In *Proceedings of the 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Redondo Beach, CA, USA, November 2012.

Multiresolution Select-Distinct Queries on Large Geographic Point Sets

Sarana Nutanong Marco D. Adelfio Hanan Samet

Center for Automation Research, Institute for Advanced Computer Studies,
Department of Computer Science, University of Maryland
College Park, MD 20742, USA
{nutanong, marco, hjs}@cs.umd.edu

ABSTRACT

Many spatial applications require the ability to display locations of data entries on an online map. For example, an online photo-sharing service may wish to display photos according to where they were taken. Since many photos can occupy the same area and overlap each other within a display window, less popular or older images (based on a given measure of importance) can be discarded so that these more popular or newer photos become more distinct. A straightforward solution to this problem is (i) to use a window query to retrieve data entries within a given display window; (ii) to discard data entries in proximity of a more important one. This method works well in a high spatial selectivity setting, e.g., when the window query returns a small number of entries, but the performance drastically degrades as the spatial selectivity decreases. We consider this problem as selecting distinct data entries from a given dataset, where the "distinctiveness" of a data entry depends on its relative importance in comparison to that of other data entries in proximity. In this paper, we propose a new query type called the *multi-resolution select-distinct (MRSD) query*. The main novelty of our query processing method is a voting system built upon an ensemble of interrelated indexes, which allows us to efficiently determine the degree of distinctiveness of all points within a query window. Using a real dataset of over 9 million locations, our experimental results show that our proposed method is capable of consistently producing subsecond response times, while the window query-based method takes more than 10 seconds on average in a low spatial selectivity setting.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*; H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms, Design

Keywords

Spatial databases, Query processing.

ACM SIGSPATIAL GIS '12, November 6-9, 2012. Redondo Beach, CA, USA

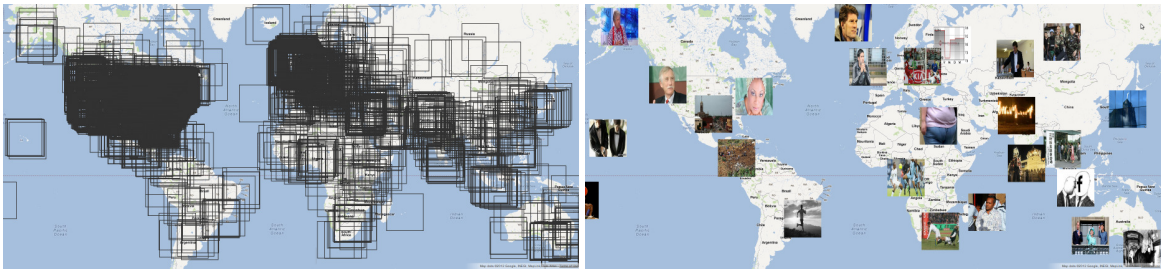
1. INTRODUCTION

All queries on spatial data can be classified as location-based (e.g., QUILT [9, 14] and the SAND Browser [7]) or feature-based (also known as spatial data mining) [1]. In this paper, we focus on a variant of a location-based query where the location is a map window, which is motivated by the problem of displaying the locations of data entries on an online map where a user manipulates the display window and the zoom level to specify an area of interest. For example, an online photo-sharing service may wish to display photos according to where they were taken and to allow its users to browse these images through an online map interface. Since many photos can occupy the same area in a display window, we may choose to display only a subset of photos instead. These photos can be laid out based on a scoring system denoting their importance, which can be derived, for example, from how recent each image was taken and the number times it has been viewed.

Our problem can be formally described as finding an appropriate layout for a collection of spatial data entries in a display window. Specifically, given a large geographic dataset S and a geographic query window W, select a subset T of data entries from S that fall within W. Our investigation focuses on a case where data entries are represented as equally-sized image thumbnails. Figure 1(a) shows the locations of all images from an example database that fall within a sample query window W. Only the image outlines are displayed, in order to expose the differences in density of images around the map. Figure 1(b) shows a set T of images selected using our method from a database of images from news article [11]. Notice that the selected images do not result in large overlaps, and are distributed throughout the query window. In particular, we want the way in which the data entries in T are selected to satisfy the following design objectives which have also been used to evaluate the iOS6 Apple Maps API [8].

(i) *Minimize overlaps.* Displaying or labeling any entry from S on a map requires some amount of display space. If multiple proximate data entries are selected, their representations or labels may overlap, resulting in reduced legibility and less data clarity than is desired. In some spatial sampling applications, overlap between entries of T must be avoided completely. Assume that each data entry is represented as an image thumbnail with a size of $(\epsilon \times \epsilon)$. We can completely avoid overlaps by ensuring that no two data entries in T have a chessboard distance smaller than a proximity threshold ϵ. In applications where a slight overlap between pairs of entries is acceptable, this proximity threshold can be relaxed.

(ii) *Respect relative importance of entries.* Consider geographic datasets that include an importance measure for each entry. When two data entries/points are in proximity and one of them has to be discarded, we should keep the more important one and discard the other entry. In other words, T should include the most important entry for different regions in W.



(a) Outlines of all images (b) Distinct images selected using the MRSD query

Figure 1: Selecting images from a set of geographically overlapping images obtained from a database of geotagged news articles.

(iii) *Maximize spatial fullness.* The subset T should cover most of the area that is covered by all entries of S within W. That is, if one geographic region within W contains far more entries than others, we still expect to see entries in all regions of W that contain data points in S.

(iv) *Provide panning/zooming consistency.* If a geographic window W contains a region R, and another geographic window W', obtained by panning the map, also contains R, the spatial sample within R should be identical in both cases. Similarly, zooming consistency requires that entries selected within W should also be selected within W', when W' is obtained by zooming in from W.

(v) *Enable efficient sampling.* Even with millions of candidate entries of S within W, we still desire a fast response time to select the subset T. Note that this property restricts us from using standard map labeling algorithms that provide super-linear running time (with respect to the number of entries in W) for the selection and placement of non-overlapping map labels.

(vi) *Support filtering conditions.* Allow feature-based filters to guide the spatial sampling process, so that T does not contain any data entries that are filtered out. The filters may vary in their selectivity, in the sense that some filters could remove many data entries from being selected into T, while others may have little effect on T because they do not exclude many of the entries. This goal (in combination with the spatial fullness property) precludes the use of some precomputation-based approaches [13].

In order to fully appreciate the challenges of designing a solution to satisfy these properties, we describe three sample approaches and their drawbacks. One approach is to randomly sample a subset of n entries within W, and iteratively select elements from the subset that do not overlap previously selected entries. This approach can be implemented efficiently (using a spatial index) and avoids overlaps, but does not fulfill the spatial fullness or panning/zooming consistency properties, nor does it respect entry relevance. A second approach is to select the most important n images within W, and then to iteratively select elements from this subset that do not overlap with previously selected entries. This can be done efficiently and avoids overlaps, but again does not fulfill the spatial fullness or panning/zooming consistency properties. A third approach is to start with all entries within W ordered by importance, and then iteratively select entries that do not overlap with previously selected entries. This is optimal for minimizing overlaps, achieves spatial fullness, and respects relative importance of entries. However, it can be very inefficient to consider all elements in W (when W is large or S is dense within W) and panning/zooming consistency will be violated in border areas.

We model this layout problem as a problem of selecting "distinct" data entries/points based on a measure of importance where a data point is considered distinct if there are no other nearby data points with a greater importance. Our solution is motivated by the `SELECT DISTINCT` query used in relational database management systems (RDBMS) which provides a way of selecting representative data entries by (i) grouping identical values from a given column; (ii) selecting one representative/distinct data entry from each group based on a set of ordering criteria. This query can be efficiently processed using an appropriate index, which will be further explained in Section 5. Since grouping works for only discretized attributes, it cannot be directly applied to spatial data where grouping is based on proximity instead of an exact match.

One way to adapt the `SELECT DISTINCT` query to our layout problem is to partition the data space into a grid where the grid cell size is given by a proximity threshold ϵ. We can then group data points according to the grid cells in which they reside. However, when two data points are separated by a grid boundary, this method considers them as being in two different groups regardless of how close they are. If these two data points are represented as image thumbnails with a geographic size of $(\epsilon \times \epsilon)$, they can significantly overlap. It is also important to point out that a user may change the zoom level while the pixel size of thumbnails remain unchanged, which effectively changes the proximity threshold relative to the map area. Hence, our indexes need to be able to answer queries as the proximity threshold ϵ changes according to the zoom level.

We introduce a new query type called *Multiresolution Select-Distinct (MRSD)*. Our query differs from the one which detects spatial duplicates [2] in the sense that MRSD uses a proximity threshold which is determiend by the current zoom level/resolution of the map. Specifically, the query accepts a proximity threshold ϵ (e.g., the width of each image thumbnail) and a query window (x_1, y_1, x_2, y_2) and returns a result set containing data points residing in the query window. The crux of our method is a voting system built upon an ensemble of interrelated indexes, which allows us to efficiently determine the degree of distinctiveness of all points within a query window. Specifically, our indexes are derived from the Morton code representations [5] of data points under multiple translations. As we explain, the fractal nature of the Morton order combined with our choice of translations enables us to support the MRSD queries at multiple resolutions efficiently.

In summary, contributions of our work are given as follows.

- A new query type called *Multiresolution Select-Distinct (MRSD)* which can be used to solve image layout problems.
- A complete RDBMS solution containing a method to process MRSD queries and a set of indexes that work with multiple map resolutions.
- Experimental studies which demonstrate the effectiveness of our proposed solutions in comparison to an alternative method which considers all entries in the query window.

The rest of this paper is organized as follows. In Section 2, we discuss work related to the image layout and proximity query problems. Section 3 contains background materials. Section 4 provides the definitions of our MRSD query and the distinctiveness score. Section 5 presents our proposed method to process MRSD queries. In Section 6, we report results from our experimental studies. Section 7 concludes the paper.

2. RELATED WORK

Our problem can be defined as one of spatial sampling. That is, our aim is to present an appropriate sample of available data by taking spatial considerations into account. The related problems of map labeling and "thinning" geographic datasets involve similar requirements. In the map labeling scenario, the desired output is a selection of features to label, along with positions for those labels that avoid overlaps [3, 17]. Map labeling has been the subject of a large body of work, which generally focuses on selecting a maximal set of labels to display, and finding optimal placements for those labels. In contrast, our focus is on highly selective pruning of large datasets. For example, when labeling a country-level map, neighborhood street names can be safely pruned without considering their spatial properties.

A similar problem of "thinning" large geographic datasets for display has been defined by Sarma et al. [13], which is currently being used to perform spatial sampling by Google Maps. Specifically, they focus on selecting a finite list of features to appear within uniform regions at each zoom level. The authors argue that features displayed on their mapping application hardly change and can be precomputed periodically. Specifically, features are sampled in advance using an integer programming model, which includes constraints that enforce consistency while panning and zooming. The precomputation approach prioritizes query speed, at the expense of flexibility to perform run-time query filtering, and increased costs of inserting and updating data features. Our approach differs because it computes the list of data entries to display separately for each query, instead of during precomputation, which allows us to handle filters and avoids recomputation after each insert, update, or delete operation. Yet, as our experiments show, querying is still very efficient, even with several million entries in our dataset.

3. BACKGROUND

The notion of selecting distinct/representative data entries from a large collection of data is well known in relational database management systems (RDBMSs). In this section, we discuss the concept of selecting distinct data entries from different groups in the RDBMS context and a simple technique to apply this query type to spatial data. Let us consider an example of the `SELECT DISTINCT` query in Table 1. Assume that we use the population as our *importance score* and the cities are grouped according to their states. That is, we wish to select the largest city by population in each state from Table 1 (`table_1`). This operation can be done using the following SQL statement in PostgreSQL [4].

```
SELECT DISTINCT ON (State) * FROM table_1
 ORDER BY State DESC, Population DESC
```

This SQL statement produces the data entries highlighted in Table 1. The `DISTINCT ON (State)` clause dictates that only one entry from each state can be selected and `ORDER BY State DESC, Population DESC` dictates that we want the data entries to be sorted by states and with in each state we want the entry with the largest population to appear as the first one. Given that the column pair `(State, Population)` is indexed in a B-tree, this query can be processed via a *loose index scan* [18]. Specifically, we can traverse the index in descending order (from Providence, Rhode Island to Waterbury, Connecticut). Once the city with the largest population in a state is found, the rest of the cities in that state need not be considered. When the selectivity is high, the loose index scan method allows us to retrieve the desired results by considering a small portion of the dataset.

Alternatively, the given `SELECT DISTINCT` query statement can also be rewritten using the `PARTITION BY` operator to organize the data entries into states and to select the one with the highest population in each partition. For our presentation, we use the former alternative because we wish to keep our query statements concise and emphasize the fact that we want to select "distinct" spatial data entries from a dataset.

Table 1: Cities in New England with population of 100,000 or greater where the largest city in each state is highlighted.

State	City	Lat	Long	Population
Connecticut	Waterbury	41.09	-73.00	107,902
Connecticut	Stamford	41.01	-73.09	120,045
Connecticut	Hartford	41.13	-72.11	124,397
Connecticut	New Haven	41.05	-72.15	124,791
Connecticut	Bridgeport	41.03	-73.03	139,008
Massachusetts	Cambridge	42.06	-71.02	101,355
Massachusetts	Lowell	42.11	-71.05	105,167
Massachusetts	Springfield	42.02	-72.10	152,082
Massachusetts	Worcester	42.04	-71.13	172,648
Massachusetts	Boston	42.06	-71.01	589,141
New Hampshire	Manchester	42.16	-71.08	109,691
Rhode Island	Providence	41.14	-71.07	176,862

A simple method to allow the `SELECT DISTINCT` query to be applied to spatial data is to discretize the data space using a space filling curve like the Morton order. The Morton order is a grid decomposition technique, where the data space is subdivided into $(2^b \times 2^b)$ equal sized blocks, where b is the number of decomposition levels. This decomposition scheme is common to a family of spatial data structures, namely the quadtree and its variants [5]. The Morton order exhibits a recursive structure, which we exploit to support our query on multiple map resolutions. Figure 2 shows how a Morton order of data points in 2D can be calculated. Given a set $\mathcal{D}$ of data points in a finite space, Morton order of the data points in $\mathcal{D}$ can be obtained by subdividing the space into $2^b \times 2^b$ equal sized blocks, where b is a positive integer. These blocks are then numbered according to the Z-order curve. We can consider b as the number of levels of decompositions. In this example, the value of b is 2, which means that we have two levels where the first level corresponds to the decomposition which subdivides the space into four blocks and the second level corresponds to the decomposition which subdivides the space into 16 blocks. All blocks in the same quadrant share the same first two digits. For example, all blocks in the top left quadrant (1000, 1001, 1010 and 1011) share the same first two digits of 10. A further decomposition (the third level) would result with a similar pattern. This recursive nature of the Morton order can be exploited to allow us to use the same set of codes in multiple map resolutions where each level adds the number of Morton code digits we have to consider by two. Consider the data entries $\boldsymbol{c}$ and $\boldsymbol{d}$ at their original locations as an example. At the first level, we only need to consider the first two digits. In this case, $\boldsymbol{c}$ and $\boldsymbol{d}$ share the same first two digits of 00, and hence they are considered nearby. When we examine blocks at the next level, we consider the next two digits of $\boldsymbol{c}$ and $\boldsymbol{d}$, which indicates that they are no longer considered nearby in this zoom level.

4. DETERMINING DISTINCTIVENESS

In this section, we describe how to identify a set of spatially distinct data entries from a large geographic dataset. Unlike the example given in Table 1, where grouping is done on a discretized attribute, the continuous nature of spatial data means that grouping has to be done based on proximity rather than an exact match. We derive our distinctiveness measure which reflects the importance of a data entry relative to its surroundings. This cannot be captured by simply discretizing the data space into grid cells, because two points that are in proximity but on different sides of a cell boundary are not grouped together. Consider points $\boldsymbol{a}$ and $\boldsymbol{b}$ (at their original locations) in Figure 2 as an example. For the second level, these two

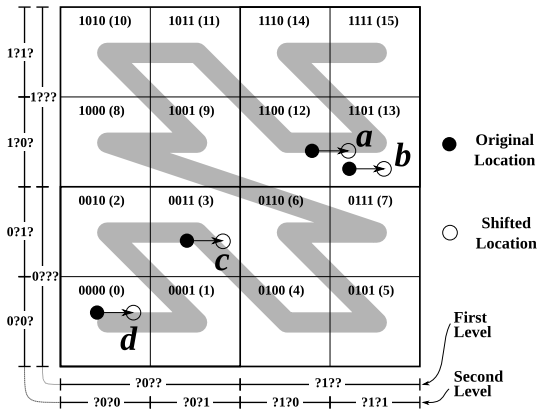


Figure 2: Handling boundary cases through shifting.

points $\boldsymbol{a}$ and $\boldsymbol{b}$ are not considered nearby since they are in different blocks 1100 and 1101, respectively. However, $\boldsymbol{b}$ is much closer to $\boldsymbol{a}$ than a lot of points that could be in the same block as $\boldsymbol{a}$.

4.1 Morton Index with Translations

Our proposed method is based on the fact that two points that are in proximity are likely to be in the same block. Figure 2 shows four data points, along with translated locations, slightly shifted to the right. The two data points $\boldsymbol{a}$ and $\boldsymbol{b}$ are in different blocks according to their original locations, but are in the same block according to the shifted locations. We can also see that there are a number of other translations that would yield a similar result. Hence, for each dataset, we can compute Morton codes with different translations. Determining whether two points are in proximity can be done by comparing the two points in all translations, and checking whether they share the same block for a majority of the translations.

Let us now consider Figure 3, which contains five data entries $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$ and $\boldsymbol{q}$ under three different translations. The figure shows that the number n_t of translations in which two data points share the same block has a negative correlation with the distance between them. For example, $\boldsymbol{a}$, which is the closest entry to $\boldsymbol{q}$, shares the same block as $\boldsymbol{q}$ in all three translations. We can also see that farther entries such as $\boldsymbol{c}$ and $\boldsymbol{d}$ have a smaller number n_t of translations in which they share the same block as $\boldsymbol{q}$. The lower and upper bound of the distance from $\boldsymbol{q}$ is given in Table 2 for each possible value of n_t. In order to optimize proximity detection, these offsets must be "well-interleaved" (Definition 1). In the next subsection, we describe the way we determine a set of well-interleaved offsets for multiple levels.

DEFINITION 1 (WELL-INTERLEAVED OFFSETS). *Given a block size w, a set $\{o_1, o_2, ..., o_n\}$ of n translation offsets is well interleaved if and only if the difference between any two adjacent offsets (i.e., o_i and o_{i+1}) is equal to (w/n).*

Table 2: Correlation between the number of shared blocks and the distance between two points.

Number n_t of blocks shared with $\boldsymbol{q}$	Distance from $\boldsymbol{q}$		Example
	Lower Bound	Upper Bound	
3	0	$(1/3) \cdot w$	$\boldsymbol{a}$
2	0	$(2/3) \cdot w$	$\boldsymbol{b}$
1	$(1/3) \cdot w$	w	$\boldsymbol{c}$
0	$(2/3) \cdot w$	∞	$\boldsymbol{d}$

To prevent calculating offsets for different levels of decomposition, we devise a set of offsets that are well-interleaved at all levels of decomposition. Note that the block size decreases as the level increases. For example, if the block width at a level d is w, then the block width at $(d+1)$ becomes $(w/2)$. As a result, a translation offset with respect to the block width at one level increases by a factor of 2 for the next level, which means that we can have translation offsets that are greater than the block width. Our observation is that, the arrangement of data entries returns to the original pattern every time the offset o is divisible by w. Using Figure 3 as an example, if we perform a third translation (i.e., Translation 3, where $o = 3/3w = w$), then we will obtain the same arrangement of points as in Translation 0. Specifically, each data point will have the same distances from the left and right boundaries of the block in which it resides, and if a pair of data points $\boldsymbol{x}$ and $\boldsymbol{y}$ share a block according to their original locations, then after applying an offset of w or any multiple of w, they will again share the same block. As a result, an offset o is equivalent to o any multiple of w as long as it does not put any data point outside the dataspace.

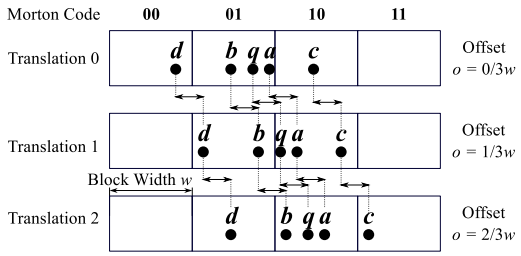


Figure 3: Data points under three different translations in 1D.

We introduce a notion of *modulo offset* (Definition 2) to express an effective offset from a given original offset.

DEFINITION 2 (MODULO OFFSET). *The modulo offset can be defined as a function $\mathrm{MO}(o, w)$ of an offset o and a block size w and is calculated as the floating point modulo $(o \bmod w)$ normalized by w. That is, $\mathrm{MO}(o, w) = (o \bmod w)/w$.*

According to this notion, we need to find a set $\{o_1, o_2, ..., o_n\}$ of n offsets such that the modulo offsets in

$$\{\mathrm{MO}(o_1, w/2^z), \mathrm{MO}(o_2, w/2^z), ..., \mathrm{MO}(o_n, w/2^z)\}$$

are well-interleaved for any integer value of z from 0 to ∞. To better explain this concept, let us first consider a "bad example" (Example 1) of an offset set.

Example 1. Given a set $\{o_1, o_2, o_3, o_4\}$ of a well-interleaved offset at level 1, which has a block size of w, this implies that we have an offset set of $\{0, \frac{w}{4}, \frac{2 \cdot w}{4}, \frac{3 \cdot w}{4}\}$. At level 2, the corresponding modulo offsets are given as • o_1 : $\mathrm{MO}(0, w/2) = 0$; • o_2 : $\mathrm{MO}(\frac{w}{4}, w/2) = 1/2$; • o_3 : $\mathrm{MO}(\frac{2 \cdot w}{4}, w/2) = 0$; • o_4 : $\mathrm{MO}(\frac{3 \cdot w}{4}, w/2) = 1/2$. Since the block size is normalized to 1 unit, the four modulo offsets need to be separated by $1/4$ units to be considered well-interleaved; which is clearly not the case here.

The main problem of the offsets in Example 1 is that o_1 and o_3 yield the same modulo offset of 0 and o_2 and o_3 yield the same modulo offset of $1/2$ as the offset with respect to the block size increases by a factor of 2. Ideally, we want these modulo offsets to assume different values in any given level. As a result, we turn the problem of finding a set of n offsets that work for all levels to finding values of n such that when the offsets are well-interleaved at level z they are also well-interleaved at level $(z+1)$. In other words, the two expressions $\left\{0, \frac{1}{n}, ..., \frac{n-1}{n}\right\}$ and $\left\{0, \frac{1 \cdot 2 \mod n}{n}, ..., \frac{(n-1) \cdot 2 \mod n}{n}\right\}$ have to be equivalent.

From our investigation, we find that any odd integer value of n satisfies the condition above as formally shown in Lemma 1 and its associated proof.

LEMMA 1. *For any odd integer n, the set* $\left\{0, \frac{1}{n}, ..., \frac{n-1}{n}\right\}$ *= is equal to* $\left\{0, \frac{1\cdot 2 \mod n}{n}, ..., \frac{(n-1)\cdot 2 \mod n}{n}\right\}$.

PROOF. First, we cancel out all denominators n in the equation in Lemma 1 and try to prove for any odd integer n that $\{0, 1, ..., n-1\} = \{0, 1 \cdot 2 \mod n, ..., (n-1) \cdot 2 \mod n\}$. Note that we only want the two sets to be equal. Hence, offset i at level z does not have to correspond to $(i \cdot 2 \mod n)$ at the next level $(z+1)$. Proving that the two sets are equal can be done by showing that the function $f(i) = (i \cdot 2 \mod n)$ is a one-to-one function. In other words, for any two integers i and j in $\{0, 1, ..., (n-1)\}$ where i is not equal to j,

$$i \cdot 2 \mod n \neq j \cdot 2 \mod n. \quad (1)$$

In order for $(i \cdot 2 \mod n)$ to be equal to $(j \cdot 2 \mod n)$, the difference $|i \cdot 2 - j \cdot 2|$ has to be a multiple of n. Since both i and j are less than n, $|2 \cdot (i-j)|$ has to be less than $(2 \cdot n)$. This condition leaves us with only two options: $(0 \cdot n)$ and $(1 \cdot n)$, which are also impossible because i is not equal to j and n is an odd number, respectively. As a result, we can conclude that the inequality (1) holds for any odd number n. □

In our two-dimensional layout problem, the same set of offsets is applied to each dimension. For example, if we use three offsets in each dimension, then 9 translations are applied to each point (x, y) using translation offset tuples of $\left\{(0,0), \left(0, \frac{1\cdot w}{3}\right), ..., \left(\frac{2\cdot w}{3}, \frac{2\cdot w}{3}\right)\right\}$, where w is the block size at level 0. In this way, the degree of similarity between two locations can be expressed as a "similarity score" ranged from 0 to 9 denoting the number of translation in which they share the same Morton block. In the next subsection, we show that the same concept can also be used to represent the degree of distinctiveness of a data point among its surroundings.

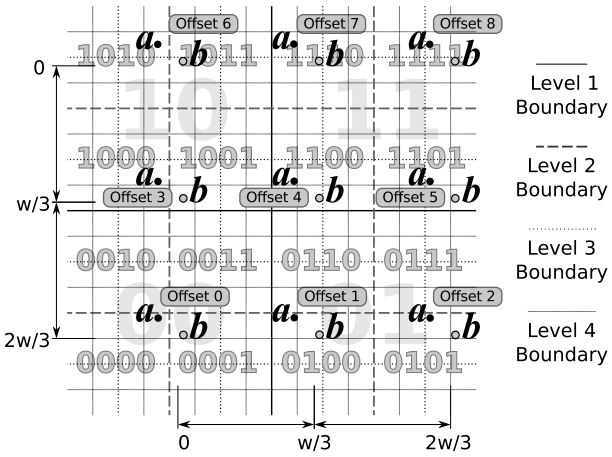


Figure 4: Illustration of how entries a and b get separated at different levels in 9 different offsets.

In Figure 4, we demonstrate how the 3×3 offsets can provide proximity detection at different levels. In this example, level 1 corresponds to the Morton code length of 2, which corresponds to the level whichs subdivides the data space into the four regions label as the Morton codes of 00, 01, 11, 10. At each level increment, the Morton code length is increased by 2 digits. Given a Morton code length l and two data entries $\boldsymbol{a}$ and $\boldsymbol{b}$, we use the notion of *proximity score* to denote the number of offsets in which $\boldsymbol{a}$ and $\boldsymbol{b}$ share the same Morton block using the Morton code length of l. At level 1 (i.e., l is 2 digits), the proximity score is 9, since $\boldsymbol{a}$ and $\boldsymbol{b}$ share the same Morton code for all offsets. As we increase the Morton code length l, we can see that $\boldsymbol{a}$ and $\boldsymbol{b}$ get separated at different levels. At the next level (l is 4 digits), we further subdivide each of the regions in level 0 into four subregions. This subdivision separates $\boldsymbol{a}$ and $\boldsymbol{b}$ for the offset numbers 0, 3, 6. Hence, the proximity score of $\boldsymbol{a}$ and $\boldsymbol{b}$ at this level is 6. At level 3, $\boldsymbol{a}$ and $\boldsymbol{b}$ are further separated by the offset numbers 7 and 8 and the proximity score becomes 4. Finally, at level 4, $\boldsymbol{a}$ and $\boldsymbol{b}$ occupy the different Morton blocks for all offsets and the proximity score becomes 0.

As we have shown, the proximity score reflects the degree of closeness between a pair of data entries based on the number of times they reside in the same Morton block for a given block size over all offsets. In the next subsection, we will make use of the same mechanism to determine the degree of distinctiveness of a data entry relative to others in its proximity.

4.2 Distinctiveness Score

In this subsection, we describe how to compute distinctiveness scores from multiple translations of the data points by evaluating a simple version of the *Multiresolution Select-Distinct (MRSD)* query. We can consider the distinctiveness score of a data entry as a vote count, where each vote indicates whether the data entry is considered distinct in a particular translation. Table 3 shows data points where each point is associated with Morton codes for the 9 translations. Two sample translations are given in Figure 5.

The main MRSD query consists of 9 subqueries where each subquery performs an independent `SELECT DISTINCT` subquery on one of the Morton orders (ordered by the importance score (IMP)). The result set consists of distinctiveness scores of data points which represents the number of times a data point is returned as a distinct result from the subqueries. Assume that the columns `id`, `ps`, and `t0` to `t8` from Table 3 are stored in a relational database table called `table_3`. An MRSD query can be evaluated using the following statement.

```
SELECT id, COUNT(*) AS ds FROM (
   SELECT DISTINCT ON (t0) id FROM table_3
            ORDER BY t0 DESC, imp DESC
   UNION ALL
   SELECT DISTINCT ON (t1) id FROM table_3
            ORDER BY t1 DESC, imp DESC
   UNION ALL
   ...
   UNION ALL
   SELECT DISTINCT ON (t8) id FROM table_3
            ORDER BY t8 DESC, imp DESC
) temp GROUP BY id;
```

The `UNION ALL` operator is used to combine results from 9 `SELECT DISTINCT` subqueries on 9 different columns `t0` to `t8`. Note that the subqueries corresponding to columns `t2` to `t7` are omitted. Each `SELECT DISTINCT` operation produces the list of entries that have the highest importance score in each Morton block (for a particular translation). The final result set contains the counts (i.e., distinctiveness scores) of entries returned by the 9 `SELECT DISTINCT` subqueries.

A distinctiveness score of 9 guarantees that if we set the image width/height ϵ for each data entry to $(2w/3)$, then no two entries with a distinctiveness score of 9 can overlap. This is because the smallest chessboard distance between two entries with a distinctiveness score of 9 is 2/3 of the Morton block size w, e.g., entries $\boldsymbol{a}$ and $\boldsymbol{d}$ in Figure 5. If one wishes to allow some overlaps and increase the number of displayed data entries, then we can set the image width ϵ to be greater than $(2w/3)$. For example, setting ϵ to w means that in the worst case, we allow a $(1/9)$ of a data entry to overlap with a data entry with a greater importance. We use this value of ϵ in our experimental studies (Section 6).

Table 3: Data points (from Figure 5) where each point is associated with a importance score (IMP) and Morton codes from 9 different translations. The resultant distinctiveness score (DS) for each data point is given in the right most column.

Object Identifier (id)	Importance Score (imp)	Morton Codes in Different Translations									Distinctiveness Score (ds)
		$(0,0)$ (t0)	$(\frac{w}{3},0)$ (t1)	$(\frac{2\cdot w}{3},0)$ (t2)	$(0,\frac{w}{3})$ (t3)	$(\frac{w}{3},\frac{w}{3})$ (t4)	$(\frac{2\cdot w}{3},\frac{w}{3})$ (t5)	$(\frac{0,2\cdot w}{3})$ (t6)	$(\frac{w}{3},\frac{2\cdot w}{3})$ (t7)	$(\frac{2\cdot w}{3},\frac{2\cdot w}{3})$ (t8)	
a	0.95	1000	1000	1001	1000	1000	1001	1000	1000	1001	9
b	0.52	1001	1001	1100	1001	1001	1100	1001	1001	1100	3
c	0.47	0010	0010	0011	1000	1000	1001	1000	1000	1001	3
d	0.80	0011	0011	0110	1001	1001	1100	1001	1001	1100	9
e	0.65	0011	0011	0110	0011	0011	0110	1001	1001	1100	1
f	0.45	0011	0011	0110	0011	0011	0110	1011	0011	0110	2
g	0.66	0011	0110	0110	0011	0110	0110	0011	0110	0110	7
h	0.14	0000	0001	0001	0000	0001	0001	0010	0011	0011	8
i	0.90	0001	0100	0100	0001	0100	0100	0011	0110	0110	9

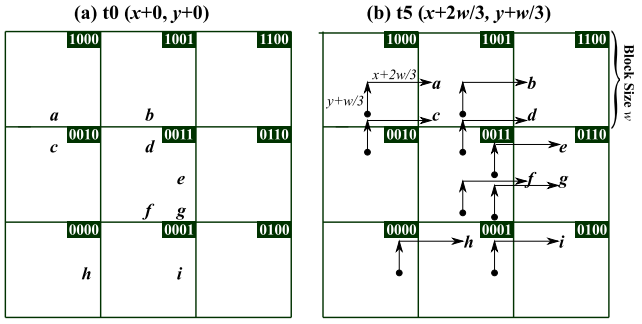


Figure 5: Data points $\{a, ..., i\}$ under translations T0 and T5 (the other translations can be computed in a similar manner).

5. INDEXING AND QUERYING

We now discuss how to set up a set of indexes to help process the distinctiveness score query described in Section 4.2. We have shown how the distinctiveness scores of data entries can be calculated using multiple `SELECT DISTINCT` subqueries. Hence, the efficiency with which these subqueries are processed is important to the overall performance of the whole query. As stated in the introduction, our indexes need to allow efficient sampling. To enable efficient sampling, our indexes need to be able to discard non-distinct entries without having to consider each entry individually. In the database context, this can be accomplished by creating a B-Tree index similar to that described in the example of Section 3. Specifically, we can create a B-Tree index on the columns holding the corresponding Morton code and importance score. This index enables us to skip to the most important entries of each group sharing the same Morton code using a *loose index scan* [18].

Note that certain RDBMSs do not directly support loose index scans on `SELECT DISTINCT` statements but provide an alternative. In PostgreSQL [4], for example, a loose index scan can be implemented using the `WITH RECURSIVE` statement, which provides us a mechanism for customized index traversal and enables us to apply arbitrary filters to index traversal process. See Appendix A for a working implementation. For conciseness, however, we continue to use the notion of `SELECT DISTINCT` to denote an operation of grouping based on one column and selecting the maximum based on another column from each group.

The described method allows us to efficiently process a `SELECT DISTINCT` subquery which covers the entire map area at the zoom level corresponding to a selected Morton code length. However, in the actual application environment, the display window may cover any portion of the total map area and can be at any zoom level. In the following subsections, we show how these concerns are addressed and how the main MRSD query is constructed.

5.1 Supporting subregion selection

Given a query bounding box (x_1, y_1, x_2, y_2), one way to use the Morton index for a subregion selection is to convert the bounding box into a Morton code range using the corners (x_1, y_1) and (x_2, y_2) to compute the starting and ending codes, respectively. Representing a 2D query window using a range on a space filling curve like the Morton order has the drawback that such a range is likely to cover many extra blocks, i.e., those outside the actual query window. In the example shown in Figure 6, the corners (x_1, y_1) and (x_2, y_2) yield a block range of [001100,110100]. We can see that (i) the number n_w of blocks that overlap with the query window is 15 and (ii) the number n_r of Morton blocks in this range is 41 (the entire length of the z-order curve shown in Figure 6). We measure the *efficiency* of a Morton code range as n_w/n_r, which is only 37% in this example.

To address the efficiency problem, we apply the quadtree decomposition principle [5, 16] to recursively split the original range into more efficient ranges. Our prefix-based splitting algorithm (Algorithm 1) accepts a starting Morton code c_s and the ending Morton code c_e and returns a list of Morton code ranges. The first step is to compute the efficiency of the input range and if it satisfies the threshold then it can be returned immediately. Otherwise, the algorithm continues. The next step is to compute the common prefix length l. If the common prefix length is an even number, then we know that the first digit that differs between the two Morton codes corresponds to the y dimension. As a result, we split the range along the x-axis to create one portion above and another portion below the axis. Otherwise, we split the range along the y-axis to create left and right portions. Within each portion the common prefix length is extended by one digit. We then continue to split each of the new ranges by recursive calls of SplitRange(). The query returns the concatenation of the results of the two SplitRange() calls.

Using the example given in Figure 6, we show that our method can improve the overall efficiency from 37% to 57% in only two splits. As exemplified in the bottom left corner of the figure assume the same block labelling scheme as that in Figure 2. For conciseness, we omit the rest of the Morton block labels. The resultant ranges of each splitting step is shown in Table 4. Assume that we set the efficiency threshold to 50%. The first split happens along the x-axis, which divides the original range [001100,110100] into [001100,011110] and [100100,110100] containing 19 and 17 blocks respectively. The first range (the bottom portion) has an efficiency of $\frac{10}{19}$, which is greater than the threshold and hence does not require a further split. The second range (the top portion) has an efficiency of $\frac{5}{17}$, which is smaller than the threshold and hence is further split into [100100,110100] into [100100,100101] and [110000,110100], with efficiencies of $\frac{2}{2} = 100\%$ and $\frac{3}{5} = 60\%$, respectively. These two splits result in three

ranges [001100,011110], [100100,100101] and [110000,110100] with an aggregate efficiency of $\frac{10+2+3}{19+2+5} = 57\%$.

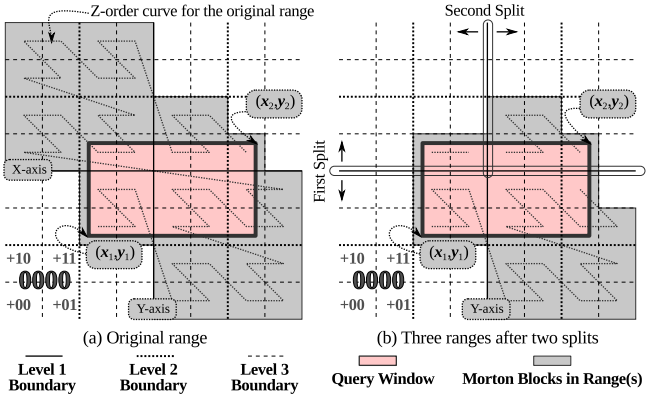


Figure 6: Representation of a query window (x_1, y_1, x_2, y_2) using multiple ranges of Morton blocks.

Algorithm 1: SplitRange(c_s, c_e)

input : Morton Code Range $[c_s, c_e]$
output : A list of ranges
env : Predefined efficiency threshold E_T

if Efficiency of the range is greater than E_T **then**
  return $[(c_s, c_e])]$;
$l \leftarrow$ Common prefix length of c_s and c_e;
if $(l \mod 2)$ is 0 **then**
  $((c_s, c_1), (c_2, c_e)) \leftarrow$ Split (c_s, c_e) along the x-axis;
else
  $((c_s, c_1), (c_2, c_e)) \leftarrow$ Split (c_s, c_e) along the y-axis;
return SplitRange(c_s, c_1) ++ SplitRange(c_2, c_e);

Table 4: Resultant ranges and efficiency for each of the two splits with the original range [001100,110100].

Split	Input Range	Resultant Ranges	Eff.	Cont.
1	[001100,110100]	[001100,011110]	53%	No
		[100100,110100]	29%	Yes
2	[100100,110100]	[100100,100101]	100%	No
		[110000,110100]	71%	No

We can also see that we can increase the threshold variable E_T if a greater efficiency is desired. For example, an efficiency threshold of 60% would result with a further split of [001100,011110] along y-axis will result in an aggregate efficiency of $\frac{4+6+2+3}{4+7+2+5} = 83\%$. The choice of the efficiency threshold E_T represents the balance between the number of Morton code ranges we need to consider and the number of extra blocks. Setting E_T too high will result in too many Morton ranges, which each incurs a separate `SELECT DISTINCT` subquery, while setting E_T too low will result in the inclusion of too many irrelevant results. In our implementation, we empirically determined that 60% is an appropriate value for E_T.

5.2 Supporting multiple zoom levels

To efficiently support multiple zoom levels, we share indexes between levels. Recall that the length of the Morton code that we use for a `SELECT DISTINCT` subquery corresponds to the level of decomposition, where each decomposition level in turn corresponds to a zoom level of a map API (e.g., Google Maps and Bing Maps). A brute-force method to handle `SELECT DISTINCT` subqueries at all zoom levels is to create multiple B-Tree indexes with different Morton code lengths for each zoom level. We devise a more space-saving solution based on the observation that the results of `SELECT DISTINCT` at a zoom level z can be computed from the results at the next zoom level $(z+1)$ by grouping blocks which share the same prefix with the length corresponding to the Morton code length of zoom level z. Let us reconsider the example in Table 3. Assume that zoom levels 0 and 1 correspond to a Morton code length of 2 and 4 respectively. According to the importance scores in the second column, the statement

```
SELECT DISTINCT ON substr(t0,1,4) id
  FROM table_3
 ORDER BY substr(t0,1,4) DESC, imp DESC
```

returns entries $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{h}$ and $\boldsymbol{i}$ for the Morton blocks 1000, 1001, 0010, 0011, 0000, 0001 at zoom level 2. We can use the returned results to find out the entry at one zoom level above by grouping these entries according to the first two digits of their Morton code, which can be done using the following statement.

```
SELECT DISTINCT ON substr(r.t0,1,2) r.id
  FROM (
    SELECT DISTINCT ON substr(t0,1,4) id, t0
      FROM table_3
     ORDER BY substr(t0,1,4) DESC, imp DESC
  ) r
```

In this case, we have two entries $\boldsymbol{a}$ and $\boldsymbol{i}$ for the Morton blocks 10 and 00 as our results at zoom level 1.

This additional prefix grouping step allows multiple zoom levels to share the same index. Note that the amount of work required to perform this additional step depends on the difference between the zoom level of the desired result and the zoom level of the index. Choosing the number of indexed zoom levels is a trade-off between the query processing time and the space used for indexes. In our implementation, we find that indexing every 3 zoom levels ensures a response time suitable for our interactive application.

We now describe how each subquery is formulated. For a given Morton code length, each subquery handles data retrieval for a Morton order and a Morton code range. We can consider a subquery as the following `SELECT DISTINCT` statement defined by four parameters. • `$1`: one of the 9 offset columns, e.g., [`t0`, `t1`, ..., `t8`] in `table_3`; • `$2`: the starting Morton code; • `$3`: the ending Morton code; • `$4`: the Morton code length.

```
SELECT DISTINCT ON (substr((d).$1,1,$4)) id
FROM
(
 SELECT DISTINCT ON (substr($1,1,idx_len($4)))
        id, substr($1,1 idx_len($4))
   FROM table_3
  WHERE substr($1,1,idx_len($4))
        BETWEEN $2 and $3
  ORDER BY substr($1,1,idx_len($4)) DESC,imp DESC
) as f
```

Assume that `idx_len()` is a predefined function which accepts a Morton code length and returns the Morton code length corresponding to the next indexed level. An equivalent of this query statement using `WITH RECURSIVE` is given in Appendix A.

5.3 Main MRSD Query

The main query combines results from multiple subqueries and counts the occurrences of each entry, which may appear at most once per offset. Algorithm 2 illustrates how the main query statement is constructed as the union of multiple subqueries defined in the previous subsection (line 11). The main part of the statement (lines 1 and 10), i.e., the distinctiveness score calculation part, is identical to the basic query statement given in Section 4.2. At line 2, we calculate the Morton code length corresponding to the proximity threshold ϵ (which is given by the width of each image relative to the entire map width). Formally, in a data space of one unit square, the Morton code length l corresponds to a Morton block width w of (2^{-l}). As discussed in Section 4.2, we set the proximity threshold ϵ to w. Hence, the Morton code length l is calculated as $(-\lfloor \log_2 \epsilon \rfloor)$. The algorithm loops though the 9 offset columns. At each iteration, we compute Morton code ranges corresponding to the query window using Algorithm 1 described in Section 5.1.

Algorithm 2: CreateMainMRSDQuery((x_1, y_1, x_2, y_2), ϵ)

```
   input  : Query window (x1, y1, x2, y2), Proximity threshold ε
   output : A SQL statement st
1  st ← "SELECT id, count(*) as ds FROM (";
2  l ← Compute the Morton code length as (−⌊log2 ε⌋);
3  foreach col in ["t0"..."t8"] do
4      (codes, codee) ← Compute Morton code range for
       (x1, y1, x2, y2) with respect to the offset of col;
5      ranges ← SplitRange(codes, codee) ;
6      foreach (cs, ce) in ranges do
7          st ← st ++ "(" ++ Subquery(col, cs, ce, length) ++ ")";
8          if not last subquery then
9              st ← st ++ "UNION ALL";
10 st ← st ++ ") temp GROUP BY id";
11 return st;
```

6. EXPERIMENTS

In this section, we compare two image layout solutions.

- Our multiresolution select distinct (MRSD) query, which (i) performs a loose index scan on each of the 9 Morton orders to find the most important entry in each Morton block in the query window, and then (ii) counts the total number of times each entry occurs as the distinctiveness score of that entry.
- A window-query-based (WQB) method, which (i) retrieves all data entry in the query window, and then (ii) performs proximity check to prune data entries that are within a proximity threshold of ϵ from a more important entry.

We used two real datasets downloaded from `cloudmade.com`. Our first dataset contains 1,793,622 locations of *tourist destinations (TD)* around the world. Our second dataset contains 9,964,607 locations of general *points of interest (POI)* around the world. In their original form, these datasets did not contain scoring information of how important each data entry is. As a result, we assigned an importance score for each data entry as a random floating point value between 0 and 1. In order to obtain a realistic set of geographic query windows, we collected a sample of 2,000 query windows from `newsstand.umiacs.umd.edu`, a system for browsing news articles geographically [6, 10, 15] (see also the related TwitterStand system [12]). We transform the coordinates of the queries and data points using the Mercator projection for compatibility with Google Maps and Bing Maps. For ease of exposition, all coordinates are normalized onto the unit square. The query size distribution of our query set is given in Figure 7.

We use the example context of web-based mapping applications to determine the relationships between the query window size S_Q and other parameters. In particular, for each query window with a size S_Q, we calculate the corresponding zoom level z, proximity threshold ϵ and Morton code length l. The relationships between these parameters are summarized in Table 5 and can be worked out as follows. Let us first establish the corresponding zoom level z for a given value of S_Q. Using the Google or Bing Maps API, at the zoom level z of 0, the map size is (256×256) pixels. At each zoom level increment, the number of pixels is increased by a factor of 2 for each dimension. That is, the total map size is equal to $(256 \cdot 2^z \times 256 \cdot 2^z)$ pixels. This means that a typical display window size of 900×900 pixels would cover 77% of the total map area at the zoom level z of 2. As a result, we associate the first bin $[2^{-2}, 2^0]$ with the zoom level z of 2. Each bin in Figure 7 corresponds to one zoom level increment.

The relationship between the zoom level z and the Morton code length l depends on the image size. Each location is represented as an image thumbnail with a typical size of (128×128) pixels. At the zoom level z of 2, the image width of 128 pixels in a total map width of 1024 pixels is normalized to 0.125 units. For each zoom level increment, the normalized image width is reduced by a factor of 2 due to the increase in the total pixel width of the map. We use this normalized pixel width as the proximity threshold ϵ. As described in Section 4.2, we want to match the proximity threshold ϵ with the Morton block size. In a unit square, ϵ of 0.125 units is equal to the width of Morton blocks in a $(2^3 \times 2^3)$ grid, which requires the Morton code length l of 6 digits to identify each grid cell. Hence, we can conclude that z of 2 corresponds to l of 6 digits. And at each zoom level increment, the length l is increased by 2 digits as ϵ is reduced by a factor of 2.

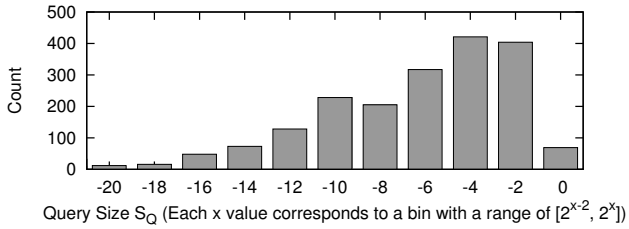


Figure 7: Distribution of query windows by query size

Table 5: Relationships between the query size S_Q, zoom level z, proximity threshold ϵ and Morton code length l.

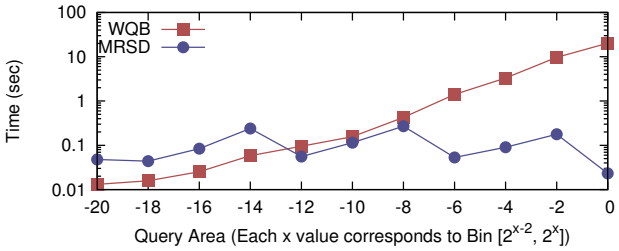
Query size S_Q	Zoom Level z	Proximity Thresholds ϵ	Code Length l
$[2^{-2}, 2^0]$	2	2^{-3}	6
$[2^{-4}, 2^{-2}]$	3	2^{-4}	8
⋮	⋮	⋮	⋮
$[2^{-26}, 2^{-24}]$	14	2^{-15}	30

Experiments were conducted on an Intel i7-2720QM @ 2.20 GHz with 16GB RAM. We used PostgreSQL 9.1.4 to store the data entries and to index the Morton codes. For the MRSD query, we index the Morton code lengths of 6, 12, 18, 24 and 30 digits, which correspond to the zoom levels of 2, 5, 8, 11 and 14, respectively. For WQB, we used the spatial index in the PostGIS extension to index the coordinates of the data entries to accommodate window query processing. Query statements were prepared in a Java program connecting to a local PostgreSQL server via JDBC 4.1.

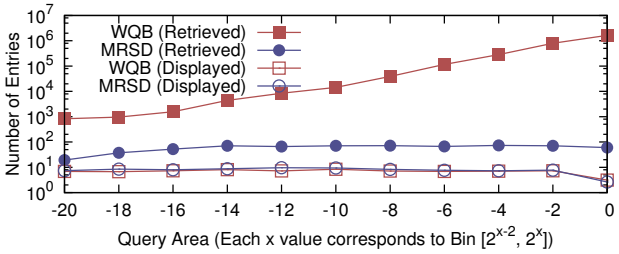
6.1 Performance

We compared MRSD to WQB using the three following measures. The first measure is the query response time. For MRSD, the query response time includes the time taken to convert a query window into ranges of Morton codes, statement preparation time, and the actual query processing time. For WQB, the query response time includes the window query statement preparation, data retrieveal and overlap removal. The second measure is the number of entries returned from the query. For WQB, this measure corresponds to the number of entries in the query window. For MRSD, this measure corresponds to the number of data entries returned from the main MRSD query, which includes data entries of all distinctiveness scores 1 to 9. The third measure is the number of displayed entries. WQB selects entries to display by iterating through all retrieved entries to find entries $\boldsymbol{p}$ such that (i) does not have other entries nearby (entries within a proximity threshold of ϵ); or (ii) all nearby entries are less important than $\boldsymbol{p}$. For MRSD, we display only data entries with a distinctiveness score of 9 (the hightest possible score) to minimize the overlaps between thumbnails.

Figure 8(a) shows the total response time for the TD dataset. We can see that the performance of WQB drastically degrades as the query size increases. This is because an increase in the query size

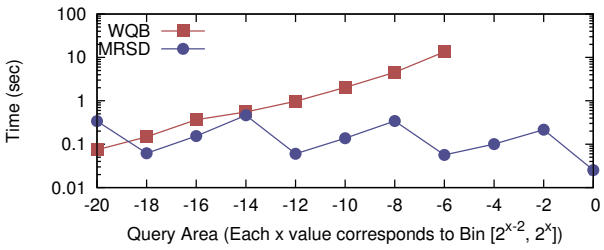


(a) Query response time (TD)

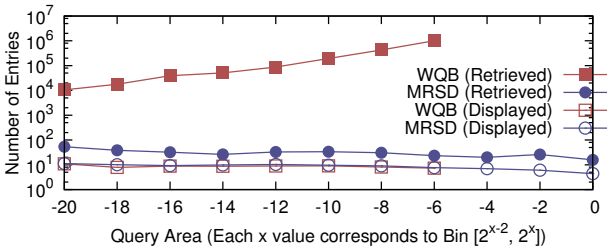


(b) #Retrieved and #displayed entries (TD)

Figure 8: Results for the TD dataset (1.79 million entries)



(a) Query response time (POI)



(b) #Retrieved and #displayed entries (POI)

Figure 9: Results for the POI dataset (9.96 million entries). Note that WQB failed to produce results for the last three bins due to excessive memory consumption.

provides more data entries for WQB to retrieve and consider, while the image size restriction keeps the number of displayed images stable as shown in Figure 8(b). In other words, for a large query window, WQB retrieves much more than what is really needed to display. The performance of MRSD, on the other hand, is unaffected by changes in the query size. As the query size S_Q increases and the zoom level decreases, the size of Morton blocks increases at the same rate as S_Q. Consequently, the number of Morton blocks we have to consider remains unchanged relative to S_Q. MRSD uses a loose index scan to prune data entries that are in proximity of more important ones for each Morton block in range. Hence, the query cost of MRSD depends on the number of considered Morton blocks in the indexes rather than the query size S_Q. Figure 8(a) also shows the sawtooth pattern of the total response time with troughs (local minimums) at bins $[2^{-2}, 2^{-0}]$, $[2^{-8}, 2^{-6}]$, $[2^{-14}, 2^{-12}]$ and $[2^{-20}, 2^{-18}]$. This is because these bins correspond to the indexed zoom levels of 2, 5, 8 and 11 where query results can be directly obtained from one of the indexes. To obtain results from a non-indexed level z_1, we have to combine results from the next indexed level z_2. The number of Morton blocks we have to consider increases as the difference between z_1 and z_2 increases.

Figures 9(a) and 9(b) show results from the POI dataset, which is approximately 5.6 times as large as TD. A greater density of data entries means that WQB has to retrieve more data entries for each window query as shown in Figure 9(b). Figure 9(a) shows that the total response times of WQB for the POI dataset are almost one order of magnitude greater than the corresponding response times for TD due to the increase in the number of data entries. The response time of MRSD, on the other hand, remains unchanged in comparison to that of the TD dataset. This again demonstrates the usefulness of our indexes which allow us to prune data entries that are in proximity of more important ones.

6.2 Layout Consistency

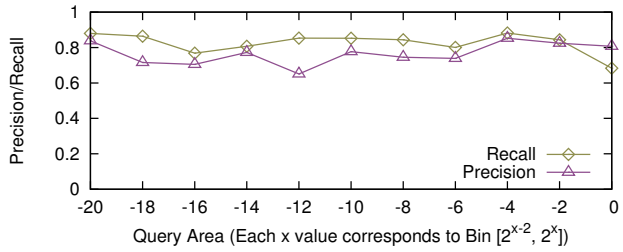
We use *precision* and *recall* to measure the layout consistency of MRSD with respect to WQB. Let n_m be the size of the MRSD result set, n_w be the size of the WQB result set, and n_i be the size of the intersection of the two result sets. Then precision is n_i/n_m and recall is n_i/n_w. In this case, precision indicates how often MRSD results are truly non-overlapping, and recall indicates how often truly non-overlapping entries are returned by MRSD.

Figures 10(a) and 10(b) shows the precision and recall of the TD and POI datasets. The fact that the precision is approximately 75% indicates that around 25% of the results produced by MRSD are those that overlap with more important ones. However, as discussed in Section 4, each of these overlaps can occupy at most $\frac{1}{9}$ of the thumbnail area representing the data entry. The fact that the recall is approximately 85% means that around 15% of the non-overlapping results returned by WQB are not included in the result set of MRSD. One way to improve the recall is to reduce the distinctiveness score threshold from 9 to a lower value. However, this may negatively affect the precision by introducing more overlaps. Setting the value of distinctiveness score threshold is therefore a trade-off between precision and recall, or between overlap minimization and spatial fullness maximization, respectively. On the other hand, if we wish to improve the precision (i.e., reducing overlapped entries), we can increase the proximity threshold ϵ by decreasing the corresponding Morton prefix length at each level. As discussed in Section 4.1, by ensuring that the image width/height ϵ is not greater than two thirds of the block width w, we can completely eliminate overlaps and hence obtain a precision of 100%.

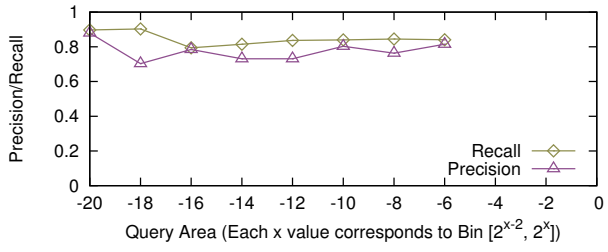
7. CONCLUDING REMARKS

We proposed an efficient layout solution for selecting spatially distinct data entries from a large geographic point set. The novelty of our method lies in an ensemble of interrelated indexes, and a query algorithm which performs separate `SELECT DISTINCT` subqueries on these indexes and combines the results of these subqueries to determine the degree of spatial distinctiveness of each entry in a query window.

Our proposed method satisfies the design objectives given in Section 1. Specifically, our measure of distinctiveness reflects the



(a) Precision and recall for the TD dataset



(b) Precision and recall for the POI dataset

Figure 10: Precision and recall. Note that for the POI dataset, the results for the last 3 bins are omitted, since WQB failed to produce results in those query sizes.

importance of each data entry relative to its surroundings. We can use it to remove overlaps between image thumbnails representing the data entries by setting a distinctiveness threshold to discard data entries with a low distinctiveness score, since these entries are likely to be in proximity of a more important one causing their image thumbnails to overlap. The design of our distinctiveness measure ensures that it satisfies the first three objectives, namely it minimizes overlaps, respects relative importance of entries, and maximizes spatial fullness. With regard to the fourth objective, our method also provides panning and zooming consistency, since the distinctiveness score of each entry does not change when panning and can only increase when zooming in. As discussed in Section 5 and demonstrated by the experimental results in Section 6, our method supports efficient index scans hence it satisfies the fifth objective. Appendix A shows how we can incorporate filtering conditions (the sixth objective) into our query processing method.

We compared our solution to a competitive window-query-based method. For a dataset of more than 9 million data points, we showed that our solution is capable of providing a respond time suitable for interactive applications where most query results are returned in less than a second. The competitive method on the other hand spends more than 4 seconds as the query window starts to occupy more than 2^{-8} of the total map area and the query response time consistently increases as the query window size increases. The experimental results showed that our solution is more scalable with respect to the sizes of the query window and dataset and produces results comparable to the window-query-based method.

ACKNOWLEDGEMENTS. This work was supported in part by the National Science Foundation under Grants IIS-07-13501, IIS-08-12377, CCF-08-30618, IIS-09-48548, IIS-10-18475, and IIS-12-19023.

8. REFERENCES

[1] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *PODS*, pages 265–272, 1990.
[2] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *CIKM*, pages 347–354, 1994.
[3] S. Doddi, M. V. Marathe, A. Mirzaian, B. M. E. Moret, and B. Zhu. Map labeling and its generalizations. In *SODA*, pages 148–157, 1997.
[4] T. P. G. D. Group. *Postgresql 8.4 Official Documentation - volume iii. Server Programming*. Fultus Corporation, 2009.
[5] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.
[6] H. Samet, M. D. Adelfio, B. C. Fruin, M. D. Lieberman, and B. E. Teitler. Porting a web-based mapping application to a smartphone app. In *GIS*, pages 525–528, 2011.
[7] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *Commun. ACM*, 46(1):61–64, 2003.
[8] H. Samet, B. C. Fruin, and S. Nutanong. Duking it out at the smartphone mobile app mapping api corral: Apple, Google, and the competition. In *MobiGIS*, 2012.
[9] H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber. A geographic information system using quadtrees. *Pattern Recognition*, 17(6):647–656, 1984.
[10] H. Samet, B. E. Teitler, M. D. Adelfio, and M. D. Lieberman. Adapting a map query interface for a gesturing touch screen interface. In *WWW (Companion Volume)*, pages 257–260, 2011.
[11] J. Sankaranarayanan and H. Samet. Images in news. In *ICPR*, pages 3240–3243, 2010.
[12] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. Twitterstand: news in tweets. In *GIS*, pages 42–51, 2009.
[13] A. D. Sarma, H. Lee, H. Gonzalez, J. Madhavan, and A. Y. Halevy. Efficient spatial sampling of large geographical tables. In *SIGMOD Conference*, pages 193–204, 2012.
[14] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *IJGIS*, 4(2):103–131, April–June 1990.
[15] B. E. Teitler, M. D. Lieberman, D. Panozzo, J. Sankaranarayanan, H. Samet, and J. Sperling. Newsstand: a new view on news. In *GIS*, pages 18:1–18:10, 2008.
[16] H. Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Angewandte Informatik*, 23(2):71–77, 1981.
[17] F. Wagner and A. Wolff. An efficient and effective approximation algorithm for the map labeling problem. In *ESA*, pages 420–433, 1995.
[18] M. Widenius and D. Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

Appendix A.

To enable a *loose index scan* in PostgreSQL, the final `SELECT DISTINCT` statement in Section 5.2 is rewritten as a `WITH RECURSIVE` statement. The query performs an index scan on the specified column (`$1`) starting at the first Morton code (`$2`) and terminating at the last one (`$3`) for a given Morton code prefix length (`$4`). Note that here we introduce a fifth parameter (`$5`), which can be used as a filtering condition for the index scan.

```
SELECT DISTINCT ON (substr((d).$1,1,$4)) id FROM
  WITH RECURSIVE t AS (
    SELECT d FROM (
      SELECT d FROM poi_table d
       WHERE substr($1, 1, idx_len($4)) <= $3
         AND $5
       ORDER BY substr($1, 1, idx_len($4)) DESC,
             score DESC
       LIMIT 1
    ) q
    UNION ALL
    SELECT
      (SELECT di FROM poi_table di
        WHERE (substr(di.$1, 1, idx_len($4))
              > substr((t.d).$1, 1, 6))
          AND substr(di.$1, 1, idx_len($4)) >= $2
          AND $5
        ORDER BY substr(di.$1,1,idx_len($4)) DESC,
              di.score DESC
        LIMIT 1)
      FROM t
     WHERE d IS NOT NULL
  )
  SELECT d FROM t WHERE d IS NOT NULL
) AS f;
```