The VLDB Journal (2015) 24:537–555
DOI 10.1007/s00778-015-0388-z

REGULAR PAPER

Finding top-k relevant groups of spatial web objects

Anders Skovsgaard[1] · **Christian S. Jensen**[2]

Received: 4 November 2013 / Revised: 22 February 2015 / Accepted: 5 May 2015 / Published online: 16 June 2015

Abstract The web is increasingly being accessed from geo-positioned devices such as smartphones, and rapidly increasing volumes of web content are geo-tagged. In addition, studies show that a substantial fraction of all web queries has local intent. This development motivates the study of advanced spatial keyword-based querying of web content. Previous research has primarily focused on the retrieval of the top-k individual spatial web objects that best satisfy a query specifying a location and a set of keywords. This paper proposes a new type of query functionality that returns top-k groups of objects while taking into account aspects such as group density, distance to the query, and relevance to the query keywords. To enable efficient processing, novel indexing and query processing techniques for single and multiple keyword queries are proposed. Empirical performance studies with an implementation of the techniques and real data suggest that the proposals are viable in practical settings.

Keywords Query processing · Indexing · Spatial web object · Histogram · Group of objects · Aggregate information

1 Introduction

Information on the Internet is increasingly acquiring a spatial aspect, with still more types of content being geo-tagged, such as points of interest, status updates from social network users, and web pages [1,9,21]. Coupled with the increasing mobile use of the web from geo-positioned devices such as smartphones, it has become relevant to explore a new kind of spatial web query that takes into account both a user location and user-supplied keywords when returning geo-tagged web content, called spatial web objects.

This paper considers a new type of query that targets the type of situation where a user wishes to explore colocated, nearby places that match the supplied query keywords. This browsing behavior typically occurs when people shop or when tourists visit new cities. The shopper may know the type of item of interest, e.g., shoes, but not the specific place to find such items. Most likely, the shopper will want to visit several shoe shops before deciding which shoes to buy. A tourist visiting a new city may not know any interesting places to eat. Being able to browse a number of places of interest is important to get an impression of the options.

Already today, many of the queries performed by users concern local objects. Google reports that 20 %[1] of all searches have local intent, while Microsoft finds that 53 % of all mobile searches have local intent[2]. With approximately 1 billion tourists per year (2010 numbers) spending more than 900 billion US dollars[3], this new type of query is well motivated.

To support the browsing behavior, the query returns k groups of objects where the objects in a group are close to each other and the query location, and where each object is relevant to the query keywords.

✉ Anders Skovsgaard
anderssk@cs.au.dk

Christian S. Jensen
csj@cs.aau.dk

[1] Department of Computer Science, Aarhus University, Århus, Denmark

[2] Department of Computer Science, Aalborg University, Aalborg, Denmark

[1] https://sites.google.com/a/pressatgoogle.com/googleplaces/metrics.

[2] http://searchengineland.com/microsoft-53-percent-of-mobile-searches-have-local-intent-55556.

[3] http://www.bbc.co.uk/schools/gcsebitesize/geography/tourism/tourism_trends_rev1.shtml.

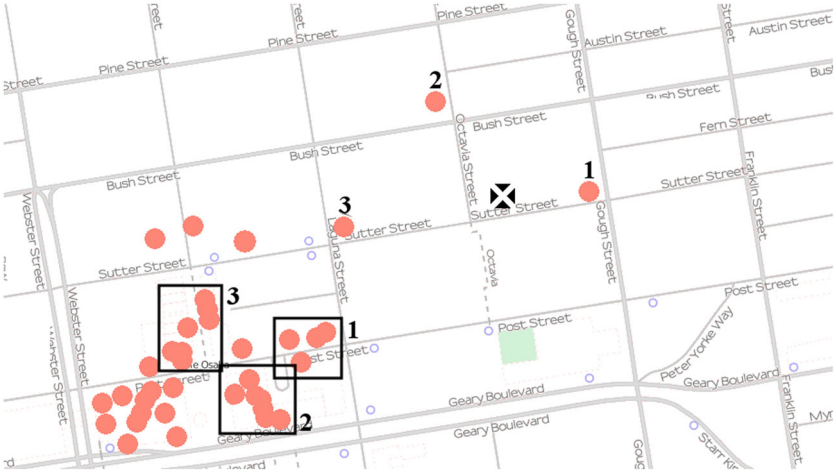


Fig. 1 Restaurants near Kenmore Hotel, San Francisco

Figure 1 shows a real-world example where a tourist located at a hotel in San Francisco issues the query "restaurant" in preparation for exploring the neighborhood for a good place to eat. Consider the case where a list of k individual restaurants is returned. The top-1 restaurant may be closed, the top-2 restaurant may be too expensive, and the food may look too spicy at the top-3 restaurant; and these restaurants may be located relatively far apart. In the figure, the three restaurants (dots) nearest to the query (the cross) are indeed located in different directions from the query. This makes it a potentially tedious exercise to explore multiple restaurants before making a dining decision.

The query we propose identifies groups of places that are relevant to the query keywords such that the places in a group are near each other and such that the groups are near the query location. The query returns a list of k such groups. With k set to 3, the query in the example returns the three dense groups of restaurants seen in the lower left part of the figure. By visiting one or more of these groups, the tourist can conveniently explore multiple restaurants before making a decision.

In contrast to the proposed query functionality, most existing spatial keyword queries return a list of k individual objects, where each object in isolation satisfies the query [4,7,8,18,19,23,26,30,31]. Only four studies in the literature are known to the authors that do not return single objects, but rather return an aggregate result [5,20,28,29], namely a result consisting of a single set of objects that collectively cover the query keywords. Next, Google Maps can display objects that match a given keyword. But as only locations are displayed on the map, users cannot easily assess the textual relevancies of the objects. The user also cannot take into account objects outside the visible map borders. We cover related work and baseline approaches in Sect. 5.

We offer a precise definition of a carefully designed top-k relevant groups spatial keyword query and then propose indexing and query processing techniques that aim to enable efficient computation of the query. In doing so, we build on existing state-of-the-art indexing techniques for spatio-textual data. This involves mapping from terms to an R-tree. Specifically, we extend a spatial inverted index to include compressed histograms in each node entry in order to obtain compact descriptions of the objects contained in subtrees, which then enables effective search space pruning.

We present an algorithm capable of building and maintaining the compressed histograms in the index structure, and we present query processing algorithms that are able to prune the search space by utilizing the compressed histograms. The algorithms work without spatial ranges that limit the search space and guarantee correct results that take into account the entire dataset. They calculate the spatial distance to a group and use the compressed histograms to calculate the diameter and textual relevance of a group.

The proposed techniques are evaluated on real data collected from Google Places [11]. Three datasets are used: one that covers San Francisco, one that covers Las Vegas, and one that covers a synthetic city where objects from four major US cities have been shifted to the same region. We study different parameter settings and offer insight into the impact on the properties of the resulting groups along with the performance in terms of runtime and page accesses. The study suggests that the proposed solution is capable of finding relevant groups while maintaining good performance.

The contributions are as follows: (i) a novel type of query that exploits the locations of objects to return aggregate results, namely groups of relevant spatial web objects; (ii) a new indexing technique for spatial web objects; (iii) query processing algorithms that exploit the indexing technique; and (iv) an empirical performance study of the proposed techniques that suggests that these are applicable in realistic, practical settings.

The rest of the paper is organized as follows. Section 2 formally defines the problem. Section 3 presents the new index

structure and query processing algorithms along with two enhancements. The empirical study is covered in Sect. 4. Finally, Sect. 5 covers baseline approaches and gives an overview of related work, and conclusions and future work are presented in Sect. 6.

2 Problem design

Intuitively, given a set of spatial web objects with spatial coordinates and text descriptions, the top-k groups spatial keyword query takes a location and a set of query keywords as arguments and returns k groups of objects such that the objects in a group are close to each other, the group is close to the query location, and the objects in the group are textually relevant to the query keywords. A number of definitions are needed to formalize this setting and query.

The spatial web objects being queried are objects with a geographical point location and a textual description. One type of such objects is web pages for places such as restaurants, attractions, and public offices that may be given meaningful locations. Another type is listings of business in online directories such as Google Places.

We let D be the set of spatial web objects o, where each object is a pair (λ, ϕ) such that $o.\lambda$ is o's location and $o.\phi$ is o's textual description. Specifically, we assume that the location is a point location in two-dimensional Euclidean space.

Similarly, a top-k groups query q takes a triple of arguments (λ, ϕ, k), where $q.\lambda$ is a point location, $q.\phi$ is a set of keywords, and $q.k$ is the number of groups in the result. Intuitively, the former is the querying user's location, and the latter is keywords entered by the user.

The next step is to define the spatial and textual relevance notions used in the query. First, we define the distance between a query location λ and a group G of objects as the distance between the query location and the nearest object in the group: $d(\lambda, G) = min_{o \in G} ||\lambda, o.\lambda||$, where $||\cdot, \cdot||$ denotes the Euclidian distance.

Intuitively, a group of relevant objects that are close to each other is more attractive than a group of equally relevant objects that are further apart. We define the spatial compactness of a group G as the largest inter-object distance between any two objects in the group: $diameter(G) = max_{o_1, o_2 \in G} ||o_1, o_2||$.

Another natural definition might be the area of the convex hull of the location of the objects in a group. However, if the objects tend to be aligned along a straight line, it is possible for objects in a group with small area to be far apart. Our definition of diameter avoids such groups.

Numerous measures exist for evaluating textual relevance, and the techniques presented in this paper are applicable to many of these. In this paper, we employ the state-of-the-art language models developed by Ponte and Croft [22], which are also used in related work by Cong et al. [7]. Here, a text document is represented by a vector where each dimension corresponds to a distinct term in the document. The relevance of an object o to a query term, $t \in q.\phi$, is defined as follows:

$$TR(t, o.\phi) = (1 - \gamma)\frac{tf(t, o.\phi)}{|o.\phi|} + \gamma\frac{tf(t, Coll)}{|Coll|}, \quad (1)$$

where $tf(t, o.\phi)$ is the number of occurrences of term t in $o.\phi$, $tf(t, Coll)$ is the count of term t in the collection $Coll$ of D, and γ is a smoothing parameter.

The above equation defines the relevance of a single object to a query term. The following definition of group proximity extends the definition to apply to all keywords, or terms, in a query and groups of objects.

The underlying idea is to give more weight to larger groups of relevant objects and to give more weight to groups with several relevant objects for each query term that represents a separate entity. For example, a query may contain "restaurant" and "cinema," in which case a good result group should contain multiple objects for each of "restaurant" and "cinema." In contrast, the term "Chinese restaurant" represents a single entity. This entity may then be combined with, e.g., a "cinema" entity, yielding a query that returns groups that each contains multiple objects relevant to Chinese restaurant and cinema.

$$GP(q.\phi, G) = \prod_{t \in q.\phi} \left(\left(\left(\sum_{o \in G^t} TR(t, o.\phi) \right) + 1 \right) \cdot |G^t| \right)^{-1}, \quad (2)$$

where G^t is the objects in G containing the term t.

The equation sums up the textual relevance of each object and adds 1 to ensure that the final result is a value between 0 and 1. The sum yields the total textual relevance of the group without taking into account the number of objects. The lower the score, the better the proximity.

Next, we wish to give preference to groups where the query terms are distributed relatively evenly among objects (rather than a single term being contained in many objects and other terms being contained in few objects). This is achieved by multiplying the sum for each query terms by the number of objects involved—the more evenly terms are distributed, the larger this product becomes. An example is given in Example 1.

If a group contains no objects for some query term, the score is undefined because of division by zero. We will only compute scores for which this does not occur. Also, we note that the following proposals remain valid when using alternative definitions of group proximity. The definition given

Table 1 Scores for groups with different term distributions

O_{t_1}	O_{t_2}	O_{t_3}	$GP(q.\phi, G)$
10	1	1	0.041
4	4	4	0.006

here provides good results on real datasets, to be shown in Sect. 4.2.

Example 1 We consider two datasets that each contains 12, each of which contains one of three different terms, t_1, t_2, and t_3. An object containing term t_i is denoted $O_{t_i}, i \in [1, 3]$. As shown in Table 1, the group consisting of objects with terms that are evenly distributed has the best (i.e., lowest) score. □

The score, or cost, $Cost(q, G)$ of a group G with respect to a query q takes into account both the spatial and textual characteristics of the objects in the group.

As in other works, we introduce a parameter α that makes it possible to control the importance of the spatial properties versus the textual relevance. To further be able to control the importance of the spatial distance between a query and a group versus the diameter of the group, we introduce a parameter β. For example, this parameter can be used for the modeling of situations where a user wants to drive by car to reach a group, but then wants to visit the objects by foot. In this case, a relatively long distance to the group is acceptable, while the diameter should be small. Our prototype[4] uses parameters α and β to provide settings that provide results that are convenient for driving, bicycling, and walking. The three options can be selected both for travel to the groups and for movement within the groups. This way, a user can specify different means of transportation, e.g., traveling by car to the group and then walking between the objects in the group. The parameters α and β are not exposed to the user.

As a result of these design considerations, we get the following ranking function for groups.

$$Cost(q, G) = \alpha \frac{\beta d(q.\lambda, G) + (1-\beta) diameter(G)}{max D} + (1-\alpha) GP(q.\phi, G), \quad (3)$$

where $maxD$ is the largest Euclidean distance of the underlying space.

Parameter $\alpha \in [0, 1]$ enables balancing spatial proximity and group proximity. An α close to 1 means that spatially nearby groups are preferred, while with α close to 0, textual relevance is given priority. In fact, $\alpha = 0$ gives the lowest cost to the group of all objects in D that contain at least one of the query keywords. Setting parameter $\beta \in [0, 1]$ close to 0 favors dense groups, whereas setting it close to 1 assigns importance instead to the distance between the query and the group.

With the above definitions in place, we can define the result of the top-k groups spatial keyword query.

Definition 1 *Top-k groups spatial keyword query* A top-k groups spatial keyword query q takes a set D of spatial web objects as an argument together with three parameters (λ, ϕ, k). It returns k subsets $G_1, \ldots G_k$ of D such that $G_i \subseteq D_i$, $i = 1, \ldots, k$, where $D_1 = D$ and $D_i = D \backslash \cup_{j=1}^{i-1} G_j$, $i = 2, \ldots, k$, and such that there does not exist a subset $G \subseteq D_i$ for which $Cost(q, G) < Cost(q, G_i)$, $i = 1, \ldots, k$.

The definition ensures that the groups of objects in a result are disjoint. This avoids results with near-identical groups. The definition guarantees that when G_i is computed, no better group can be computed from the objects available.

The definition involves parameters that may be used to accommodate different means of transportation, e.g., driving, biking, and walking. A user may not know how to set the values of the parameters in order to get the most relevant result. In our demonstration paper [2], we suggest settings for the parameters that yield results that are suitable for different means of transportation. More specifically, values for α and β are chosen such that a user may select one form of transportation to get to the group and another when moving inside the group.

Example 2 Figure 2 gives the locations of eight spatial objects. We set $\alpha = 0.4$ and $\beta = 0.4$, and for simplicity, we assume that the textual description of each object consists of a single term. Costs needed for computing top-k queries are presented in Table 2. For example, the cost for $G = \{o_1\}$ is

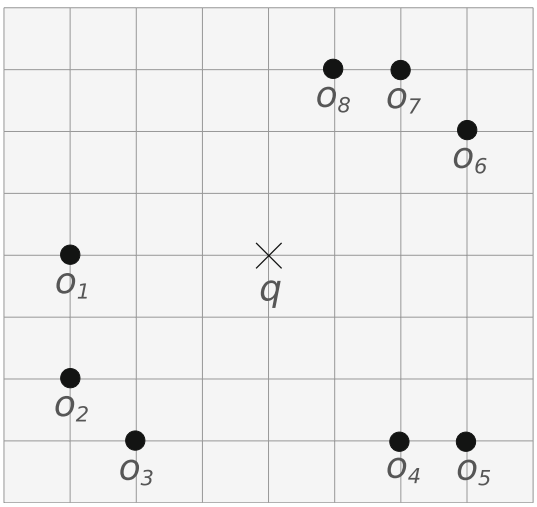


Fig. 2 Objects and *MBR*s

[4] http://cs.au.dk/~anderssk/groupfinder/.

Table 2 Function values

G	$d(q.\lambda, G)$	$diameter(G)$	$GP(q.\phi, G)$	$Cost(q, G)$
o_6, o_7, o_8	3.16	2.24	0.08	0.199
o_7, o_8	3.16	1.0	0.17	0.207
o_4, o_5	3.61	1.0	0.17	0.217
o_1, o_2, o_3	3.0	3.16	0.08	0.227
o_2, o_3	3.61	1.41	0.17	0.231
o_1, o_2	3.0	2.0	0.17	0.24
o_1, o_2, o_3, o_4	3.0	5.83	0.05	0.3
o_1, o_6, o_7, o_8	3.0	6.32	0.05	0.32
o_1	3.0	0.0	0.5	0.37
o_8	3.16	0.0	0.5	0.37
o_4	3.61	0.0	0.5	0.38

calculated as follows:

$$Cost(q, G) = 0.4 \cdot \frac{0.4 \cdot 3 + (0.6) \cdot 0}{7} + 0.6 \cdot ((1+1) \cdot 1)^{-1} = 0.37$$

The top-3 groups are $G_1 = \{o_6, o_7, o_8\}$, $G_2 = \{o_4, o_5\}$, and $G_3 = \{o_1, o_2, o_3\}$. Notice that $\{o_7, o_8\}$ has a lower cost than G_2 and G_3, but is excluded because it is a subset of G_1. Setting $\beta = 0.2$ changes the order of G_2 and G_3, while G_1 remains the top-1 result. □

3 Proposed solution

Before presenting the proposed techniques, Sect. 3.1 presents a solution overview. Section 3.2 describes the proposed index structure along with related definitions. Section 3.3 covers the compressed histograms used in the index, and index construction is covered in Sect. 3.4. Finally, Sect. 3.6 presents the query processing algorithms for single and multiple keyword queries.

3.1 Solution overview

We proceed to give an overview of the proposed solution. Initially, the data are preprocessed such that each term (or entity) of the object documents is inserted into a vocabulary, as shown in Fig. 3. Each term is assigned to its own index structure, a Group Extended R-Tree, that maintains information about the objects' documents and their inter-object distances. Specifically, each index entry that represents as subtree in this tree has a compressed histogram that stores this information for its subtree. Detailed descriptions of the creation of Group Extended R-Trees and compressed histograms are provided in the following section.

Next, query processing occurs as shown in the lower part of the figure. First, the query keywords are preprocessed,

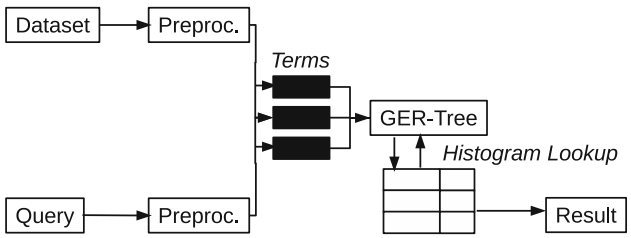


Fig. 3 Solution overview

e.g., to identify terms that represent entities, and each resulting term is used to perform a lookup in the vocabulary. The Group Extended R-Tree is then searched, while the information about inter-object distances and textual relevancy in the compressed histograms is used to prune sub-trees efficiently. The query processing is detailed in Sect. 3.6.

3.2 Group Extended R-Tree

We proceed to present a new indexing technique that makes it possible to compute an exact query result while still being able to prune the search space substantially.

Overall, the indexing technique encompasses a vocabulary and a collection of Group Extended R-Trees (GER-trees). First, the *vocabulary* associates each term in a data object in the database with a separate GER-tree. Thus, a GER-tree exists for each term, and the GER-tree for a term indexes all objects that contain the term in their textual description. Each entry in a non-leaf GER-tree node contains a *compressed histogram* that describes its subtree.

Table 3 gives an example of the vocabulary that maps terms to GER-trees. Parts used to compute the second term of Eq. (1) and the total number of objects are maintained together with the mapping to facilitate fast lookup.

A GER-tree resembles an R-tree. Leaf node entries are of the form $(mbr, cp, tf(t, o.\phi)/|o.\phi|)$, where mbr is the minimum bounding rectangle of the object, and cp is a pointer to the object in the database D. The information about the object

Table 3 Mapping from terms to GER-trees

Term	$tf(t, Coll)$	$\|Coll\|$	$\|D\|$	Pointer
bar	452	721	421	GER_{bar}
hotel	295	642	354	GER_{hotel}
pizza	124	524	245	GER_{pizza}
shoes	117	236	189	GER_{shoes}

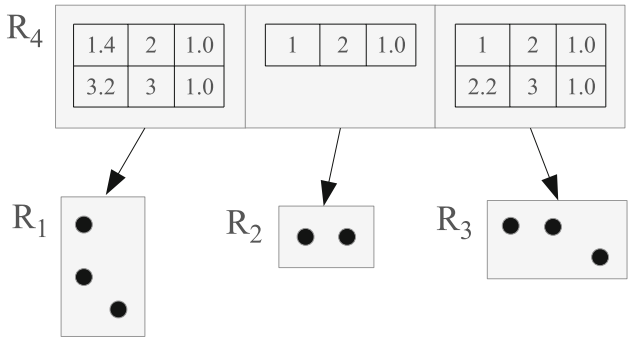


Fig. 4 Example of a GER-tree

needed to calculate the textual relevance is also maintained in the leaf node entry.

Non-leaf node entries are of the form $(histogram, mbr, cp)$, where $histogram$ is a compressed histogram that represents the objects contained in the subtree. On the one hand, the histogram must "describe" all objects in the subtree to enable pruning. On the other hand, it must be compact to enable a large node fanout. The histograms are covered in detail shortly.

The mbr encloses all objects in the subtree, as in a standard R-tree. The child pointer, cp, points to the root of the subtree. Figure 4 shows the GER-tree created from the objects in Fig. 2. For simplicity, each object contains only one single term.

A compressed histogram has entries of the form $(minDist, n_d, avg_{tf})$, where distance $minDist$ and count n_d state a density property of the objects in the subtree that the entry refers to. Specifically, they state that all set of objects with more than n_d objects have a diameter that exceeds $minDist$. Put differently, for the set of objects S contained in a given subtree, the following holds for $minDist$:

$$\forall O \subseteq S\ (|O| > n_d \Rightarrow diameter(O) > minDist)$$

Consider Fig. 4 and recall that the diameter of a set of objects is the largest distance between two objects in the set. The first entry in the histogram that describes R_1 states that any set with more than two objects has a diameter that exceeds 1.4. Likewise, the second entry states that any set with more than three objects has a diameter that exceeds 3.2. This entry then describes all objects in the subtree.

This information is used to prune the search space during query processing. According to the definition of the meaning of an entry, exaggerations of the compactness are allowed. For example, an entry (3.2,4,1.0) for R_1 that states that any set with more than four objects has a diameter that exceeds 3.2 is implied by the second entry and is correct. Similarly, an entry (2.0,3,1.0) that states that any set with more than three objects has a diameter that exceeds 2.0 is also an exaggeration of the second entry and is correct. However, such entries provide lower pruning power. In contrast, an entry (3.2,2,1.0) saying that any set more than two objects has a diameter that exceeds 3.2 can lead to incorrect pruning and is wrong.

The textual descriptions of the n_d objects that an entry states exist are not available in non-leaf nodes. However, Eq. (1) relies on information about the term occurrences in both the objects and the complete collection. Thus, to be able to calculate textual relevance when processing a query, the average value of the first term in Eq. (1), $tf(t, o.\phi)/|o.\phi|$, is stored in avg_{tf}. In Fig. 4, all objects have average textual relevance 1.0. This simplifies the example.

The average textual relevance of the objects referred to in a compressed histogram entry is calculated by the following function:

$$TR^a_{entry}(t, e) = (1 - \gamma)e.avg_{tf} + \gamma \frac{tf(t, Coll)}{|Coll|}, \tag{4}$$

where t is a term and e is an entry in a compressed histogram. The collection information is available in the vocabulary as shown in Table 3. Having the average textual relevance, the group proximity for an entry for a specific term can be inferred from Eq. (2) as shown next. We use the average textual relevance instead of the objects' actual textual relevance. This allows us to propagate information up through the non-leaf nodes in the tree.

$$GP_{entry}(t, e) = \left(\left(\left(TR^a_{entry}(t, e) \cdot e.n_d\right) + 1\right) \cdot e.n_d\right)^{-1}, \tag{5}$$

where t is a term and e is a compressed histogram entry.

Thus, we can calculate the group proximity of the objects described by a compressed histogram entry.

Theorem 1 *Given a single term t and a compressed histogram entry e that describes a set of objects G, the following holds:*

$$GP(t, G) = GP_{entry}(t, e)$$

Proof Since $e.avg_{tf}$ is the average value of the first term in Eq. (1), we can compute the per-object average textual relevance using Eq. (4). Thus, we can calculate the total textual

relevance of the objects in G:

$$\frac{\sum_{o \in G} TR(t, o.\phi)}{|G|} = TR^a_{entry}(t, e) \iff$$
$$\sum_{o \in G} TR(t, o.\phi) = TR^a_{entry}(t, e) \cdot e.n_d$$

Substituting this into Eq. (2), we get:

$$\left(\left(\left(\sum_{o \in G^t} TR\left(t, o.\phi\right)\right) + 1\right) \cdot |G^t|\right)^{-1} =$$
$$\left(\left(\left(TR^a_{entry}\left(t, e\right) \cdot e.n_d\right) + 1\right) \cdot e.n_d\right)^{-1}$$

This completes the proof. □

3.3 Building compressed histograms

So far, we have shown how compressed histograms can be used in query processing. It is intuitively clear that the specific choices of $minDist$ values to be used in the entries in a histogram affect the pruning power of the histogram. If the values are not chosen with care, several entries may cover almost the same objects, which wastes pruning power. Thus, we proceed to detail means of carefully selecting $minDist$ values. Specifically, we present a heuristic approach to building compressed histograms. A more theoretical study that targets optimality is left for future work.

First of all, to guarantee correct query results, a histogram must describe all objects in its subtree. This is so because if objects exist that are not covered in the histogram, these are not taken into account during search, and the subtree can be pruned incorrectly.

We next consider how to select suitable $minDist$ values that describe all objects in a leaf node. Compressed histograms with a good distribution of $minDist$ values are important for query processing because different queries may form different candidate groups with different diameters and group proximity scores. The quality of a group is measured by the group proximity as defined in Eq. (2). Therefore, a good distribution of $minDist$ values should be according to the group proximity.

Adding an object to a small group has a bigger impact on the group's proximity score than when adding an object to a large group. If we were to select $minDist$ values for entries according to the numbers of objects covered, the resulting histogram may have low pruning power since the group proximity may not be very different across different $minDist$ values.

Instead, we select $minDist$ values according to the group proximity. By choosing $minDist$ values for entries based on group proximity, we get resulting compressed histograms that are effective.

Compressed histogram construction is described in Algorithm 1.

Algorithm 1: BuildCompressedHistogram($term$, O)

```
1  h_tmp ← new empty temporary histogram;
2  Calculate all possible groups and group proximity values and add
   them to h_tmp;
3  Order h_tmp by minDist ascendingly and group proximity value
   descendingly;
4  Remove entries from h_tmp that do not provide lower group
   proximity values when comparing with the previous entry;
5  h_r ← new empty compressed histogram;
6  h_r.Add(h_tmp.peekFirst());
7  while h_r.Size() < n do
8      if h_tmp.Size() ≤ h_r.Size() then
9          h_r ← h_tmp;
10         break;
11     mostDiscGP_entry ← −∞;
12     nextCandidate ← empty;
13     foreach Entry e_tmp in h_tmp do
14         if e_tmp ∈ h_r then
15             continue;
16         diff ← e_tmp.GP_entry − h_tmp.peekNext().GP_entry;
17         if diff > mostDiscGP_entry then
18             mostDiscGP_entry ← diff;
19             nextCandidate ← e_tmp;
20     h_r.Add(nextCandidate);
21 foreach Entry e in h_r do
22     foreach Entry e_tmp in h_tmp do
23         if e_tmp.minDist < h_r.peekNext().minDist ∧
           e_tmp.GP_entry < GP_entry(term, e) then
24             e.n_d ← e_tmp.n_d;
25             e.avg_tf ← e_tmp.avg_tf;
26 h_r.last.n_d ← h_tmp.last.n_d;
27 h_r.last.avg_tf ← h_tmp.last.avg_tf;
28 return h_r;
```

First, all possible groups in the leaf node are calculated and inserted into a temporary histogram h_{tmp} with entries of the form $(minDist, n_d, GP_{entry}, avg_{tf})$, where n_d is the cardinality of a set O' of objects and GP_{entry} is the group proximity of the objects in O' to the query term, $term$, of the tree. The entries in this histogram are a result of either (a) the grouping of leaf objects, (b) several compressed histogram entries being combined, or (c) the duplication of existing compressed histogram entries.

An example of a temporary histogram for the objects in Fig. 2 is shown in Table 4. For ease of understanding, sets are indicated that correspond to the n_d values. Irrelevant entries have been removed according to Line 4 in Algorithm 1.

The entry with lowest group proximity value is chosen if more entries exist with the same $minDist$ value because of this filtering step. This is important since we always need to be conservative. For example, if we have two entries with the same $minDist$ value, we must keep the one that provides

Table 4 Temporary histogram of leaf objects

$minDist$	$n_d = \|O'\|$	GP_{entry}	avg_{tf}
1.0	$\|\{o_7, o_8\}\|$	0.17	1.0
2.24	$\|\{o_7, o_6, o_8\}\|$	0.08	1.0
5.83	$\|\{o_2, o_4, o_3, o_1\}\|$	0.05	1.0
6.32	$\|\{o_7, o_6, o_4, o_8, o_1\}\|$	0.03	1.0
6.71	$\|\{o_2, o_4, o_3, o_8, o_1, o_5\}\|$	0.02	1.0
7.21	$\|\{o_7, o_2, o_6, o_4, o_3, o_8, o_1, o_5\}\|$	0.01	1.0

the weakest pruning, i.e., the one that has the smallest group proximity value. If we do not discard the entries with higher group proximity, we do not describe the best case for the subtree, which may in turn yield incorrect pruning.

The next step is to select some of the entries from the temporary histogram for use in the size-constrained, compressed histogram h_r. In doing so, we wish to select a set of entries that provides high pruning power.

Initially, the entry from h_{tmp} with the smallest $minDist$ value is moved to the result histogram. When using $peek$, we do not advance the position in the histogram to the next entry. An example is given in Table 5, where the first entry is equal to the first one in Table 4. This entry is important because if it is excluded, it is necessary to create an entry with $minDist = 0$, which adversely affects the pruning capability.

Then, starting in Line 13, we find the entry with the group proximity value, GP_{entry}, that differs the most from that of the next entry in the table. For example, the second entry in Table 4 is chosen because $0.08 - 0.05 = 0.03$ is the largest difference in group proximity. Note that the first entry has already been selected. This entry is then added to the resulting histogram in Table 5.

The only information available when later making a pruning decision is the subset of entries in the compressed histogram. Therefore, it is important that the entries describe the objects in the subtree correctly. In particular, when omitting an entry, a higher number of objects must be assumed to exist in the entry located before it in the temporary histogram.

For example, if we were to omit the third entry in the resulting compressed histogram in Table 5, we are missing information about a candidate group. In response to this, we adjust the number of objects, n_d, and average textual relevance, avg_{tf}, described by the second entry. The search algorithm could assume n_d for this entry to be 8, since this is the n_d value for the fourth entry, which is the next entry assuming that the third entry is omitted.

While this assumption is safe, the pruning power is unnecessarily low. Instead, we replace the n_d and avg_{tf} of the second entry with the values from the omitted entry. By using these values of n_d and avg_{tf}, we describe the best case for the subtree. This occurs starting in Line 21. In the example, we set the n_d value of the second entry to 4 if we remove the third entry.

Table 5 Compressed histogram

$minDist$	n_d	avg_{tf}
1.0	2	1.0
2.24	3	1.0
5.83	4	1.0
6.32	8	1.0

Notice that adding a relevant object to a group will always result in a lower (i.e., better) group proximity score according to Eq. (2), independently of the textual relevance score of the object. Thus, we keep the best case information in order to prune correctly when searching.

The entries in the resulting compressed histogram in Table 5 are selected from the temporary histogram in Table 4 assuming that the compressed histogram can hold four entries. The first entry was added in Line 6. Next, the temporary histogram is iterated, and the second temporary histogram entry is added to the compressed histogram because it has the largest difference in group proximity to its successor entry. This is done in Lines 7–20. This loop runs until the compressed histogram is full. The next two entries are added in the same manner. Finally, in Lines 26–27, the last entry in the compressed histogram is assigned all objects in the subset in order to provide the best case information.

3.4 Building the index

Insertion of objects occurs generally as in the R-tree. The new aspect is that of updating the compressed histograms in the entries during insertions. The algorithm that accomplishes this takes two arguments, namely the *MBR* and $text$ of the object to be inserted, and is shown in Algorithm 2. The operations $ChooseLeaf$ and $Split$ are those of the R-tree [12].

When adding a new object to a leaf, the histograms in the parent non-leaf node entries are updated using the strategy described above. To have correct information in the complete tree, the parent nodes up to the root also have to be updated. This occurs in Lines 16 and 20 in Algorithm 2. The compression approach is similar to the one described in Algorithm 1 starting from Line 3.

3.4.1 Combining entries

When propagating information up through non-leaf nodes, not only the information propagated from the leaf level is considered. In addition, the compressed histograms in the non-leaf node entries are combined into new temporary histogram entries while taking into account the Euclidean distances between the *MBR*s of the subtrees. This ensures

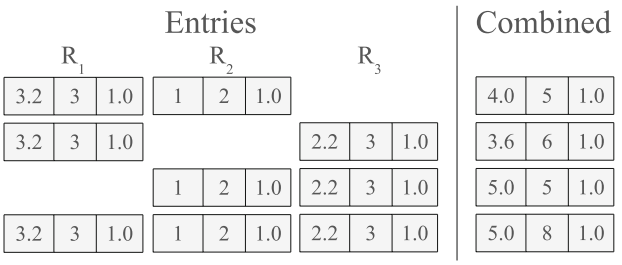


Fig. 5 Combining compressed histogram entries

that the $minDist$ values of the combined entries are consistent with the actual distances between the underlying objects.

Thus, the information about the combined non-leaf nodes is included in a temporary histogram so that accurate information can be used when creating the compressed histogram entries. Throughout, it must be ensured that the pruning is conservative. This is achieved by assuming that the objects that correspond to the object count in a compressed histogram entry are located on the *MBR*'s boundary nearest to the entry.

Example 3 If we consider Figs. 2 and 4 and focus on the first histogram entry for R_1 and the single histogram entry for R_2, we can see the shortest distance between the two *MBR*s of the two subtrees is 4.0 (this follows from Fig. 2). The entry covering R_1 states that all groups with more than two objects have a diameter higher than 1.4. Likewise, the entry covering R_2 states that all groups with more than two objects have diameter higher than 1.

From this information, it is safe to compose a temporary combined entry with $n_d = 4$ and $minDist = 4.0$, which states that all groups with more than four objects have a diameter that exceeds 4.0. The group proximity value of the combined entry can be calculated using Eq. (5). However, this entry is temporary and cannot be applied directly to the compressed histogram.

The second histogram entry for R_1 can be combined with the single entry describing R_2 to obtain a temporary entry with $n_d = 5$ and $minDist = 4.0$. This entry has the same $minDist$ value as the previous temporary entry. Thus, when building the compressed histogram, in a similar manner as in Algorithm 1 starting from Line 3, the previous entry is discarded since it does not describe the best case for the subtree.

Figure 5 provides an overview of the best combinations of compressed histogram entries for the three subtrees. The first row describes the best case combination of the entries describing R_1 and R_2. The second row describes the best case for R_1 and R_3, the third row provides the best case for R_2 and R_3, and the fourth row provides the best case when considering all three subtrees. □

If two compressed histogram entries have larger $minDist$ values than the distance between the *MBR*s, the $minDist$ value of the combined histogram entry can be set to the smallest of the two $minDist$s. This is possible because the covered objects cannot be within a distance smaller than this distance. The objects with a distance smaller than this are described by a previous histogram entry and will therefore still be considered when creating the resulting compressed histogram. Since we do not describe single objects in the histograms, but only more than one objects, we will always assume that a single object lies on the border *MBR*s to ensure correctness. This is important in order to prune correctly.

Algorithm 2: Insert(*MBR*, *text*)

```
1  N ← ChooseLeaf(MBR);
2  Add MBR to node N;
3  Update the histogram of N;
4  if N requires a split then
5      (O, P) ← N.Split();
6      Update the histograms of O and P;
7      if N is the root then
8          Create a new node M;
9          Add O and P to M;
10         Use the histograms and MBRs in O and P to update the
           histograms in the entries in M;
11         Set M to be the new root node;
12     else
13         foreach Node T seen when ascending from N do
14             Adjust the covering rectangles;
15             Do splits when needed;
16             Use the histograms and MBRs in T to update the
               histograms' parent entry;
17 else if N is not the root then
18     foreach Node T seen when ascending from N do
19         Adjust the covering rectangles;
20         Use the histograms and MBRs in T to update the parent
           entry's histogram;
```

The approach described previously for controlling the size of a compressed histogram is used. Equation (5) is used to calculate the group proximity value having only the compressed histogram entries. Small histograms are faster to maintain and yield tree nodes with larger fanout. However, pruning benefits from large histograms. The experimental study covers the impact of different histogram sizes.

3.5 Updates

The objects we index are stationary and change infrequently; that is why we focus on query processing performance. However, updates can be handled by conservative deletions followed by insertions. When an object is deleted, the information in compressed histograms remains correct since it is conservative. However, some pruning power is generally lost. When inserting an updated object, the relevant histograms are updated to also describe this new object.

3.6 Query processing

In query processing, we need to be able to calculate a lower bound of the cost by using the compressed histograms. We thus present a lower bound, $MinCost(q, e_E)$, that is the minimum spatio-textual group distance between the query q and a single entry e_E of an extended compressed histogram. This bound is used to prune the search space when querying.

3.6.1 Extended compressed histograms

Because the main purpose of the compressed histograms is to enable pruning of sub-trees in which no candidate results can exist, compressed histograms maintain only a subset of the inter-object distances. When we process a query, we need to maintain more detailed information about the best entries in the compressed histograms. Specifically, we need the exact inter-object distance for candidate groups. Therefore, we extend the compressed histogram entries with more information for use when processing queries.

We thus introduce extended compressed histograms that provide more information about a histogram entry. For each query term, the corresponding number of objects, n_d, and value of avg_{tf} is stored. Several node entries may combine to form an intermediate result since a candidate group's objects may be located in different *MBR*s. Therefore, the extended histogram includes a set M that describes the *MBR*s that contain the objects.

Thus, entries in an extended compressed histogram are of the form $(minDist, M, \{(t, n_d, avg_{tf})_1, \ldots, (t, n_d, avg_{tf})_{|q.\phi|}\})$, where t is a term, n_d is a number of objects, and avg_{tf} is an average textual relevance.

The lower bound cost-function that uses extended compressed histograms is defined as follows:

$$\begin{aligned} &MinCost(q, e_E) \\ &= \alpha \left(\frac{\beta D_{qM}(q.\lambda, e_E.M) + (1-\beta) e_E.minDist}{max D} \right) \\ &\quad + (1-\alpha) \prod_{t \in q.\phi} GP(t, e_E^t), \end{aligned} \tag{6}$$

where q is the query, and e_E is an extended compressed histogram entry. Next, e_E^t contains the same information as the normal histogram entry, but only for the query term t, and $GP(t, e)$ is as defined in Eq. (5). The smallest distance from the query location $q.\lambda$ to any *MBR*, m, in the set M is defined by $D_{qM}(q.\lambda, M) = min_{mbr \in M} mindist(q.\lambda, mbr)$, where $mindist$ returns the minimal distance between a point and a rectangle.

Theorem 2 *Given a query q and an extended compressed histogram entry e_E that covers a set of objects G, the following holds:*

$$MinCost(q, e_E) \leq Cost(q, G)$$

Proof Since the set of *MBR*'s, $e_E.M$, encloses all objects in G, the minimum distance between $q.\lambda$ and the nearest *MBR* is no larger than the distance from $q.\lambda$ to any object in G:

$$D_{qM}(q.\lambda, M) \leq ||q.\lambda, o||, \quad o \in G$$

According to the definition of $minDist$, the diameter of the objects in G is at least $e_E.minDist$. Thus, we have:

$$e_E.minDist \leq diameter(G)$$

By Theorem 1, we have $GP(t, G) = GP(t, e)$ for single terms. Thus, with the extended compressed histogram entry, we have the following for multiple terms:

$$\prod_{t \in q.\phi} GP(t, e_E^t) = GP(q.\phi, G)$$

This completes the proof. □

3.6.2 Single-keyword query processing

Single-keyword queries are processed by means of best-first tree traversals. Since the objects in each result group can be distributed across several leaf nodes, the paths followed to obtain the result may include several non-leaf nodes. Therefore, when calculating the lower bound cost, all combinations of non-leaf node entries should be considered. A priority queue U that maintains the order of the sets of nodes is used to keep track of candidate groups. The candidate groups can contain both leaf and non-leaf nodes. Initially, the priority queue holds only the root node as shown in Line 2 in Algorithm 3.

The nodes in the first set are dequeued in Line 3 in Algorithm 4. In Line 6, all possible combinations of entries in the set of nodes are calculated. Recall that leaf node entries maintain information that enables calculation of the textual relevance so that a histogram entry can easily be created. Also, we always assume a single object is located on the *MBR* border. The possible outcomes are limited because of the $minDist$ value since all objects within the same diameter per definition can be grouped together. This is due to Eq. (2) that gives lower scores to groups with more relevant objects. This greatly reduces the set of combinations. An example of the reduction is provided in Example 4.

The approach described in Sect. 3.4.1 is used to create the combined entries. The possible combinations of entries are stored in an extended histogram entry.

Finally, the lower bound costs of the extended histograms are calculated in Lines 7–8, and the sets of nodes are added to the priority queue. From Line 10, when all nodes in N

are leaf nodes, we calculate the actual costs, $Cost(q, G)$, for all object combinations. The possible combinations can be reduced as mentioned above; again, an example is provided in Example 4.

Example 4 In Fig. 2, the combinations $\{o_1, o_4\}$, $\{o_1, o_3, o_4\}$, $\{o_1, o_2, o_4\}$, and $\{o_1, o_2, o_3, o_4\}$ all have diameter 5.83, and the distance from the query to the group of objects is three in all cases. Therefore, it is sufficient to include only $\{o_1, o_2, o_3, o_4\}$, which has the lowest group proximity value. The same applies when combining histogram entries. Note that single entries, e.g., $\{o_1\}$, $\{o_2\}$, are unique combinations and are always considered. □

As shown in Line 11, if the cost is lower than that of any other candidate set of nodes in the priority queue, the group is added to the result list. Since an object can be in at most one group, the result list has to be examined before computing the cost of the following candidates, as shown in Line 10.

Algorithm 3: SKSearch($index, q, k$)

```
1 U ← new min-priority queue;
2 U.Enqueue(index.root, 0);
3 DoSearch(index, q, k, U);
```

Example 5 Consider the objects and the query location in Fig. 2 and the resulting GER-tree in Fig. 4. We perform a top-1 query. Recall that for simplicity, all objects contain the same terms.

First, we enqueue the root node R_4 along with the best possible minimum cost according to Algorithm 3. Next, Algorithm 4 proceeds as follows. To simplify, we omit describing the extended histograms, but replace them with the identifiers of the corresponding nodes. Figure 4 describes the single entries and Fig. 5 the combined ones. For example, combining the last entries describing R_1 and R_3 yields $D_{qM}(q.\lambda, M) = 2.0$, which follows from Fig. 2. Thus, the lower bound cost can be calculated as:

$$\begin{aligned} MinCost(q, e_E^{R_1,R_3}) &= 0.4 \cdot \frac{0.4 \cdot 2.0 + 0.6 \cdot 3.6}{7} \\ &\quad + 0.6 \cdot (((1 \cdot 6) + 1) \cdot 6)^{-1} \\ &= 0.184, \end{aligned}$$

where q is as given in Example 2 and $e_E^{R_1,R_3}$ is the extended histogram created by combining the last compressed histogram entries describing R_1 and R_3.

1. Dequeue $\{R_4\}$. Combine entries and calculate $MinCost$ values.
   Queue: $\langle 0.178 = \{R_3\}, 0.184 = \{R_1, R_3\}, 0.194 = \{R_1\}, 0.203 = \{R_1, R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1, R_2, R_3\}\rangle$
   Groups are always assumed to start on the nearest MBR border from the query location when calculating $MinCost$ values.
2. Dequeue $\{R_3\}$. Best cost: 0.199 (set $\{o_6, o_7, o_8\}$).
   Queue: $\langle 0.184 = \{R_1, R_3\}, 0.194 = \{R_1\}, 0.203 = \{R_1, R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1, R_2, R_3\}\rangle$
3. Dequeue $\{R_1, R_3\}$. Best cost: 0.199 ($\{o_6, o_7, o_8\}$).
   Queue: $\langle 0.194 = \{R_1\}, 0.203 = \{R_1, R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1, R_2, R_3\}\rangle$
4. Dequeue $\{R_1\}$. Best cost: 0.227 ($\{o_1, o_2, o_3\}$).
   Queue: $\langle 0.203 = \{R_1, R_2\}, 0.217 = \{R_2\}, 0.225 = \{R_1, R_2, R_3\}\rangle$
5. Report the set $\{o_6, o_7, o_8\}$ as the best group since $0.199 \leq 0.203$. □

3.6.3 Multiple keyword query processing

We proceed to describe the algorithm for handling multiple keyword queries. The definitions of distance to a group and group size are independent of the terms occurring in a group. The group proximity defined in Eq. (2) recursively multiplies the group proximity score of each term and thus provides a lower group proximity value when multiple terms occur. This has the desired effect of favoring groups that contain multiple keywords.

Recall that if a group does not contain any objects with a term from the set of query keywords, we cannot evaluate the function defined in Eq. (2). Thus, we will not consider such groups as result candidates. This ensures that the resulting group will contain at least one object with the term.

As described above, Algorithm 4 finds groups of objects that can be located in several nodes. We can utilize this property when evaluating queries with multiple keywords.

Algorithm 4: DoSearch($index, q, k, U$)

```
1  R ← new result list;
2  while U ≠ ∅ do
3      N ← U.Dequeue();
4      if N contains a non-leaf node then
5          candidates ← empty set;
6          Calculate all possible combinations of entries, including
           single entries in N, and add them to candidates;
7          foreach Entry e ∈ candidates do
8              U.Enqueue(e.M, MinCost(q, e)) ;
9      else if N contain only leaf nodes then
10         Find a set of objects, O from N, where O ⊆ N and
           ∀o ∈ O (o ∉ R) such that Cost(q, O) is smallest;
11         if Cost(q, O) ≤ U.First().Value() then
12             Output O as the next relevant group;
13             R.Add(O);
14             if k = |R| then
15                 return;
```

However, the algorithms presented so far only work on single GER-trees. Since we have a GER-tree per term, it is necessary to search several trees for candidate groups. Thus, we present a multiple keyword search algorithm, $MKSearch$, in Algorithm 5.

The *MKSearch* algorithm works with several GER-trees. The relevant trees are retrieved in Line 1. Next, all root nodes from the sources are combined into a set and enqueued into a priority queue with the minimum cost 0. This queue is then passed to the $DoSearch$ Algorithm 4. The lower bound cost-function defined in Eq. (6) supports multiple keywords by using the extended histograms. Likewise, the cost-function from Eq. (3) supports multiple keywords. Thus, the search algorithm presented in the previous section can then be applied directly.

Algorithm 5: MKSearch($index, q, k$)

```
1 T_i ← GER-Tree of the term t_i ∈ q.φ;
2 U ← new min-priority queue;
3 Root ← empty set;
4 foreach GER-Tree t in T do
5 |  Root ← Root ∪ t.root;
6 U.Enqueue(Root, 0);
7 DoSearch(index, q, k, U);
```

3.6.4 Complexity

The index structure is based on the R-tree and has the same structure and minimum bounding rectangles as the R-tree. Since there is no distance search range, the most relevant groups may in theory exist anywhere. The result is that we have the same worst-case performance as does the R-Tree. In the worst case, the compressed histograms may provide unusable heuristics for the majority of the dataset, thus expanding the search range to the full dataset since nothing can be pruned. However, as shown in the experimental studies, the proposed solution performs well in practice, with realistic data and queries.

3.7 Enhancing the GER-tree

The index histograms are built at insertion time, and when performing queries, leaf node entries are combined. Although the $minDist$ bound limits the number of possible combinations, the inter-object distance calculations are still expensive. In the following, we propose two techniques that reduce the time spent on evaluating the combinations and present also a bulk insertion approach.

3.7.1 Storing calculations

At insertion time, we have to perform inter-object calculations at every insertion. And during query processing, the possible combinations of leaf objects have to be evaluated. To limit the number of calculations needed at insertion and query time, we propose a variant of the GER-tree, called the Precomputed GER-tree (PGER-tree), where leaf nodes store additional information.

In particular, we store a compressed representation of distances in leaf nodes such that a leaf node contains enough information to enable recreation of all possible combinations of leaf objects without computing any inter-object distances. At insertion time, this reduces the time needed to create the compressed histograms in parent non-leaf nodes. At query time, when the leaf level is reached, the only distances that need to be calculated are those between objects located in different nodes.

We propose a simple representation from which all distances can be inferred. For each object, a list of distances to the other objects not already described is stored. The object identifiers are implied by the positions in the lists. For example, the representation of the objects in Fig. 2 results in this set of lists: $\langle 2, 3.16, 5.83, 6.71, \ 6.32, 5.83, 5.00\rangle$, $\langle 1.41, 5.10, 6.08, \ 7.21, 7.07, 6.40\rangle$, etc. The first list contains distances from object o_1 to the other seven objects. The second list contains distances from o_2 to the others except o_1 that is already described in the previous list.

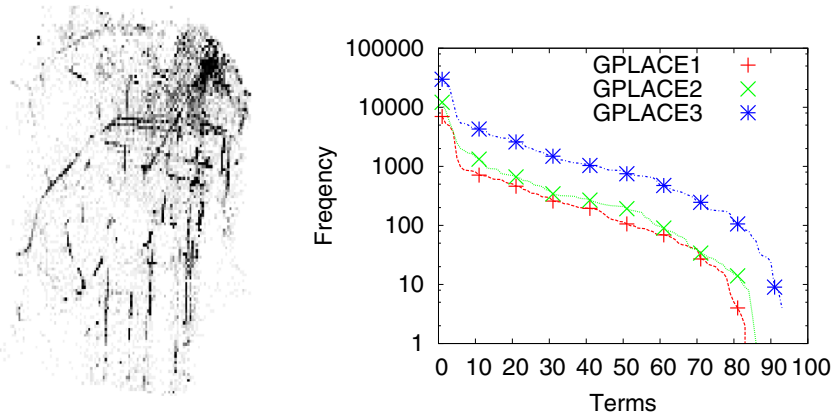
These compact representations are used in Line 2 and Line 10 in Algorithm 1 and Algorithm 4, respectively.

3.7.2 Block sharing among leaf nodes

The compressed histograms are built from the objects stored in the leaf nodes. Since only little space is available for histograms, these can only store limited information. Increasing the histogram size decreases node fanout and results in fewer entries in the leaf nodes. Therefore, the leaf nodes will have fewer entries than guaranteed by the standard R-tree that does not employ histograms in the non-leaf nodes. The histograms will be less effective in pruning when only a small amount of space is allocated to them. This will increase the number of objects in the leaf nodes, since there will be fewer node splits. On the other hand, when the histograms increase in size, the leaf nodes will contain fewer entries and not occupy a full disk block.

To prevent almost empty disk blocks, we propose a variant of the GER-tree, called the Combined Leaf GER-tree (CLGER-tree), where pairs of leaf nodes share a disk block. Since the result of a query may be located in several leaf nodes, two or more leaf nodes may have to be read to answer the query. When leaf nodes share disk blocks, a leaf node needed by a query may already be present in the buffer since the full disk block that contains the node may have been read from disk previously.

Fig. 6 Dataset properties



(a) Spatial Distribution, GPLACE1 **(b)** Term Distribution

3.7.3 Delayed compressed histogram computation

When inserting objects into the GER-tree, it is necessary to update the compressed histograms in non-leaf nodes. This has a substantial, adverse effect on the insertion time. The insertion algorithm described previously uses the standard R-tree algorithms for splits and *MBR* creation. Thus, we may simply create the R-tree, but only reserve space for the compressed histograms without computing them.

While the compressed histograms are initially empty, the spatial structure of the tree is identical to the spatial structure of the tree obtained when the compressed histograms are maintained at each insertion. When all objects have been inserted, the compressed histograms are created in bottom-up fashion, thus yielding a reduction in the insertion time.

The resulting GER-trees are identical to the ones obtained with eager histogram computation. Note also that eager histogram computation can be applied after the initial bulk insertion in order to enable ad hoc insertions.

4 Experimental evaluation

We proceed to report on an empirical study of the insertion and query performance of the proposed techniques. We first cover the experimental setup and then the findings from a range of specific experiments.

4.1 Experimental setup

Algorithms considered We implemented the techniques described in the paper, along with the different described enhancements. We use the name PCLGER-tree for tree that combines the PGER-tree and CLGER-tree, and we report on experiments with these three trees.

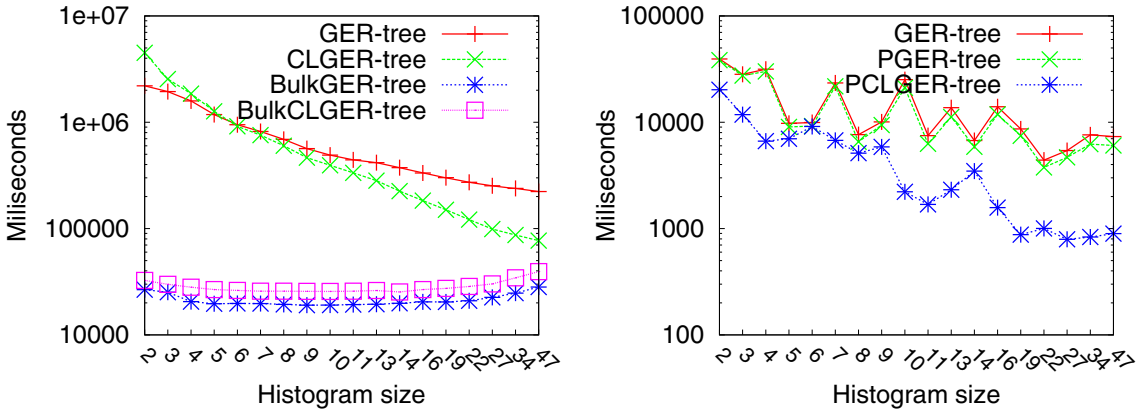
Datasets The experiments are performed on three datasets obtained from Google Places [11]. Objects from Google Places are each tagged with a set of terms from a list of 93 categories.

The first dataset, GPLACE1, consists of 27,171 real objects located in central San Francisco, USA. The maximum distance in the Euclidean space of this dataset is 14 km. The second dataset, GPLACE2, contains 43,062 real objects from Las Vegas that are within a diameter of 36 km. The third dataset, GPLACE3, is created by shifting the objects from four different US central city regions so they all appear in the Las Vegas region and thereby yield a dense, synthetic city with 139,246 real objects. Like GPLACE2, these objects are located within a region with maximum extent 36 km.

We use GPLACE1 in the experiments unless stated otherwise. We use the other two datasets when we vary the dataset size. The spatial distribution of the objects in GPLACE1 is depicted in Fig. 6a with the most central part of San Francisco visible in the upper right corner.

Queries To study the performance of the GER-tree and the two enhancements, we create a set of queries for each of the three datasets. Each query set contains 100 queries. The queries for a dataset are generated by first forming the bag of terms containing all occurrences of terms in all objects in the dataset. Terms that occur often in objects then occur often in the bag. The term distribution is shown in Fig. 6b. We then select 10 keywords at random from the bag.

This procedure aims to ensure that the choice of query terms mirrors the term distribution in the data, which we believe is more realistic than selecting simply from the set of terms. For example, "restaurant" occurs more frequently than "plumber" in objects. The query construction then assumes that users will also query for "restaurant" more often than for "plumber." For multi-keyword queries, we select multiple unique terms at random from the bag of terms.



(a) Insertion **(b)** Querying

Fig. 7 Varying histogram size

Finally, we generate 10 random locations within the dataset boundaries and obtain 100 queries by combining the 10 keywords with the 10 locations.

Settings The indexes are created and stored on disk using a page size of 4KB. The algorithms are implemented in Java, and the experiments are run on an Intel(R) Core(TM) i5-2520M CPU at 2.50 GHz.

The maximum allowed main memory buffer size is set to 10 % of the size of the tree used in a particular experiment. A random replacement buffer is used. If not specified otherwise, the compressed histogram size is set to 22 entries for the (P)GER-trees and to 27 entries for the PCLGER-tree since these values provide the best performance according to Fig. 7b.

If not specified otherwise, we fix the value of α at 0.9 and β at 0.2, and we retrieve the top-3 groups. Table 6 shows that these settings yield an average group size of 2.2 objects for the top-3 groups in a result. Further, Fig. 9b shows that these settings result in an average query-to-group distance of 745 m and an average group diameter of 10 m.

4.2 Experimental findings

Varying the compressed histogram size In this experiment, we vary the number of entries in the compressed histograms. This influences both insertion and query performance.

We consider only histogram sizes that fully utilize the space available in a node. For example, we do not consider histograms with 12 entries because these are dominated by histograms with 13 entries—the reduction in the histogram size from 13 to 12 entries does not enable higher node fanout.

Since the histogram size affects the fanout, it also affects the number of splits that occur during insertions. This then increase the time to build the index. The (CL)GER-trees update the histograms at every object insertion, whereas the Bulk(CL)GER-trees postpone the histogram updates.

Table 6 Average group size when varying α

α:	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.90	0.95
$\lvert G\rvert$:	9.2	8.7	7.7	7.0	6.3	5.5	4.4	3.5	2.2	1.2

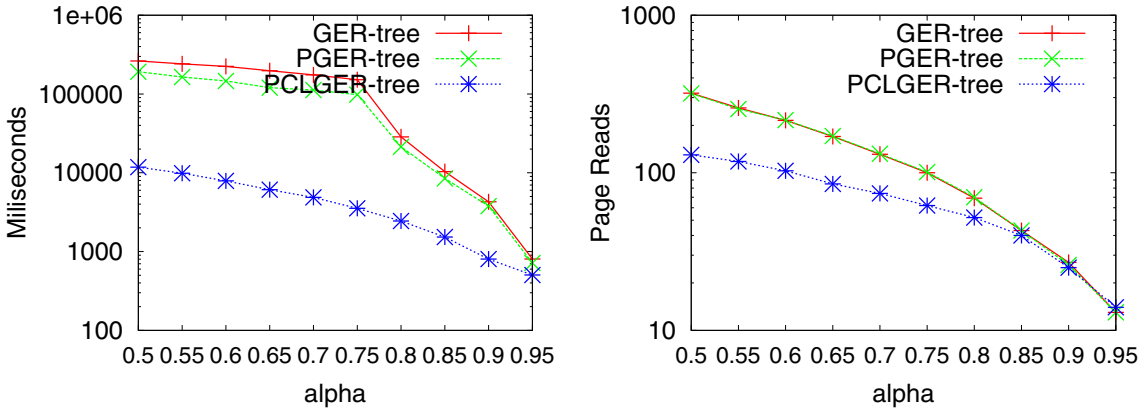
Figure 7a shows that the BulkGER-tree outperforms the other trees. This tree is lower than the CLGER-tree, and histogram updates are postponed.

The time needed to maintain the histograms in non-leaf nodes depends on the number of index entries (or subtrees) in the nodes. When propagating the histograms up the tree, all possible combinations of histogram entries from the histograms in the index entries must be examined. Therefore, more computations must be performed as the histogram size decreases. For very small histogram sizes, Fig. 7a shows that all four trees become slower as the histogram size decreases. In this range, the GER-tree is faster than the CLGER-tree since it has smaller height.

Larger histograms yield smaller fanout and thus more splits and nodes to maintain. Thus, we see increasing insertion times for the two bulk indexes since the time to build the R-tree becomes dominant. In contrast, the two non-bulk indexes become faster with smaller fanout since fewer of the dominating histogram entry combinations must be examined for every insert. The CLGER-tree becomes faster than the GER-tree since the small leaf nodes also reduce the number of possible object combinations.

Histogram size affects query performance greatly. With small histograms, the fanout is high, but little information is also available about subtrees. This adversely affects the ability to prune, which means that opportunities for improving the query performance are lost; see Fig. 7b.

As the histogram size increases, the query performance exhibits an improving trend. Only as the histograms get to be very large does the query time deteriorate slightly, which is due to the low fanout.



(a) Runtime **(b)** I/O

Fig. 8 Query performance when varying α

We attribute the "jumping" performance behavior shown in Fig. 7b to the R-tree node splitting algorithms. The locations of objects in leaf nodes greatly impact the histograms, so the particular placements of objects into leaf nodes and the splits performed during insertion also affect query performance.

All of the following experiments show a small performance difference between the PGER-tree and the GER-tree. Therefore, we omitted the PGER-tree.

Varying α We proceed to consider an experiment designed to observe the effect of varying parameter α.

Recall from Eq. (3) that parameter α allows us to prioritize between the spatial properties of a group and the group's textual relevance to the query, which takes into account both the number of objects and the textual relevance of the objects. As the value of α increases, the most relevant groups tend to be located closer to the query. This also means that the number of objects in the groups decreases with increasing α. The average number of objects per group for different values of α is shown in Table 6.

If parameter α is set to 0, the most relevant group is the set of all objects in the dataset that have a query keyword in their text description. When set to 1, the number and text relevance of objects in a group are not considered, and the nearest object is returned.

Figure 8 reports on the query performance when varying α. Figure 8a shows, as expected, an improvement in the performance for larger values of α. Figure 8b shows that the average number of disk page accesses for a query decreases. The larger leaf nodes in the (P)GER-trees results in more possible group combinations and less descriptive compressed histograms.

Varying β The next experiment is designed to observe the effect of varying parameter β, which allows us to balance between the spatial distance from the query location to the group and the group diameter.

With β close to 0, the diameter of a group becomes more important than the distance from the query to the group (cf. Eq. 3). This has the effect of improving the query time because the objects in the group are more likely to be located close together in the index, which enables more effective pruning. This is shown in Fig. 9a.

The performance of the PCLGER-tree decreases less markedly than that for the two other trees because it only holds half of the leaf node objects as do the (P)GER-trees. Therefore, more candidate groups are distributed across several leaf nodes, also for small values of β. As β increases, the diameter of the groups tends to become larger, meaning that groups tend to be distributed across more leaf nodes.

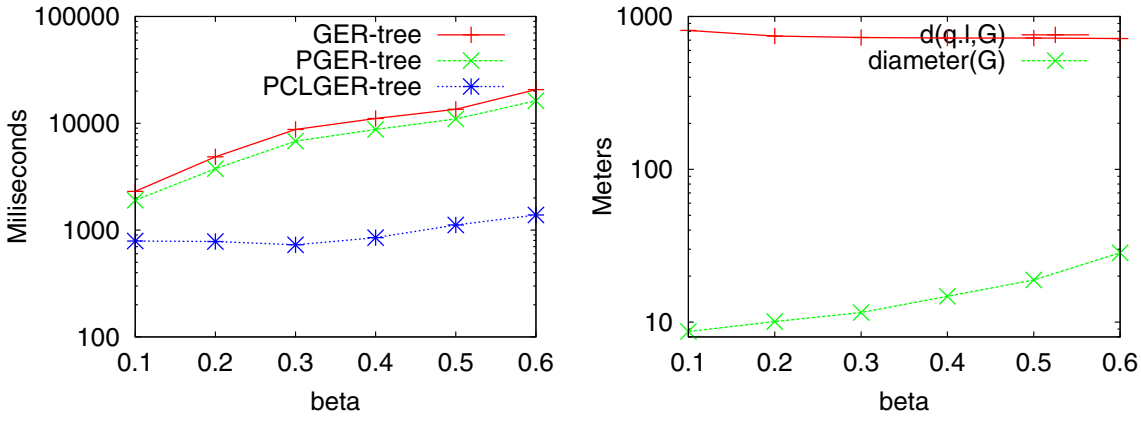
Figure 9b shows that, as expected, the group size increases while the distance to the group decreases.

Varying k Fig. 10 shows the results when varying the number of result groups. As expected, query performance decreases with increasing k, which can be seen in Fig. 10a.

Also, the PCLGER-tree outperforms the (P)GER-trees as expected and quite substantially so. This holds for all k.

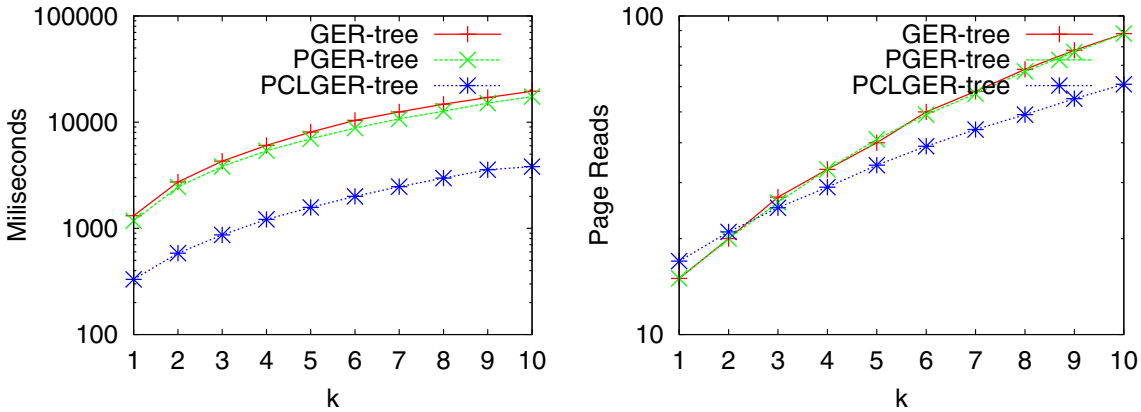
For values of $k < 3$, the PCLGER-trees access slightly more pages than the (P)GER-trees, as shown in Fig. 10b. The leaf nodes in the (P)GER-trees may describe twice as many leaf objects as does the PCLGER-tree, which may then be of lower height, which in turn has the effect of generally incurring fewer page accesses. However, more possible object combinations exist for every leaf node in the (P)GER-tree, resulting in more calculations (CPU) and thus slower performance than that of the PCLGER-tree.

Since PCLGER-tree leaf nodes may contain only half as many objects as do leaf nodes in the two other trees, the compressed histograms in the parent nodes of the PCLGER-tree are more descriptive than are those in the two other trees. Thus, fewer nodes are fetched compared to the (P)GER-trees. The difference in page accesses increases as expected



(a) Runtime **(b)** Spatial Properties

Fig. 9 Query time when varying β



(a) Runtime **(b)** I/O

Fig. 10 Query performance when varying k

with k since more leaf nodes are examined. The (P)GER-trees become more I/O bound than the PCLGER-tree as k increases.

Varying the number of keywords Figure 11a shows the query time for different numbers of keywords for $k = 1$. As the number of keywords increases, more trees are fetched since each term has its own GER-tree. Thus, the number of possible combinations of node entries increases, and the size of the candidate set calculated in Line 6 in Algorithm 4 increases with the number of keywords. This results in more calculations, and since the number of combinations of histogram entries increases, the performance drops for all three trees when increasing the number of keywords.

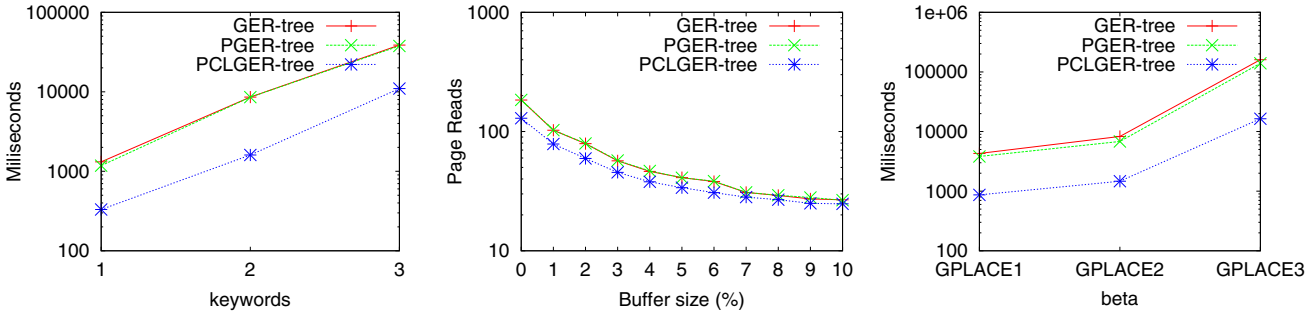
Varying the buffer size We allow a random replacement buffer to store up to 10 % of the nodes in the index. When decreasing the buffer size from the maximum 10 % of the index size, the number of page accesses increases for all three indexes—see Fig. 11b. This is as expected.

Varying the dataset In this experiment, we employ all three datasets. Dataset GPLACE2 contains 58 % more objects than GPLACE1. However, Fig. 11c shows that the query time increases only 12 %. The objects in GPLACE2 are distributed across a larger region than those in GPLACE1, making it easier to prune subtrees. Therefore, the query time does not increase significantly for any of the three algorithms. However, the GPLACE3 dataset contains a higher concentration of objects, which reduces the pruning power and causes a significant increase in the query time.

Quality As shown in Table 6, the groups contain reasonable numbers of objects and thus each provides a selection for the user. Also, Fig. 9b (FIX) shows that the distances from the query location to the groups and the diameters of the groups have reasonable values for all parameter settings.

5 Baselines and related work

We proceed to discuss baseline approaches and cover related work.



(a) Varying Number of Keywords **(b)** Varying Buffer Size **(c)** Varying Dataset

Fig. 11 Performance for varying number of keywords, buffer size, and dataset

5.1 Baselines

While no algorithm exists for computing the top-k relevant groups of spatial web objects, we proceed to outline two baseline approaches based on different ideas. One performs a full enumeration, and the other adapts existing techniques for finding top-k single spatial web objects. We first discuss the possible use of standard clustering techniques for the paper's problem.

Existing object clustering techniques such as DBSCAN and k-means [10,14] find a set of clusters given settings of parameters specific to each technique. It is not clear how to appropriately apply such techniques to compute the query we consider. First, given a query with a specific location, keywords, and number of results k, it is not clear how to "map" these parameter settings to the parameters accepted by the clustering technique so that a correct query result is computed by the clustering technique. Second, performing clustering at query time is likely to be too time-consuming. Thus, we must in some way cluster the set of objects for each combination of each possible query location and each possible parameter setting before querying can commence. Due to these difficulties, we are unable to compare with clustering techniques and do not pursue a clustering approach in our solution.

Baseline 1 (full enumeration): We create all possible subsets, $G \subseteq D$, and calculate the cost of each. We then sort the subsets according to their scores and return the top-k most relevant, non-overlapping groups. However, the number of subsets and score computations is exponential in the dataset cardinality.

Baseline 2 (index): For $\alpha > 0$ and $\beta > 0$, the groups are ranked according to both spatial proximity and textual relevance to the query. Thus, we can employ any of the existing spatio-textual index structures, e.g., the S2I index [23], to search for nearest neighbors.

We find the spatially nearest neighbor to the query location that matches one of the query keywords. Considering the object found as a starting object, we create all possible subsets, $G \subseteq D$, that contain this starting object. For each of these subsets, we calculate the cost.

We continue to find the next nearest neighbor and calculate the costs for the corresponding subsets. We do this until the distance to the nearest-neighbor object becomes so large that no matter which objects in the dataset are added to the group, the group's cost cannot be competitive. In doing this, we must assume that all inter-object distances for the group are 0. This conservative assumption is needed to ensure that the best possible cost is assumed.

The first baseline is inefficient because it materializes the entire search space of solutions and computes the cost of every possible result group. We thus disregard this solution. The second baseline has to search most parts of the dataset and is also not be considered in the paper's empirical study. Instead, different variants of the proposed solution are used for comparison.

5.2 Related work

Spatial keyword querying has recently started to attract substantial attention. With only four exceptions, to the best of the authors' knowledge, all works in the literature compute results with a single-object granularity, and no other work addresses the problem studied in this paper. Here, we first consider indexes with aggregated information, then single-object solutions, and finally, the solutions that compute a set of objects.

Aggregate queries Our GER-tree maintains a particular kind of histogram in R-tree node entries. Previous work exists that has placed aggregate information about subtrees in node entries in R-trees and other trees in order to compute spatial aggregate queries [15–17,24,25]. However, no studies employ histograms to provide efficient pruning, but store only single aggregate values such as *count* and *sum*. Also, they do not consider objects with both spatial and textual values and thus do also not take into account the textual relevance of the objects indexed.

Top-k most relevant spatial web objects Cao et al. [3] review recently proposed spatial keyword querying techniques, and Chen et al. [6] review and compare proposals for indexing in relation to spatial keyword querying.

Zhou et al. [30] propose to use inverted files [31] for spatial keyword querying and create an R*-tree for each distinct term. This is similar to the Spatial Inverted Index (S2I) by Rocha-Junior et al. [23]. We employ the same general approach of having one index per data term. However, unlike our proposal, these previous works consider results with single-object granularity and do not employ compressed histograms to prune the search space.

Another approach, yielding the IR-tree family of indexes, has been proposed by Cong et al. [7,26] and Li et al. [19]. Here, inverted files are attached to the nodes in R-tree-type structures. Work with hybrid indexes has also been proposed by others [8,13]. Li et al. [18] investigate direction-aware spatial keyword querying, the idea being to take into account the querying user's movement direction. Wu et al. [27] study the processing of continuous spatial keyword queries for moving users.

Cao et al. [4] address the problem of retrieving the top-k prestige-based spatial web objects. They introduce PageRank-like techniques that give objects prestige and improved rankings in results if they are located close to other textually similar objects. Unlike in all other works, objects are thus not independent of each other. However, only k single objects are returned, and the IR-tree is used for query processing.

The above solutions all return single-object granularity results, while we compute groups of objects. They store at most simple information in nodes, e.g., the *count* of the terms in a subtree, while we employ compressed histograms. We know of no way to adapt previous techniques to efficiently process the top-k groups spatial keyword query.

Sets of spatial web objects Four works consider queries that return a group of objects [5,20,28,29]. However, their queries differ significantly from our proposal since they aim to find a set of objects that, when considered as one object, matches the query keywords. If a single object matches all the keywords in a query, only that object is returned. Thus, they do not aim to support browsing behavior, but instead return the minimal set of objects that together match the query keywords. Also, these queries return only a single group of objects rather than k groups. Further, when evaluating textual relevance, the objects in a group are treated as a single object, and a group of objects satisfies the textual relevance if its objects collectively contain the query keywords, meaning that the textual matching is Boolean.

In contrast, our type of query ranks objects individually (each object must be relevant to the query) and quantitatively. Also, while the previous works find a spatially nearest (in some specific sense) minimal set of objects that collectively satisfies the query keywords, our query aims to find large groups of objects. As a result of these differences, the type of query we study calls for a very different solution.

A demo of a system based on this paper's proposals was presented at VLDB and is covered in a four-page demo paper [2].

6 Conclusions and future work

We present a novel and carefully motivated and designed top-k groups spatial keyword query that, given a query location and keywords, retrieves nearby, dense, and relevant groups of spatial web objects such as geo-tagged web pages, business directory entries, or check-ins.

We also present techniques aimed at enabling the efficient processing of such queries. The techniques include a new R-tree-based indexing technique that stores compact histograms in node entries that approximate subtrees in a particular manner while preserving reasonable node fanout. The provided query processing algorithms use the index and its histograms for pruning the search space and directing the search while taking into account group diameter and distance and relevance to the query.

An empirical study with real data offers insight into the design properties of the proposed techniques and suggests that the techniques are capable of supporting query processing in realistic, real-world settings.

Future research may consider other types of indexes and new kinds of histograms, which then calls for new query processing algorithms.

Acknowledgments This research was supported in part by the European Union Seventh Framework Programme—Marie Curie Actions, Initial Training Network Geocrowd (http://www.geocrowd.eu) under Grant Agreement No. FP7-PEOPLE-2010-ITN-264994.

References

1. Amitay, E., Har'El, N., Sivan, R., Soffer, A.: Web-a-where: geotagging web content. In: *SIGIR*, 273–280 (2004)
2. Bøgh, K., Skovsgaard, A., Jensen, C.S.: Groupfinder: a new approach to top-k point-of-interest group retrieval. PVLDB **6**(12), 1226–1229 (2013)
3. Cao, X., Chen, L., Cong, G., Jensen, C.S., Qu, Q., Skovsgaard, A., Wu, D., Yiu, M. L.: Spatial keyword querying. In: Atzeni P., Cheung, D., Ram S. (eds.) Conceptual Modeling. Proceedings of the 31st International Conference ER 2012, Florence, Italy, October 15–18, 2012. Lecture Notes in Computer Science, vol. 7532, pp 16–29. Springer, Berlin, Heidelberg (2012)
4. Cao, X., Cong, G., Jensen, C.S.: Retrieving top-k prestige-based relevant spatial web objects. PVLDB **3**(1–2), 373–384 (2010)
5. Cao, X., Cong, G., Jensen, C. S., Ooi, B. C.: Collective spatial keyword querying. In: *SIGMOD*, pp. 373–384 (2011)

6. Chen, L., Cong, G., Jensen, C.S., Wu, D.: Spatial keyword query processing: an experimental evaluation. PVLDB **6**(3), 217–228 (2013)
7. Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial web objects. PVLDB **2**(1), 337–348 (2009)
8. De Felipe, I., Hristidis, V., Rishe, N.: Keyword search on spatial databases. In: *ICDE*, pp. 656–665 (2008)
9. Ding, J., Gravano, L., Shivakumar, N.: Computing geographical scopes of web resources. In: *VLDB*, pp. 545–556 (2000)
10. Ester, M., Kriegel, H.-P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. Kdd **96**, 226–231 (1996)
11. Google Inc., Google Maps API (2012)
12. Guttman, A.: R-trees: a dynamic index structure for spatial searching. SIGMOD Rec. **14**(2), 47–57 (1984)
13. Hariharan, R., Hore, B., Li, C., Mehrotra, S.: Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In: *SSDBM*, p. 16 (2007)
14. Hartigan, J.A., Wong, M.A.: Algorithm AS 136: a k-means clustering algorithm. Appl. Stat. **28**(1), 100–108 (1979)
15. Ho, C.-T., Agrawal, R., Megiddo, N., Srikant, R.: Range queries in OLAP data cubes. SIGMOD Rec. **26**(2), 73–88 (1997)
16. Jurgens, M., Lenz, H.-J.: The Ra*-tree: an improved R*-tree with materialized data for supporting range queries on OLAP-data. In: *DEXA*, pp. 186–191 (1998)
17. Lazaridis, I., Mehrotra, S.: Progressive approximate aggregate queries with a multi-resolution tree structure. SIGMOD Rec. **30**(2), 401–412 (2001)
18. Li, G., Feng, J., Xu, J.: DESKS: Direction-aware spatial keyword search. In: *ICDE*, pp. 474–485 (2012)
19. Li, Z., Lee, K., Zheng, B., Lee, W.-C., Lee, D.L., Wang, X.: IR-tree: an efficient index for geographic document search. TKDE **23**(4), 585–599 (2011)
20. Long, C., Wong, R.C.-W., Wang, K., Fu, A.W.-C.: Collective spatial keyword queries: a distance owner-driven approach. In: *SIGMOD*, pp. 689–700 (2013)
21. McCurley, K.S.: Geospatial mapping and navigation of the web. *WWW*, pp. 221–229 (2001)
22. Ponte, J.M., Croft, W.B.: A language modeling approach to information retrieval. In: *SIGIR*, pp. 275–281 (1998)
23. Rocha-Junior, J.a.B., Gkorgkas, O., Jonassen, S., Nørvåg, K.: Efficient processing of top-k spatial keyword queries. In: *SSTD*, pp. 205–222 (2011)
24. Srivastava, J., Tan, J., Lum, V.: TBSAM: an access method for efficient processing of statistical queries. TKDE **1**(4), 414–423 (1989)
25. Tao, Y., Papadias, D.: Range aggregate processing in spatial databases. TKDE **16**(12), 1555–1570 (2004)
26. Wu, D., Cong, G., Jensen, C.: A framework for efficient spatial web object retrieval. In: *VLDBJ, Online First*, p. 26 (2012)
27. Wu, D., Yiu, M.L., Jensen, C.S. , Cong, G.: Efficient continuously moving top-k spatial keyword query processing. In: *ICDE*, pp. 541–552 (2011)
28. Zhang, D., Chee, Y.M., Mondal, A., Tung, A., Kitsuregawa, M.: Keyword search in spatial databases: towards searching by document. In: *ICDE*, pp. 688–699 (2009)
29. Zhang, D., Ooi, B.C., Tung, A.: Locating mapped resources in web 2.0. In: *ICDE*, pp. 521–532 (2010)
30. Zhou, Y., Xie, X., Wang, C., Gong, Y., Ma, W.-Y.: Hybrid index structures for location-based web search. In: *CIKM*, pp. 155–162 (2005)
31. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comp. Surv. **38**(2), article no. 6 (2006). doi:10.1145/1132956.1132959