving spatial and non-spatial predicates possibly requiring joins among multiples tables. The Oracle optimizer is a cost-based optimizer which provides methods for accurate statistics collection as well as cost functions for single-dimensional data. However, object-relational cartridges such as Oracle Spatial and Graph are expected to provide spatial predicate selectivity values and relevant cost values back to the optimizer through an extensibility mechanism. This paper discusses our experiences with implementing this functionality for Oracle Spatial and Graph.

Statistics collection has been studied for building histograms for query optimization purposes. Sampling is used extensively for conventional data [27, 20, 19, 15, 6, 13] but has been shown to be ineffective in the spatial domain [9, 8, 7]. Sampling is based on the assumption that a small subset of data can represent the overall data distribution. This does not hold true in the spatial domain due to two factors. Firstly, individual spatial geometries differ in shape and size. Secondly, distribution of frequencies over the input domain does not vary dramatically in spatial data, whereas the values are spread non-uniformly in space [8]. For these reasons, spatial histogram construction methods must allow for the skew in size and placement of input data in the domain space to be taken into account.

Partitioning the input space into regions (or buckets) is a feasible way of constructing spatial histograms. Two issues need to be considered while constructing such histograms. Firstly, we need to determine the criteria for grouping the spatial geometries into buckets. Secondly, we need a technique for using the resulting set of buckets to estimate the selectivity values. Construction of disjoint buckets simplifies the latter step. Each bucket contains information about the number of intersecting spatial geometry objects and the average height and width of the minimum bounding rectangles (MBRs) of the spatial geometry objects within the bucket for determining the predicate selectivity value for different query geometry types.

Spatial indexing methods like the R-tree and its variants [18, 32, 10] use the MBRs of complex geometry shapes

Proceedings of the VLDB Endowment, Vol. 6, No. 11



(a) Example Input Space (b) Equi-Area Partitioning (c) Equi-Count Partitioning

Figure 1: Current Partitioning-based Techniques

for indexing purposes to speedup query evaluation using the two-step filtering approach [11]. Oracle Spatial and Graph uses a variant of the R-tree index for indexing of spatial data [24, 26]. For very large datasets, consisting of billions of rows, the index creation process may take up to a few hours. Given this fact, we aim to effectively utilize the spatial index for histogram construction. We also need to consider the fact that although many of our customers use engineered systems like Exadata [3] which have terabytes of main memory available, a large number of customers still use conventional servers with gigabytes (or tens of gigabytes) of main memory available across a set of distributed machines. Hundreds of users may simultaneously issue statistics collection SQL queries in a database. Each query may be constructing histograms for spatial data tables with potentially billions of rows. In such a scenario, partitioning-based algorithms need to read/write data from/to the disk multiple times during the bucket splitting process as limited amount of memory is available for each SQL query. Such disk-based algorithms slow down the statistics collection process. We utilize the R-tree index to implement a main memory-based algorithm which speeds up the statistics collection time. Surprisingly enough, when combined with our partitioning-based mechanisms, the main memory implementation provides better selectivity estimates than the slower disk-based approaches.

Spatial histogram construction has been extensively researched in academia for some time; to the best of our knowledge, this is the first work which discusses implementation of statistics collection methods in a commercial GIS database. With the above discussion in mind, we outline the contributions of this work as follows:

- We discuss prevalent issues with existing partitioning-based spatial histogram construction methods which lead to inaccurate statistics at best or fail to collect statistics in the worst case scenario. We propose a modified algorithm to fix the encountered issues.

- We propose a main memory statistics collection algorithm for performing fast statistics collection by utilizing the higher level nodes of our R-tree index.

- A detailed experimental evaluation with real world datasets shows that our statistics collection methods lead to determination of accurate selectivity values for different query geometry types. We evaluate the different proposals for statistics collection and outline a strategy for effective statistics collection in Oracle Spatial and Graph for database administrators.

- We provide an example study utilizing the Oracle *bitmap conversion* operation to display the efficacy of our framework in Oracle.

The rest of this paper is outlined as follows. We provide the background and motivation for this work in section 2 by discussing the issues encountered with current partitioning-based spatial histogram construction schemes. This is followed by a description of our proposed algorithms in section 3. We perform an extensive experimental evaluation of our algorithms using real world datasets in section 4 along with an example study in Oracle. In section 5, we provide a brief overview of the current state of art methods for spatial histogram construction. Finally, we conclude in section 6 with a brief summary of our experiences.

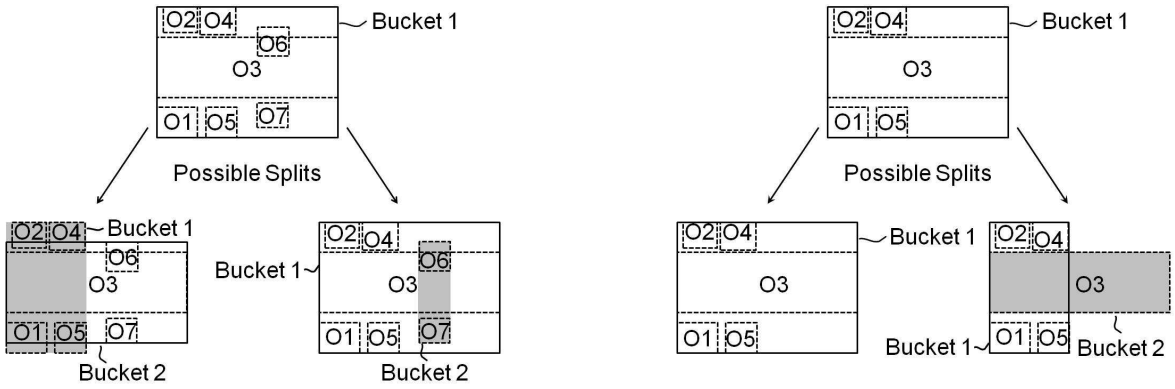
2. BACKGROUND AND MOTIVATION

2.1 Spatial Query Execution Plans

The main motivation behind the work is to provide accurate spatial predicate selectivity and cost values to the Oracle optimizer. In the absence of accurate statistics, the optimizer does not provide the best execution plan for queries involving spatial predicates. The default approach is to guide the optimizer towards using the spatial domain index when spatial predicates are encountered. This may not be the best approach for queries where spatial predicate selectivity is high. Further issues are encountered, as the optimizer, being a cost-based optimizer, still does not necessarily choose the spatial index over full table scans in the absence of meaningful statistics. In many cases, we encountered plans which were significantly worse as they involved joins between numerous tables some of which contained spatial data. In such situations, it is important to have accurate predicate selectivity values as this determines the appropriate join order for the tables.

2.2 Partitioning-based Histogram Construction

Previous work considers the problem of constructing histograms for spatial data using equi-area or equi-count heuristics. The goal of each technique is to split the input space into buckets (or regions) which have the same area or the same number of data objects. [8] describes these techniques for point and range queries over two-dimensional rectangular data. The histogram creation process is initiated by constructing a first bucket which encloses the MBRs of all spatial geometry objects. Further buckets are constructed by splitting an existing bucket into two smaller buckets, using



(a) Creation of buckets with unnecessarily large MBRs

(b) Creation of buckets with too few objects

Figure 2: Issues with Current Partitioning-based Techniques

either the equi-count or equi-area heuristic, until the desired number of buckets is obtained. Each bucket is defined by its MBR which encloses all spatial objects contained within.

Figure 1 provides an example for equi-area and equi-count heuristic-based splitting. The input space of spatial geometries comprises of the MBRs of objects *O1* to *O7* (shown with dotted lines) as shown in figure 1(a). The first bucket is formed by computing the MBR which encloses all the spatial geometry MBRs as shown using the solid rectangle labeled *Bucket 1* in the figure. The goal of any partitioning-based scheme is to split this initial bucket into the required number of final buckets iteratively using a heuristic. The equi-area heuristic aims to create buckets whose MBRs have equal areas. The MBR of a bucket is split along the longer dimension into two equal halves and the MBRs are grouped into either half depending on where their centers lie. Figure 1(b) shows the bucket in figure 1(a) being split along the length (longer dimension) using the dotted vertical line at the center. On the other hand, equi-count partitioning attempts to create buckets containing the same number of MBRs to limit the worst case errors. The algorithm is similar to the equi-area partitioning with the exception that the bucket chosen for splitting along a dimension has the largest projected MBR count along that dimension. *The projected MBR count of a dimension in a bucket is the number of distinct MBR centers in the bucket when projected on that dimension.* Figure 1(c) shows the split of the input space in figure 1(a) according to the equi-count heuristic. Next, we discuss the issues encountered when applying these partitioning schemes to complex spatial geometry datasets with large spatial extent.

2.3 Issues with Existing Approaches

We observed that when the above techniques are applied to datasets containing *complex spatial objects with large spatial extent* and consequently large MBRs we may obtain histograms which provide poor selectivity estimates. The histogram construction algorithms described above may cause a bad split in presence of complex spatial objects with large spatial extents. Two main issues are encountered which are explained with the help of figure 2.

The first issue is that existing methods may lead to *creation of buckets with large MBRs.* Larger buckets lead to larger estimation errors. We explain this in further detail with the help of an example (figure 2(a)). The figure displays the MBRs (dotted lines) for seven objects *O1* to *O7* with object *O3* being much larger than the other objects. The solid line displays the MBR for *Bucket 1* which encloses the seven objects. According to equi-area heuristics, the bucket is split along the length in an attempt to create two buckets with smaller equal areas. As the center of *O3* lies exactly on the split line, this can lead to two possible splits based on which bucket object *O3* is assigned to, left or right. As can be viewed from the figure, one of the two buckets formed by splitting is as large (or almost as large) as the original bucket. If *O3* is assigned to the right bucket (bottom left of figure 2(a)), *Bucket 2* comprising of *O3*, *O6* and *O7* is almost as large as the original bucket. If *O3* is assigned to the left bucket (bottom right of figure 2(a)), then *Bucket 1* is as large as the original bucket.

Another issue encountered is the *creation of buckets with too few data objects.* Buckets with few objects will affect the selectivity estimates as other buckets are likely to have larger number of objects which adversely affects the selectivity estimates. Figure 2(b) displays an example scenario. Five objects *O1* to *O5* are shown in the figure with object *O3* being much larger than the others. Once again, two splits are possible according to equi-area heuristics based on assignment of *O3* to the left bucket or the right bucket. The bottom left of figure 2(b) shows that no split occurs if *O3* is assigned to the left bucket. Although this issue is simple to detect and overcome, an algorithm which does not account for this may get stuck in an infinite loop. If *O3* is assigned to the bucket on the right, the split occurs with *Bucket 2* comprising of a single object *O3* with a large MBR. Equi-count heuristic also leads to the issues described here. In the following section, we outline our partitioning algorithm which overcomes these issues by performing *fuzzy splitting* of large objects among multiple buckets.

3. ALGORITHMS

We use the following observation for overcoming the above mentioned weaknesses of partitioning-based schemes. In presence of complex spatial objects with large spatial extents it may be possible to use fuzzy bucket creation where

a single large geometry object may belong to multiple buckets. This avoids the problem of large MBRs as well as large number of geometries being assigned to a single bucket if the partitioning is performed in a prudent manner. The idea of fuzzy clustering has been used in the past [17, 32, 34] in various contexts, although never in the case of spatial statistics collection. We first describe the fuzzy partitioning algorithm which can incorporate both the equi-area as well as the equi-count heuristic. However, in presence of very large number of objects and limited amount of main memory, the algorithm is disk-based as we need to read/write buckets from/to the disk. In order to overcome the drawbacks, we introduce a main memory-based algorithm which uses the R-tree hierarchy to obtain the *best* set of MBRs which fit in main memory. The main memory approach outperforms the disk-based approaches in terms of speed of statistics collection as well as predicate selectivity estimation accuracy when combined with our fuzzy partitioning schemes.

3.1 Fuzzy Partitioning Algorithm

Algorithm 1: Fuzzy Histogram Construction Algorithm

Input: $ROOT\_MBR, ndim(=2), MAX\_MEM$
Output: $B_1, B_2 ... B_m$
1 $B_1 \leftarrow rTreeTraversal() \mid \{O_1 ... O_n\} \in B_1$ && $O_i = [M_i, W_i]$, $W_i = 1 \ \forall \ i$;
2 $[W_{B_1}, H_{B_1}] \leftarrow MBR(B_1)$;
3 $[W^{avg}_{B_1}, H^{avg}_{B_1}] \leftarrow Avg(M_i) \ \forall \ i$;
4 $[W^{avg}_{B}, H^{avg}_{B}] = [W_{B_1}, H_{B_1}]/pow(m, 1/ndim)$;
5 $N_{avg} = n/m$;
6 $|B| = 1$;
7 $Initialize(pq)$;
8 $pq.push(max(W_{B_1}, H_{B_1}), B_1)$ /*equi-area heuristic*/;
9 **while** $(|B| < m)$ **do**
10 $B_j \leftarrow pq.pop()$;
11 $[B_j, B_{|B|+1}] \leftarrow splitBucket(B_j)$;
12 $[W_{B_j}, H_{B_j}] \leftarrow MBR(B_j)$;
13 $[W_{B_{|B|+1}}, H_{B_{|B|+1}}] \leftarrow MBR(B_{|B|+1})$;
14 $[W^{avg}_{B_j}, H^{avg}_{B_j}] \leftarrow Avg(M_i) \ \forall \ i \in B_j$;
15 $[W^{avg}_{B_{|B|+1}}, H^{avg}_{B_{|B|+1}}] \leftarrow Avg(M_i) \ \forall \ i \in B_{|B|+1}$;
16 **if** $(\sum W_i \ (\forall \ O_i \in B_j) > x * N_{avg})$ **then**
17 $pq.push(max(W_{B_j}, H_{B_j}), B_j)$ /*equi-area heuristic*/;
18 **end**
19 **if** $(\sum W_i \ (\forall \ O_i \in B_{|B|+1}) > x * N_{avg})$ **then**
20 $pq.push(max(W_{B_{|B|+1}}, H_{B_{|B|+1}}), B_{|B|+1})$ /*equi-area heuristic*/;
21 **end**
22 $|B| + +$;
23 **end**
24 $writeToStatsTable(B_j) \ \forall \ j \in 1...m$;
25 **return** $\{B_1, B_2 ... B_m\}$;

We use an Oracle R-tree index created for a spatial data table for fast spatial histogram construction. The leaf level nodes for an Oracle R-tree contain the MBR and the row identifier for each spatial geometry object. Since the spatial geometry object MBRs stored in the R-tree leaf nodes are used for bucket creation, this ensures that no rows need to be retrieved from the corresponding Oracle table. This prevents us from retrieving a large number of data blocks from the disk. We now describe our algorithm for constructing multidimensional histograms for complex spatial geometry objects based on the pseudocode listed in algorithm 1. The algorithm accepts as input a pointer to the root node of the R-tree Index ($ROOT\_MBR$), number of dimensions of the geometry data ($ndim$) as well as the determined main memory limit available for the operation (MAX_MEM). The algorithm returns the final set of constructed buckets denoted by $B_1, B_2 ... B_m$, where m denotes the desired number of buckets, as the output. For ease of explanation we assume that $ndim = 2$ here; an extension can be easily performed for higher dimensional data. Our experimental results are based on 2D geodetic (3D in geocentric) coordinate datasets.

We first retrieve all the leaf nodes from the R-tree index for the spatial table by traversing down the index hierarchy using a breadth first approach (procedure $rTreeTraversal()$). We store the MBR M_i and weight W_i for each geometry object O_i as a *node* in a linked list (line 1). Note that weight $W_i = 1$ initially for each geometry object O_i. The set of *nodes* forms a single linked list in memory which constitutes the first bucket B_1. As soon as bucket size is larger than what the determined main memory limit MAX_MEM can accommodate, the linked list nodes are stored in a temporary Oracle table. The addition of *nodes* to the linked list involves computation of the bucket MBR. The bucket MBR, with width W_{B_1} and height H_{B_1}, simply encloses the MBRs of the leaf level node entries each of which represents the MBR for a single geometry object (line 2). Additionally, we also compute the average MBR width $W^{avg}_{B_1}$ and average MBR height $H^{avg}_{B_1}$ for spatial geometry objects belonging to the bucket (line 3).

We assume that we will obtain the desired number of buckets m by iteratively splitting this first bucket into smaller buckets. We compute the average bucket width W^{avg}_{B} and average bucket height H^{avg}_{B} for the final set of constructed buckets as a space region split of this initial bucket (line 4). We also compute the average number of objects N_{avg} in each final bucket as $N_{avg} = n/m$ (line 5). The current number of buckets $|B|$ is set to 1 (line 6). We also initialize a priority queue pq with key as the longest bucket dimension (for equi-area) or the number of objects (for equi-count) in order to determine which bucket should be split next. The first bucket is *pushed* into this queue using the appropriate key (lines 7-8).

Next, we iteratively split the bucket(s) till the desired number of buckets is obtained (lines 9-23). The split can be performed according to either equi-area or equi-count heuristic. Note that any other heuristic may be applied here. In order to prevent buckets from containing too few objects, only buckets which contain at least $x * N_{avg}$ objects $(x < 1)$ are considered for splitting. This threshold criterion prevents too many splits for a bucket which already contains a small number of objects as this bucket is expected to cause low relative errors in selectivity estimation. We empirically determined that $x = 0.3$ works well for a large number of datasets that we experimented with and we use this value for our experiments.

The bucket at the head of the priority queue pq is removed and split next (lines 10-11). The linked list which constitutes this bucket is traversed and for each *node* we obtain its MBR M_i. Remember that for large buckets we may need to read the nodes from disk. We assign the *node* to one of the two newly formed buckets based on whether the

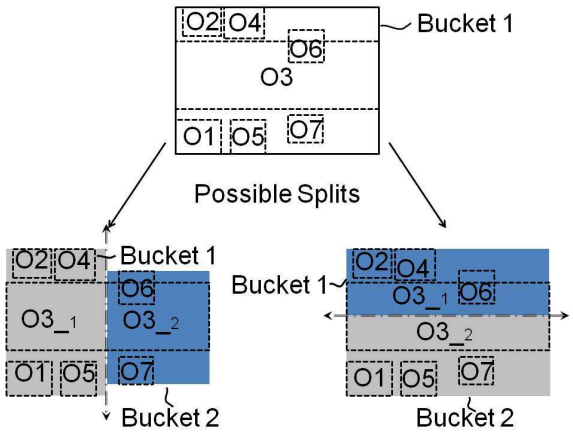


Figure 3: Fuzzy Partitioning Example

center of its MBR lies in the left or right half of the original bucket (procedure *splitBucket*()). After assignment of all the geometry objects to one of the two newly formed buckets, the bucket MBRs for the new buckets are recomputed to ensure tightness as well as the enclosure of all contained geometry MBRs. This step may lead to the formation of buckets with large MBRs if one or more contained objects have large MBRs.

In order to avoid formation of buckets with unnecessarily large MBRs, we split large geometry MBRs. We determine geometries for which $W_i > W_B^{avg}$ or $H_i > H_B^{avg}$. For such geometries, if the geometry MBR is fully contained in one of the two halves representing the newly formed buckets no additional processing is necessary. However, if the geometry MBR is spread across both buckets, we split the geometry MBR allowing it to be shared between the two buckets with weights W_{i1} and W_{i2} equivalent to the fraction of the original MBR area contained within each bucket. Note that W_{i1} and W_{i2} are the weights for the objects in the two newly formed buckets with $W_{i1} + W_{i2} = W_i$, where W_i is the weight of the geometry MBR in the parent bucket. This prevents the MBR recomputation step from resulting in large bucket MBRs.

The newly formed buckets are inserted into the priority queue if the sum of the weights of objects in the bucket meets the minimum object count requirement (lines 16-21). Once the desired number of buckets m are obtained, they are written to an Oracle table which is cached in memory for selectivity computation (procedure *writeToStatsTable*()). Each bucket B_j is classified by its bounding box $MBR(B_j)$, the average width $W_{B_j}^{avg}$ and height $H_{B_j}^{avg}$ of objects belonging to the bucket and the sum of weights of objects belonging to the bucket $\sum W_i \; \forall \; O_i \in B_j$.

For very large datasets, Oracle recommends partitioning of the datasets. For spatial data, users can create *local partitioned indexes* where a separate R-tree structure is created for each partition of the dataset. Our implementation gathers the leaf level entry MBRs for each partition in a single pass and writes them to disk. The MBRs for each partition are stored in a permanent Oracle table as any changes made to a single (or few) partitions can be accounted for by reanalyzing only the changed partitions. In a final pass, these MBRs are read from disk in order to form the first bucket and proceed with the bucket partitioning process.

Algorithm 2: Main Memory Algorithm

Input: $F, H, NN, FF, ndim, MAX\_MEM$
Output: $\{M_i, W_i : i \in 1...max\_mbrs\_read\}$

```
1  max_mbrs_read =
   MAX_MEM/((2 * ndim + 1) * sizeof(double));
2  factor = 1.0/round(F * FF);
3  for (i = 1; i < H; i++) do
4  |  factor = (1 + factor)/round(F * FF);
5  end
6  num_mbrs_at_level = round(NN/factor);
7  level_mbrs_read = 1;
8  while (num_mbrs_at_level > max_mbrs_read) do
9  |  num_mbrs_at_level/ = round(F * FF);
10 |  level_mbrs_read++;
11 end
12 Initialize(pq);
13 for (all R-tree nodes at level level_mbrs_read) do
14 |  pq.push(node, MBR(node));
15 end
16 while (pq.size() < max_mbrs_read) do
17 |  dqnode = pq.pop();
18 |  for (all children nodes of dqnode) do
19 |  |  pq.push(node, MBR(node));
20 |  end
21 end
22 for (all nodes in pq) do
23 |  M_i = MBR(node);
24 |  W_i = Subtree(node);
25 end
```

Figure 3 displays example splits obtained for the input space at the top of the figure using the fuzzy splitting algorithm. The bottom left part of the figure shows the fuzzy split obtained using the $equi - area$ heuristic where $O3$ is split between the buckets into components $O3_1$ and $O3_2$. The bottom right part of the figure shows the corresponding split achieved using the $equi - count$ heuristics. Note that we may choose to assign any part of $O3$ to either of the two buckets. A simple solution splits the object $O3$ along the split line which is the center of the original bucket.

3.2 Main Memory Implementation

In the presence of very large datasets and limited main memory, the fuzzy partitioning algorithm requires disk-based processing as the buckets may be too large to fit in main memory. This is due to the fact that we need to store the MBR of each object in a bucket till all the required bucket splits have been performed. In this section, we explain our main memory implementation which performs the bucket splitting in memory.

The basic idea behind the approach is to avoid using the large number of leaf level entry MBRs as the starting point for bucket creation. Instead, we use MBRs at a higher level of the index, each of which may enclose a large number of leaf level entry MBRs. The problem reduces to determining the *best* set of MBRs to be used for initial bucket creation. The pseudocode for the procedure is listed in algorithm 2. The algorithm accepts as input the fanout F, height H, number of nodes NN, and the fill-factor FF of the R-tree index along with the main memory limit MAX_MEM and the dimensionality of the geometry data $ndim$.

The algorithm first determines the number of MBRs to be read from the index which will fit within the main memory

limit (line 1). Note that the MBR M_i and its weight W_i (number of geometry MBRs in its subtree) will be stored for each node entry that we will read for creating the initial bucket of the fuzzy partitioning algorithm. The algorithm then estimates the number of leaf level entry MBRs in the R-tree index which is equal to the number of geometry MBRs in the dataset (lines 2-6). The level at which MBRs should be read is set to 1 (line 7) which is the leaf level. The algorithm proceeds to determine at which level of the R-tree index MBRs should be read so as not to violate the main memory limit. It also estimates the number of MBRs at this level of the R-tree (lines 8-11). We experience typical fanout values of 34 which means the number of MBRs may be 30 or so times lower when we move up the R-tree hierarchy by a single level. Hence, it may be possible to accommodate a large number of children MBR entries (but not all) at the next lower level (lines 12-21).

We probe the R-tree nodes with the largest MBRs to see if their children nodes may fit in memory as nodes at the same level are expected to have similar number of child entries due to balancing properties of our R-tree. This procedure is facilitated by initializing a priority queue of R-tree nodes with node MBR area as the key (lines 12-15). Children entries are inserted into the priority queue and may be probed further if they have large MBRs (for regions which have lower data density) (lines 16-21). Once the size of the priority queue reaches the maximum permissible MBR count, we compute the MBR (line 23) and the weight (line 24) for each node in the priority queue. This set of weighted MBRs is passed onto the fuzzy partitioning algorithm for creating the first bucket in memory. Further bucket splitting is performed in main memory (as in algorithm 1) without the requirement for writing/reading buckets to/from the disk. The weight W_i for a node entry can be estimated to avoid traversal of lower levels of the index or determined exactly by index traversal. Experimental results reveal the pros and cons of each approach.

For *local partitioned indexes*, the process is similar to the above algorithm. However, a separate determination of the set of MBRs to be read is made for each partition with the requirement that the collective set of MBRs from all partitions must fit in main memory. In the next section, we present a comprehensive evaluation of our algorithms using real world datasets and show that the initial packing produced by our R-tree index combined with our fuzzy partitioning mechanism outperforms the disk-based approaches.

4. EXPERIMENTAL EVALUATION

In this section, we present the results from a comprehensive experimental evaluation performed using datasets with different geometry types. All experiments were performed on a virtual machine hosted on an Intel® Xeon® (CPU X5675 @ 3.07GHz) server with 6GB of RAM. We test with different algorithmic implementations described in table 1 for easy cross-referencing with the results. The Equi-Area and Equi-Count algorithms were refined from the description in [8] in order to ensure that bucket splitting is always performed and does not fail in worst case scenarios. We describe the three datasets used in our experiments and the query workload generation process before proceeding with the experimental results.

4.1 Datasets

Algorithm	Description
EA	Bucket splitting utilizes equi-area heuristic.
EC	Bucket splitting utilizes equi-count heuristic.
FZEA	Equi-area heuristic used along with the fuzzy splitting algorithm.
FZEC	Equi-count heuristic used along with the fuzzy splitting algorithm.
MM	Equi-count heuristic with fuzzy splitting algorithm utilizing the main memory implementation with index traversal.
MMA	Equi-count heuristic with fuzzy splitting algorithm utilizing the main memory implementation approximating the weights of higher level nodes.

Table 1: Algorithmic implementations

We use three different datasets in our evaluation which contain point data, polyline data and polygon data. Figure 4 displays visualizations of the samples of the datasets. Each dataset is a 2D *geodetic* dataset which implies that we store the latitude, longitude for each point or a vertex of a geometry. When the data is indexed, *geocentric* 3D conversion is performed. Since we use the index for statistics collection, all experimentation is performed on 3D data in this section.

ABI: The American Business Information (ABI) dataset contains point data representing business locations in US. The dataset contains 10,279,241 point geometries.

EDGE: The EDGE dataset contains 30,379,990 polyline geometries which represent the road network in US.

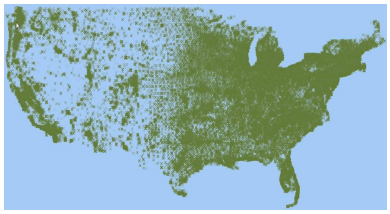
BLOCKS: The BLOCKS dataset is a polygon dataset containing 11,038,500 polygons representing US census blocks for the census data.

4.2 Query Workloads

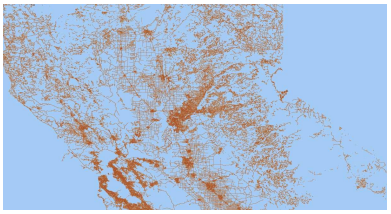
Previous work uses query workloads which have uniform distribution or zipfian distribution and query windows which are as large as 25% of the total space of the dataset. In general, we do not observe workloads with such large query windows. The space of our datasets covers the entire map of USA as visible in figure 4. We generate three workloads for each dataset with average query extent of 10 miles, 20 miles and 50 miles. Each workload comprises of a set of 1000 queries generated with buffers of appropriate distance around the centroids of the geometries in the datasets. Since we pick the geometries by sampling the dataset, the query workload can be reasonably expected to reflect the dataset distribution. Instead of characterizing the workloads using the query extent with respect to the total space, we determine the average selectivity of the queries in a workload. We believe this provides a better measure of the *size of the queries* instead of the query extent. The average selectivity values for the various workloads are as displayed in table 2 and we observe that almost all of our query workloads have less than 1% selectivity. We expect all our algorithms to provide more accurate selectivity estimates for workloads with larger selectivity values, a trend which is visible in our results.

	W_1, Q=10 mi	W_2, Q=20 mi	W_3, Q=50 mi
ABI	0.32%	0.68%	1.63%
EDGE	0.064%	0.18%	0.57%
BLOCKS	0.11%	0.3%	0.95%

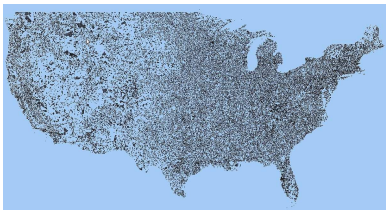
Table 2: Average Selectivity for Different Experimentation Workloads



(a) ABI point dataset



(b) EDGE polyline dataset



(c) BLOCKS polygon dataset

Figure 4: Visualization of the datasets used for experimental evaluation

4.3 Experimental Results

We present a detailed evaluation of our results for statistics collection and query performance for the datasets described above. We collect the statistics for each dataset using the different statistics collection methods and generate a histogram containing 512 buckets. Oracle uses a maximum of 255 buckets for single-dimensional data. Although larger number of buckets would result in increased accuracy in selectivity estimation, remember that a query needs to compute the intersection of its geometry with each of the bucket MBRs in the histogram in order to determine the selectivity estimate using the method stated in [8]. We need to keep the number of buckets low in order to limit the overhead introduced in the query execution time. Additionally, disk-based approaches require more reads/writes from/to the disk for a larger number of buckets. Results for larger number of buckets are discussed using the main memory approximation (MMA) approach.

4.3.1 Statistics Collection Runtime

Table 3 below details the runtime for the statistics collection procedure using the different approaches. Each experimental evaluation result has been averaged over ten runs.

	ABI	EDGE	BLOCKS
FZEA	1244s	3715s	1290s
FZEC	1543s	4286s	1612s
EA	1202s	3354s	1238s
EC	1439s	4067s	1556s
MM	327s	1181s	365s
MMA	28s	33s	30s

Table 3: Statistics Collection Runtime (in seconds)

We use 20MB of memory for the main memory implementations unless stated otherwise and bucket splitting is performed using the fuzzy equi-count heuristics. The corresponding index creation times for the ABI, EDGE and BLOCKS datasets are 948, 3240 and 1268 seconds, respectively. We observe that the main memory approximation (MMA) is an order of magnitude faster than the main memory (MM) approach as it avoids index traversal and two orders of magnitude faster than the disk-based approaches. As this approach avoids index traversal, the histogram construction time is almost independent of the size of the dataset. Also note that the fuzzy approaches are a little more expensive than the corresponding non-fuzzy approaches. This is due to the fact that the fuzzy approaches incorporate additional computation when partial MBR weights need to be accounted for while splitting a large MBR among multiple buckets. Additionally, the equi-count heuristic approaches are a little more expensive than their counterpart equi-area heuristic approaches as these approaches need to perform the MBR center projection on all the three dimensions to determine the ideal dimension for splitting a bucket. Hence, we conclude that the disk-based approaches are not feasible for large datasets as far as the statistics collection time is concerned and we should use the main memory approximation (MMA) whenever possible for statistics collection.

4.3.2 Query Selectivity Estimation

The most important performance parameter for the statistics collection algorithms is the query selectivity estimate generated using the histogram. The statistics collection needs to be performed only once and then updated if the data changes drastically over time. We compute the *relative error in selectivity estimation, RE,* for each query as,

$$RE = \frac{SE - AS}{AS},$$

where SE $(0 \leq SE \leq 1)$ denotes the selectivity estimate generated by the extensible optimizer and AS $(0 \leq AS \leq 1)$ denotes the actual selectivity of the query.

In order to quantize the selectivity estimation error, we compute the *Average Relative Error in Selectivity Estimation* for each workload W_j, j = 1, 2 or 3, computed as,

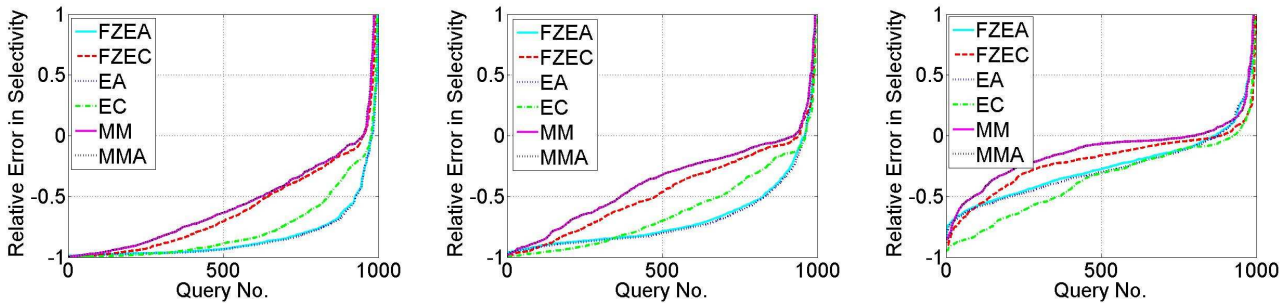
$$ARE = \frac{1}{N} \sum_{i=1}^{N} \frac{|SE_i - AS_i|}{AS_i},$$

We plot the *Actual Error in Selectivity Estimation, AE,* to display the actual differences between the query selectivity and selectivity estimate values.

$$AE = SE - AS,$$

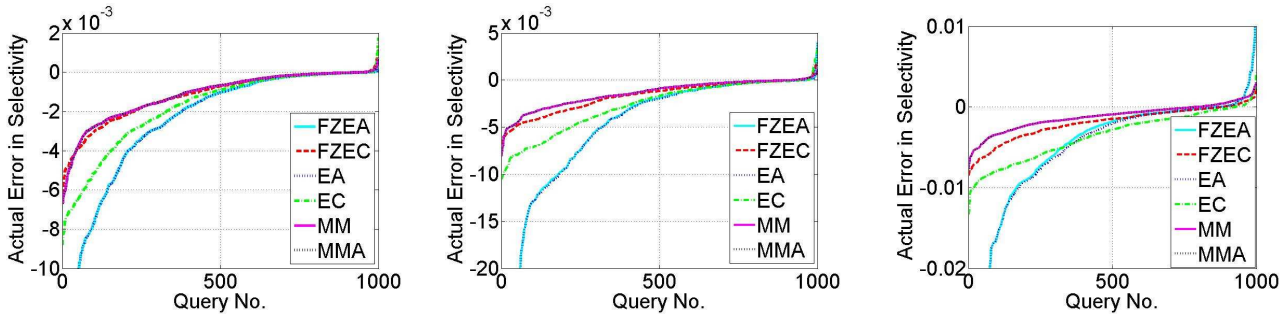
where SE denotes the selectivity estimate generated by the extensible optimizer and AS denotes the actual selectivity of the query. As long as the computed selectivity is close to the actual selectivity value there is a low probability of a bad plan being selected for query execution. We present the selectivity estimation performance using the filter operation which selects all geometry MBRs intersecting the query geometry MBR.

Figure 5 displays the results for the relative error in selectivity values for the ABI dataset for the three workloads (with query extent of 10, 20, and 50 miles). As can be observed from figure 5(a), the relative error in selectivity is reasonably high for a large number of queries with extent of 10 miles. In general, our workloads are such that the query windows generally lie inside one bucket (much smaller in extent than a single bucket) and are centered



(a) Query extent 10 miles (b) Query extent 20 miles (c) Query extent 50 miles

Figure 5: Relative error in selectivity estimation for the ABI dataset



(a) Query extent 10 miles (b) Query extent 20 miles (c) Query extent 50 miles

Figure 6: Actual error in selectivity estimation for the ABI dataset

around a dense region. This leads to lower selectivity estimates for a large number of queries as the selectivity estimation formula assumes uniform data distribution within each bucket. The skew towards selectivity underestimation is exacerbated for smaller query radii due to the same reason. Another observation from the figure is that the main memory (MM and MMA) approaches perform the best for all the query extents followed by the FZEC approach. The initial packing produced by our R-tree index is very effective and utilizing higher level MBRs instead of geometry MBRs for initial bucket construction leads to more accurate selectivity values. Various bucket splitting heuristics were used with the main memory approaches and results are shown using the fuzzy equi-count heuristic as it performs the best. The main memory approximation (MMA) is almost as accurate as the MM implementation (overlapping) although it is an order of magnitude faster. Due to the packed R-tree construction, node weight estimation is almost as accurate as node weight determination using the index traversal. In general, the equi-count heuristics approaches outperform the counterpart equi-area heuristics approaches and the fuzzy approaches outperform their non-fuzzy counterparts. This validates our development of the fuzzy methods. Note that as the dataset comprises of point data and hence lacks any large geometry MBRs, the performance improvement of fuzzy methods over the non-fuzzy methods is due to the minimal MBR count criterion which they follow for splitting a bucket.

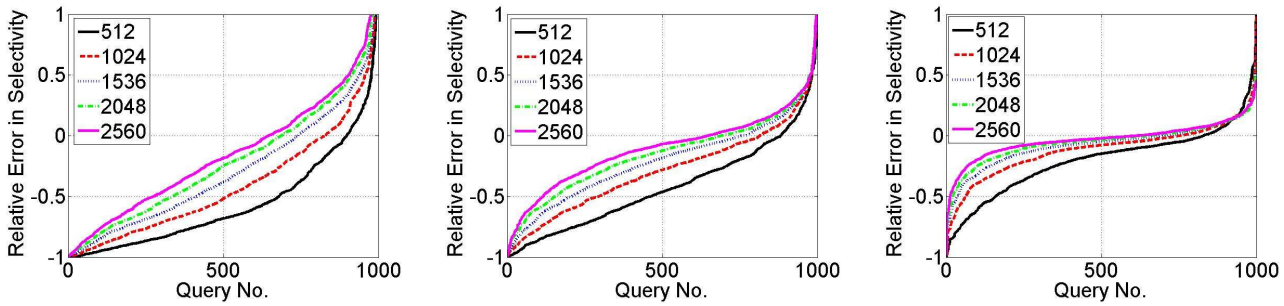
Table 4 lists the average relative error in selectivity estimates for all the query workloads for the ABI dataset which reaffirms what is visible in the figure using a single metric. The main memory approaches perform 5%, 17% and 13% better than the FZEC approach for the W_1, W_2 and W_3 workloads, respectively.

	W_1, Q=10 mi	W_2, Q=20 mi	W_3, Q=50 mi
FZEA	0.8532	0.7318	0.329
FZEC	0.6571	0.5007	0.2456
EA	0.8632	0.7446	0.3446
EC	0.7929	0.6487	0.3999
MM	0.6225	0.4271	0.2167
MMA	0.6234	0.4272	0.218

Table 4: Average Relative Error in Selectivity Estimation for the ABI Dataset

Figure 6 displays the actual error in selectivity values for the ABI dataset for the same set of queries. For the MM, MMA and FZEC approaches, the actual error in selectivity values is low even for query extent of 10 miles and 20 miles as can be seen from the figure which implies a very low probability of the optimizer picking a bad execution plan. The EA, EC and FZEA approaches do lead to higher errors for a few queries and are not recommended based on these results. The selectivity estimates for the queries which exhibit large relative errors are generally on the lower side, especially for smaller query extents, which implies that the probability of picking a full table scan when an index access is preferable is almost non-existent.

Tables 5 and 6 list the average relative error in selectivity estimates for all the query workloads for the EDGE and BLOCKS datasets, respectively. This reaffirms the performance numbers for the main memory approaches. In fact, the performance gap is larger for the non-point datasets. For



(a) Query extent 10 miles (b) Query extent 20 miles (c) Query extent 50 miles

Figure 7: Relative error in selectivity estimation for the EDGE dataset with varying number of buckets using the Main Memory Approximation approach

the polyline EDGE dataset, the main memory approaches perform 23%, 20% and 27% better than the FZEC approach for the W_1, W_2 and W_3 workloads, respectively. Similarly, for the polygon BLOCKS dataset, the main memory approaches perform 24%, 33% and 24% better than the FZEC approach for the W_1, W_2 and W_3 workloads, respectively. Hence, usage of higher level R-tree nodes for initial bucket construction positively impacts selectivity estimation performance of the main memory approaches for more complex geometry types. This is due to the fact that the R-tree packing uses the centroids of the actually geometries rather than center of the geometry MBRs. For the point dataset the centroid and MBR center are the same, whereas for more complex shapes these differ. Our results show that the higher level R-tree nodes clustered using the actually geometry centroids prove to be a better starting point for partitioning-based histogram construction when compared to an approach which uses the actual geometry MBRs as a starting point and performs partitioning based on the geometry MBR centers. The MMA approach performs as well as the MM approach for both EDGE and BLOCKS datasets.

	W_1, Q=10 mi	W_2, Q=20 mi	W_3, Q=50 mi
FZEA	0.8717	0.7633	0.4424
FZEC	0.7757	0.5893	0.3358
EA	0.8855	0.7843	0.4864
EC	0.7757	0.5893	0.3358
MM	0.6302	0.4886	0.2653
MMA	0.6302	0.4886	0.2653

Table 5: Average Relative Error in Selectivity Estimation for the EDGE Dataset

	W_1, Q=10 mi	W_2, Q=20 mi	W_3, Q=50 mi
FZEA	0.8557	0.6943	0.3109
FZEC	0.7006	0.4899	0.2087
EA	0.8934	0.7701	0.4207
EC	0.7006	0.4899	0.2087
MM	0.563	0.3667	0.1688
MMA	0.563	0.3667	0.1688

Table 6: Average Relative Error in Selectivity Estimation for the BLOCKS Dataset

4.3.3 *Query Execution Overhead*

The extensible optimizer needs to compute the intersection of the query geometry with each of the buckets in order to estimate the selectivity of the query. We limit the number of buckets generated by our histogram construction techniques to 512 to avoid unreasonable query execution overheads. This number is also appropriate as it results in acceptably low relative errors in selectivity estimates. The absolute execution overhead (in milliseconds) and the overhead expressed as a percentage of the query execution time for the different workloads for the ABI dataset are as shown in table 7. In general, larger extent workloads will intersect more buckets and hence lead to a larger absolute overhead. Similar results are seen for the other two datasets. Since we are performing a simple filter operation, the query execution times are low for the queries and we can expect even lower execution overheads for more complex spatial operators.

	W_1, Q=10 mi	W_2, Q=20 mi	W_3, Q=50 mi
FZEA	2.1ms, 2.2%	1.49ms, 1.4%	2.17ms, 1.5%
FZEC	2.33ms, 2.5%	1.8ms, 1.6%	1.8ms, 1.3%
EA	3ms, 3.1%	1.9ms, 1.9%	1.7ms, 1.2%
EC	1.5ms, 1.5%	1.6ms, 1.4%	2.4ms, 2%
MM	1.4ms, 1.4%	1.6ms, 1.4%	2.6ms, 2.1%
MMA	1.5ms, 1.5%	1.6ms, 1.4%	2.5ms, 2%

Table 7: Query Execution Overhead for the ABI Dataset

4.3.4 *Main Memory Approach Evaluation*

In this section, we vary the number of buckets in the histogram from 512 to 2560 in increments of 512 buckets and collect statistics using the MMA approach. The same query workloads as before are executed for the EDGE dataset (largest dataset) in this set of experiments. We do not observe a substantial increase in statistics collection time as we increase the number of buckets because the bucket splitting occurs in memory.

The relative error in selectivity estimation is plotted in Figure 7. The average relative error in selectivity estimates for the different workloads is displayed in table 8. As expected, the selectivity estimates improve as we increase the number of buckets. The performance improvement with larger number of buckets is more pronounced for workloads with higher selectivity. As we increase the number of buckets from 512 to 2560, we observe a 32% improvement in selectivity estimation for workload W_1. The corresponding numbers for workloads W_2 and W_3 are 50% and 62%, respectively.

Finally, we compute the query execution overhead and display its variation with the number of buckets used for the

	W_1, Q=10 mi	W_2, Q=20 mi	W_3, Q=50 mi
B=512	0.6302	0.4886	0.2653
B=1024	0.5379	0.3771	0.1756
B=1536	0.4832	0.3144	0.1377
B=2048	0.4534	0.275	0.1167
B=2560	0.4281	0.2445	0.1014

Table 8: Average Relative Error in Selectivity Estimation for the EDGE Dataset with varying number of buckets using the Main Memory Approximation approach

EDGE dataset in table 9. As expected, the query execution overhead increases as we increase the number of buckets due to larger bucket intersection computation costs. However, the overhead is less than 10% in most cases as visible from the table. This result shows that it may be feasible to have a larger number of buckets using the MMA approach, especially for more complex spatial operators where the larger absolute execution overhead will be lower when expressed as a percentage of the query execution time.

	W_1, Q=10 mi	W_2, Q=20 mi	W_3, Q=50 mi
B=512	2.5ms, 2.4%	2.7ms, 2.5%	2.9ms, 2.5%
B=1024	5.1ms, 4.9%	5.3ms, 4.3%	6.3ms, 3.4%
B=1536	10.7ms, 10.4%	8.8ms, 7.3%	9.4ms, 5.1%
B=2048	13.6ms, 13.3%	33.1ms, 7.1%	9.3ms, 5.1%
B=2560	17.3ms, 16.9%	17ms, 14.1%	17.9ms, 9.8%

Table 9: Query Execution Overhead for the EDGE Dataset with varying number of buckets using the Main Memory Approximation Approach

4.3.5 Unbalanced R-Tree

The MMA approach relies heavily on using the packed R-tree for estimating the weight of a high level MBR node. In this experiment, we delete data from the R-tree to test the performance of the two main memory approaches MM and MMA. We deleted 10%, 20% and 30% of the data from the spatial index and the underlying table for the BLOCKS dataset. The MMA approach still determines that the table has close to 11 million MBRs across all the constructed buckets as in the original table since estimates are based on a packed R-tree which is no longer the case. The MM approach determines the correct number of MBRs across all the buckets even with data deletion since it actually traverses the entire index. The selectivity estimates produced by the MMA approach are higher than the estimates produced by the MM approach as it is based on the assumption that the index has a larger number of geometry MBRs. This can be beneficial if the selectivity estimate is on the lower side for a large number of queries as in our case. However, it is important to remember that even though we observe improved average estimates for many workloads we get bad worst case errors with the MMA approach on heavy data deletion.

	W_1, Q=10 mi			W_2, Q=20 mi			W_3, Q=50 mi		
MM	0.567	0.578	0.569	0.388	0.388	0.389	0.180	0.180	0.181
MMA	0.530	0.507	0.486	0.340	0.321	0.342	0.162	0.221	0.316

Table 10: Average Relative Error in Selectivity Estimates for the BLOCKS dataset with [10%|20%|30%] of the original data deleted

The average relative error in selectivity estimates for the two approaches is displayed in table 10 when we delete 10%, 20% and 30% of the data from the R-tree and the underlying spatial table. We can observe that MMA approach performs worse than MM approach only for workload W_3 with 20% and 30% of the data deleted; for all other cases the higher estimates have a positive impact on the average relative selectivity values. As an example for the worst case errors, for workload W_2 with 20% data deletion none of the queries have a 100% relative error in selectivity estimate for the MM approach whereas 7 queries have a higher than 100% error for the MMA approach. If a large amount of data has been deleted, users either need to reconstruct the R-tree and collect statistics or they need to use the MM approach with index traversal to get accurate statistics.

4.4 An Example Study

In this section, we discuss an example study which exhibits the efficacy of our framework on queries in Oracle. Let us consider the following query on the EDGE dataset.

```
SELECT COUNT(*) FROM EDGE A WHERE
 EDGE_ID BETWEEN 395000000 and 401100000 AND
 SDO_ANYINTERACT(
  A.GEOMETRY,
  SDO_GEOMETRY(2003, 8307, NULL,
               SDO_ELEM_INFO_ARRAY(1, 1003, 3),
               SDO_ORDINATE_ARRAY(-80, 20, -60, 35.5))
                ) = 'TRUE';
```

The query attempts to retrieve the count of rows which satisfy the spatial SDO_ANYINTERACT operator as well as the non-spatial range operation on EDGE_ID. We have a spatial domain index on the SDO_GEOMETRY column as well as a B-tree index on the EDGE_ID column. In the presence of both indexes, the optimizer may decide to use neither, one or both the indexes based on the selectivity estimates. If the optimizer decides to use both the indexes it has to perform a *bitmap conversion* operation as displayed in the query plan detailed below.

```
-------------------------------------------------------------
| Id  | Operation                           | Name          |
-------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |
|   1 |  SORT AGGREGATE                     |               |
|   2 |   BITMAP CONVERSION COUNT           |               |
|   3 |    BITMAP AND                       |               |
|   4 |     BITMAP CONVERSION FROM ROWIDS   |               |
|   5 |      SORT ORDER BY                  |               |
|*  6 |       INDEX RANGE SCAN              | EDGE_ID_IDX   |
|   7 |     BITMAP CONVERSION FROM ROWIDS   |               |
|   8 |      SORT ORDER BY                  |               |
|*  9 |       DOMAIN INDEX (SEL: 2.262042 %)| SMALLEDGE_SIDX|
-------------------------------------------------------------
```

Due to incorrect selectivity and cost values, the optimizer often chooses a bad plan in the presence of a spatial and a non-spatial predicate. We conducted a study on various datasets to determine the efficacy of our extended spatial optimizer framework in handling such queries. We vary the selectivity of both the spatial and non-spatial predicates and record the query execution time and the selected plan. We present the execution times with and without our spatial cost and selectivity framework in tables 11 and 12, respectively, for queries on the EDGE dataset.

As can be observed from the above results, the query execution times are reduced by 2.5 to 10 times in many cases with our framework. The Oracle optimizer has a tendency

	S 1%	S 5%	S 10%	S 20%
NS 1%	0.7 ✓	2 ✓	3.5 ✓	6.9 ✓
NS 5%	4.1 ×	16.3 ×	35.8 ×	79 ×
NS 10%	4.2 ×	16.2 ×	35.9 ×	79.3 ×

Table 11: Execution time (in seconds) without the framework (✓ or × indicates if the bitmap conversion operation was used or not. S indicates spatial selectivity and NS indicates Non-spatial selectivity.)

	S 1%	S 5%	S 10%	S 20%
NS 1%	1 ✓	2 ✓	3.9 ✓	7.4 ✓
NS 5%	1.5 ✓	2.9 ✓	4.3 ✓	8 ✓
NS 10%	4.1 ×	3.7 ✓	5.3 ✓	8.6 ✓

Table 12: Execution time (in seconds) with the framework (✓ or × indicates if the bitmap conversion operation was used or not. S indicates spatial selectivity and NS indicates Non-spatial selectivity.)

to use the spatial index alone in the absence of the correct selectivity and cost values. The optimizer does a much better job of selecting the bitmap conversion plan as the best plan with the aid of our spatial statistics framework. Similar improvements in execution times have been observed for a number of bug cases related to the optimizer in the context of spatial queries. We verified that for spatial predicate selectivity of 1% and non-spatial predicate selectivity of 10% the spatial domain index alone performs better than the bitmap conversion operation.

5. RELATED WORK

One-dimensional histograms have been used widely for selectivity estimation in databases [30, 22]. Oracle provides effective statistics collection techniques for large single-dimensional data tables using the one-pass distinct sampling method [13]. However, multi-dimensional histograms, such as those required for spatial data, pose an entirely different challenge [21]. In order to address these challenges various heuristics-based methods, motivated by the results in [29], have been proposed.

The authors in [28] propose the *hTree* which builds non-overlapping partitions of multi-dimensional space based on frequency. An *equi-depth* histogram is constructed by utilizing a single dimension at a time with an equal number of objects in each bucket. Although the partitioning rule does not work well for skewed multi-dimensional data, low construction cost is a notable advantage associated with this method. On the other hand, *mHist* uses space partitioning where partitioning is performed along the dimension which *provides the greatest benefit*. The split decision is made based on marginal frequency distributions. This work extends approaches which were developed for relational data but are useful mainly for point data.

Selectivity estimation in spatial databases differs from traditional data as emphasized in [8]. In addition to the equi-area and equi-count approaches, a *Min-Skew* method is proposed in this work. The algorithm constructs a rectangular grid and stores the number of intersecting spatial objects for each cell. Further, recursive *binary space partitioning (BSP)* is used for histogram construction. The basic idea is to pick buckets for further splitting based on a split value that will produce the greatest reduction in the data skew. However, the performance of this strategy is sensitive to the grid resolution. Oracle Spatial and Graph has deprecated the grid-based Quadtree index methods in a past release in favor of the Oracle R-tree index [25]. Grid-based methods, being sensitive to resolution, leave the onus on the user to determine an important factor which dictates the end result, hence, we avoid their usage in Oracle Spatial and Graph.

The *GenHist* method attempts to identify high density regions [16]. As opposed to previous methods, the bucket rectangles may overlap. Moreover, a bucket may contain other buckets. Note that previous methods aim to produce non-overlapping buckets which is not true for complex spatial geometry types (with large spatial extents) as displayed in our example in section 2.3. Once again, this method uses a rectangular grid as a starting point thus making it dependent on the initial grid resolution. STHist [31] applies the idea of *GenHist* to 2D and 3D spatial objects. In the basic algorithm, dense regions are determined by applying a sliding window over each dimension, approximating the frequency distribution with a marginal distribution. The histogram is built as an unbalanced R-tree by building hierarchies over the dense regions (*Hot-spots*). An advanced variant, called the *STForest*, computes initial partitions according to the object skew. Further partitioning is avoided for regions which are already uniformly distributed. Buckets merge together if the skew of the merged buckets decreases. Although, superior to other proposed methods in selectivity estimation performance, time complexity is $O(n^2)$ for 2D and $O(n^3)$ for 3D data.

Self-tuning histograms like *STHoles* [12] and *ISOMER* [33] have also been proposed. These methods incrementally update buckets and their frequency information using the query feedback mechanism. It would be an interesting direction for Oracle Spatial and Graph to explore such self-tuning methods in the future as these methods adapt to real data distribution over time. Application of these methods on top of currently implemented algorithms would be an ideal approach to follow.

Our main memory algorithm attempts to use the R-tree hierarchy for selecting the *best* set of MBRs, which fit in main memory, for histogram construction. Methods proposed in [8, 9, 21] use the R-tree structure for spatial histogram construction. [14] proposes the *rkHist* approach which uses the R-tree bulk-loading procedure [23] for histogram construction. Data is presorted according to the Hilbert space filling curve mechanism. A greedy algorithm is proposed which utilizes a sliding window along the Hilbert order to effectively pack the leaf nodes. Our methods aim to utilize the existing Oracle R-tree to construct the appropriate histograms taking the main memory limit and time constraints into account.

6. CONCLUSION

In this paper, we describe the implementation of an extensible optimizer within Oracle Spatial and Graph and discuss issues faced. A main memory-based algorithm is proposed which limits the histogram construction time and constructs accurate histograms using our R-tree index. We provide experimental results with different spatial geometry types and observe that the main memory approach when combined with the fuzzy equi-count heuristics outperforms other methods in terms of accuracy as well as histogram construction time. The spatial statistics collection plan a DBA may

want to follow can be time constrained and workload dependent. We would recommend the main memory approximation approach for fast, accurate statistics collection. When a substantial amount of the original data has been removed from the spatial table the approximation can lead to inaccurate results and users may resort to the main memory approach with index traversal for accurate results. We believe that our results have rendered the disk-based approaches to be useless for large spatial datasets.

7. ACKNOWLEDGEMENTS

The authors would like to thank Dinesh Das from the optimizer team for his guidance with Oracle Database extensibility mechanism.

8. REFERENCES

[1] IBM Informix Spatial DataBlade. `http://www-01.ibm.com/software/data/informix/blades/spatial/`.

[2] Microsoft SQL Server 2012. `http://msdn.microsoft.com/en-us/library/bb418440(v=sql.10).aspx`.

[3] Oracle Exadata Database Machine. `http://www.oracle.com/us/products/database/exadata/overview/index.html`.

[4] Oracle Spatial and Graph. `http://www.oracle.com/us/products/database/options/spatial/overview/index.html`.

[5] PostGIS. `http://postgis.refractions.net`.

[6] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated Statistics Collection in DB2 UDB. In *VLDB*, pages 1158–1169, 2004.

[7] A. Aboulnaga and J. Naughton. Accurate Estimation of the Cost of Spatial Selections. In *IEEE ICDE*, pages 123–134, 2000.

[8] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity Estimation in Spatial Databases. In *ACM SIGMOD*, pages 13–24, 1999.

[9] P. Aoki. How to Avoid Building DataBlades (r) That Know the Value of Everything and the Cost of Nothing. In *IEEE SSDBM*, pages 122–133, 1999.

[10] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD*, pages 322–331, 1990.

[11] T. Brinkhoff, H. Horn, H. Kriegel, and R. Schneider. A Storage and Access Architecture for Efficient Query Processing in Spatial Database Systems. In *SSD*, pages 357–376, 1993.

[12] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A Multidimensional Workload-aware Histogram. In *ACM SIGMOD*, pages 211–222, 2001.

[13] S. Chakkappen, T. Cruanes, B. Dageville, L. Jiang, U. Shaft, H. Su, and M. Zait. Efficient and Scalable Statistics Gathering for Large Databases in Oracle 11g. In *ACM SIGMOD*, pages 1053–1064, 2008.

[14] T. Eavis and A. Lopez. Rk-hist: An R-tree based Histogram for Multi-dimensional Selectivity Estimation. In *ACM CIKM*, pages 475–484, 2007.

[15] P. Gibbons. Distinct Sampling for Highly-accurate Answers to Distinct Values Queries and Event Reports. In *VLDB*, pages 541–550, 2001.

[16] D. Gunopulos, G. Kollios, V. Tsotras, and C. Domeniconi. Approximating Multi-dimensional Aggregate Range Queries over Real Attributes. In *ACM SIGMOD*, pages 463–474, 2000.

[17] D. Gustafson and W. Kessel. Fuzzy clustering with a fuzzy covariance matrix. In *IEEE Conference on Decision and Control*, pages 761–766, 1978.

[18] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, pages 47–57, 1984.

[19] P. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based Estimation of the Number of Distinct Values of an Attribute. In *VLDB*, pages 311–322, 1995.

[20] P. Haas and A. Swami. Sampling-based Selectivity Estimation for Joins using Augmented Frequent Value Statistics. In *IEEE ICDE*, pages 522–531, 1995.

[21] Y. Ioannidis. The History of Histograms (abridged). In *VLDB*, pages 19–30, 2003.

[22] H. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. In *VLDB*, pages 275–286, 1998.

[23] I. Kamel and C. Faloutsos. On packing R-trees. In *ACM CIKM*, pages 490–499, 1993.

[24] K. Kanth, S. Ravada, J. Sharma, and J. Banerjee. Indexing Medium-dimensionality Data in Oracle. In *ACM SIGMOD*, pages 521–522, 1999.

[25] R. Kothuri, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison using GIS Data. In *ACM SIGMOD*, pages 546–557, 2002.

[26] R. Kothuri, S. Ravada, and N. An. Incorporating Updates in Domain Indexes: Experiences with Oracle Spatial R-trees. In *IEEE ICDE*, pages 745–753, 2004.

[27] R. Lipton, J. Naughton, and D. Schneider. Practical Selectivity Estimation through Adaptive Sampling. In *ACM SIGMOD*, pages 40–46, 1990.

[28] M. Muralikrishna and D. DeWitt. Equi-depth Multidimensional Histograms. In *ACM SIGMOD*, pages 28–36, 1988.

[29] S. Muthukrishnan, V. Poosala, and T. Suel. On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity and Applications. In *ICDT*, pages 236–256, 1999.

[30] V. Poosala, P. Haas, Y. Ioannidis, and E. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *ACM SIGMOD*, pages 294–305, 1996.

[31] Y. Roh, J. Kim, Y. Chung, J. Son, and M. Kim. Hierarchically Organized Skew-tolerant Histograms for Geographic Data Objects. In *ACM SIGMOD*, pages 627–638, 2010.

[32] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A Dynamic Index for Multi-dimensional Objects. In *VLDB*, pages 40–46, 1987.

[33] U. Srivastava, P. Haas, V. Markl, M. Kutsch, and T. Tran. ISOMER: Consistent Histogram Construction using Query Feedback. In *IEEE ICDE*, pages 39–50, 2006.

[34] X. Xie and G. Beni. A Validity Measure for Fuzzy Clustering. *IEEE Transations on Pattern Analysis and Machine Intelligence*, 13(8):841–847, 1991.