Location-Aware Type Ahead Search on Spatial Databases : Semantics and Efficiency

Senjuti Basu Roy[*]
University of Texas at Arlington
senjuti.basuroy@mavs.uta.edu

Kaushik Chakrabarti
Microsoft Research
kaushik@microsoft.com

ABSTRACT

Users often search spatial databases like yellow page data using keywords to find businesses near their current location. Such searches are increasingly being performed from mobile devices. Typing the entire query is cumbersome and prone to errors, especially from mobile phones. We address this problem by introducing type-ahead search functionality on spatial databases. Like keyword search on spatial data, type-ahead search needs to be *location-aware*, i.e., with every letter being typed, it needs to return spatial objects whose names (or descriptions) are valid completions of the query string typed so far, and which rank highest in terms of proximity to the user's location and other static scores. Existing solutions for type-ahead search cannot be used directly as they are not location-aware. We show that a straight-forward combination of existing techniques for performing type-ahead search with those for performing proximity search perform poorly. We propose a formal model for query processing cost and develop novel techniques that optimize that cost. Our empirical evaluations on real and synthetic datasets demonstrate the effectiveness of our techniques. To the best of our knowledge, this is the first work on location-aware type-ahead search.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Information Storage and Retrieval—*Information Search and Retrieval*[Search Process]

General Terms

Algorithms, Performance

1. INTRODUCTION

Spatial databases like yellow page data are often searched using keywords. For example, users use local search engines to find businesses, products or services *near their current location*. Hence, local search engines need to be *"location aware"*, i.e., the businesses are ranked not only by relevance to the keyword query but also by their proximity to user's location. Other factors like the popularity and ratings of businesses are also required to be taken into account [10, 6].

While many such searches are conducted from personal computers, users are increasingly using their mobile devices to perform such searches [1]. For example, users often need to find a nearby Starbucks or gas station on-the-go; they conduct such searches from their mobile devices like smart phones, where typing is cumbersome and susceptible to errors. We wish to reduce the amount of typing and clicking required before the user gets to see the final results.

In this paper, we propose to *progressively return relevant businesses as the user is typing in her query.* Consider a user looking for a nearby Starbucks. Suppose she has typed in the letters "star" on her mobile device. Note that the user's location can be obtained via a GPS-equipped mobile device. Based on that, it will be immensely useful to return the address and phone number of the nearest Starbucks, as it saves her from typing in the rest of the query. Furthermore, unlike in the case of autocompletion which returns likely query completions and requires the user to select the desired query, *no clicking is required* to see the final results here. This functionality is referred to as *type ahead search (TAS)* [14, 3]. In this paper, *we introduce the problem of supporting TAS on spatial databases.*

While TAS has been studied in the context of text documents and relational databases, to the best of our knowledge, ours is the first effort to solve TAS for spatial databases [14, 3]. What distinguishes TAS over spatial data from general TAS is that, like keyword search over spatial data, it needs to be *location-aware.* With every letter being typed, TAS should return k spatial objects whose names (or descriptions) are valid completions of the query string typed so far *and* which are closest to the user's location.

Semantics of Location-aware TAS: We define the semantics of TAS on spatial databases. Consider a spatial database containing names and locations of businesses shown in Table 1; locations are represented as points in 2-D space. We ignore the static score column for the time being. Suppose a user has typed in "star" so far from location (36, 0). O_7 and O_{10} are valid completions of the query. Suppose the user is interested in the top-1 result. TAS should return O_{10} as it is much closer to the user's location compared to O_7.

Note that it is crucial to consider spatial proximity *at a fine granularity.* Being proximity-oblivious for all objects inside a city might result in undesirable answers. For example, if we assume all the objects in Table 1 are located in the same city, a proximity-oblivious search might fail to distinguish between the nearest Starbucks O_{10} (for the previous query), and a much farther one O_7.

Furthermore, other factors like the popularity and ratings of the objects should also be considered; we model all these factors as a *single static score* associated with each object. Consider the query "shan" from location (37,3), and the corresponding valid completions (O_5 and O_6). Ignoring static score, we will get object O_6 as the top-1 result. But intu-

[*]Work done while visiting Microsoft Research

ID	String	Location	Static Score
O_1	Target	(3,9)	200
O_2	Thai Basil Leaf Restaurant	(50,30)	5
O_3	Sushi Rock	(9,50)	7
O_4	Sushi at Plano	(0,9)	25
O_5	Shanghai Cafe	(41,2)	500
O_6	Shanghai Garden	(38,5)	10
O_7	Starbucks	(32,8)	100
O_8	Super China Buffet	(42,5)	100
O_9	Staples	(45,12)	300
O_{10}	Starbucks	(35,0)	100

Table 1: Spatial Database $\mathcal{D}$ with 10 Objects

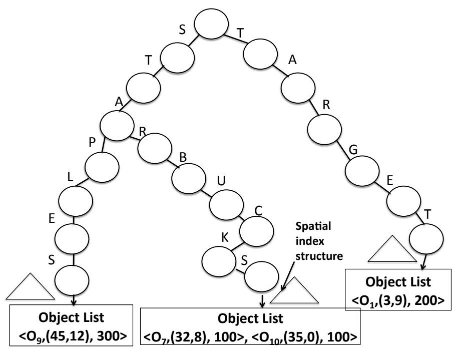

S
T
T
A
A
R
P
R
G
B
L
U
E
E
C
T
K
Spatial index structure
S
S
Object List <O₉,(45,12), 300>
Object List <O₇,(32,8), 100>, <O₁₀,(35,0), 100>
Object List <O₁,(3,9), 200>

Figure 1: Integrated Architecture for Location-Aware TAS

itively O_5 is a better answer, since it is a much more popular restaurant and is only slightly farther from the user's current location compared to O_6. In Section 2, we define semantics that takes both proximity and static scores into account.

Architecture: One may think that location-aware TAS can be supported by simply partitioning the entire geographical space into a set of spatial regions and using the location-unaware TAS solution (i.e., build a trie) for the objects in each region. In Section 3.1, we argue the limitations of that solution. We propose a novel architecture that *integrates the trie with a spatial data structure to enable location-aware TAS.* The basic idea is to maintain a single trie for the entire database, and augment it with spatial information of the objects. Such a trie for a subset of database in Table 1 (consisting of O_1, O_7, O_9, O_{10}) is shown in Figure 1. Benefits of such an architecture is discussed in Section 3.2. A simple baseline algorithm to support location aware TAS in this architecture is to first identify the trie node that matches with the query string, then traverse the entire subtree below it, compute the scores of the objects in that subtree, and finally return the top k objects based on the ranking function.

Improving Response Time: A TAS system has to be responsive – it must look instantaneous to the user. Prior work has shown that this implies a maximum response time of 100ms. In a client-server setting, this 100ms bound includes not only the trie search time but also network overhead. Hence, it is desirable to keep the trie search time minimal. The above baseline algorithm fails to meet that requirements, as it traverses too many links in the trie and scores too many objects. An efficient algorithm must avoid such unnecessary traversals and score computations.

One option is to materialize the top-k answers for each query prefix and for each query location; this is clearly infeasible due to the space overhead. A variant that maintains the top-k answers at the granularity of "regions" is also infeasible: the regions need to be granular enough to support the desirable semantics which results in high space overhead.

We propose to materialize *score bounds* at trie nodes: these are upper bounds of the scores of any object under that node. At query time, these bounds are used to *prune*, i.e., avoid traversal in parts of the trie that cannot contribute to the top-k results. Observe that the actual scores depend on the query location; therefore a single score bound per trie node is *not tight enough* for effective pruning – we need score bounds at a fine spatial granularity for effective pruning.

Since the trie needs to be memory resident, and the amount of main memory is limited, storing score-bounds at a fine spatial granularity at each trie node is not possible. We argue that we can materialize bounds only in a subset of trie nodes. Furthermore, we observe that *not all such subsets are equally beneficial* in saving *query processing cost.* For example, materializing at a parent and its child node in a trie with similar score bounds is less beneficial compared with choosing two nodes with significantly different upper bounds. Hence, it is important to select the subset *judiciously.* In addition, *the benefit of storing score bounds at a fine spatial granularity in much higher in some regions than in others.* For example, where the score-bounds exhibit high spatial locality, we do not need to store fine granularity score-bounds. Therefore, we need to be *adaptive* in our choice of granularity, i.e., store coarse-granularity bounds over regions that have high locality, and fine-granularity bounds elsewhere.

Summary of Contributions: The main technical challenge we address in this paper is: *how to distribute available memory among the nodes in the trie in order to minimize the expected query processing cost?* We refer to this as the *memory distribution problem.* We address this challenge by proposing a novel, rigorous modeling of query processing cost for TAS queries, and formalizing it as an optimization problem. Our contributions can be summarized as follows:

• We introduce the problem of location-aware TAS on spatial databases, define its semantics, and propose an integrated architecture for it (Section 3.2).

• We propose a novel analytical model for TAS query processing cost (Section 4.1)

• We formalize the memory distribution problem as an optimization problem (Section 4.4). Unfortunately, this problem is extremely hard as it is exponential in two factors: (i) number of trie nodes, and (ii) available space budget. As a reasonable alternative, we propose a slightly restricted problem: how to select a subset of M trie nodes and store R bounds in each of them such that expected query processing cost can be minimized? We refer to this as the $\{M, R\}$*-Memory Distribution Problem.* We show that the above problem can be split into two independent optimization problems:

(i) M**-node selection problem**: The problem is to select a set of M nodes where bound materialization would minimize the query processing cost. Based on that, we introduce the notion of "benefit" of materializing bounds at a set of nodes, and formalize the M-node selection as an optimization problem. We prove that the problem is NP-complete, and design a hill-climbing based algorithm as an efficient alternative (Section 5.1).

(ii)R**-cover computation problem**: The problem is to create a partitioning of size R of the entire geographic region such that the query processing cost is minimized. We formalize the problem as an optimization problem based on the popular maximum-error metric. We show the problem is NP-hard. We consider a novel, restricted version of the problem, where the rectangles are restricted to the regions

of a quadtree [11], and provide an optimal algorithm for it (Section 5.2).
• We propose a novel query processing algorithm. We prove its optimality in query processing cost (Section 6).
• Finally, we perform extensive experiments using real and synthetic dataset (Section 7). Our experiments show that our algorithms leveraging score bounds are *3-4 times faster* than the baseline algorithm that does not use score bounds.

2. SEMANTICS OF LOCATION-AWARE TAS

We start by introducing our data model, query model, proposed ranking framework, and finally define the location-aware TAS problem.

2.1 Data model

Let $\mathcal{D}$ be a spatial database. Each spatial object O is defined as the tuple $\langle O.id, O.str, O.loc, O.sscore\rangle$ where, $O.id$ is the unique id of that object, $O.str$ is the description string of O, $O.loc$ is the location descriptor in multidimensional space, and $O.sscore$ is the static score. In this work, we assume two dimensional space, i.e., $O.loc = (x, y)$ and describes x and y co-ordinates respectively. These coordinates can be derived from longitude and latitude information. Type ahead search is performed over $O.str$.

Consider a Yellow Page database containing names and locations of all businesses. There, $O.str$ denotes name of the object (this enables TAS over name), $O.loc$ is its geographical location, and $O.sscore$ is the overall score computed using a number of factors like the popularity, number of reviews, and ratings of the business.

Note that sometimes users search by the type of business instead of the name of the business. For example, users might type in "coffee shop" while looking for coffee shops. We would like the TAS system to return the nearest Starbucks, Tully's Coffee, and other coffee shops. Observe that these results will not be returned if $O.str$ is the name of the business (because the above names do not start with "coffee"). We can address this problem by associating, in addition to the name, strings that describe the type of business to the objects, and performing search over those strings.

We use the spatial database in Table 1 as a running example. In this example, we assume the business name is only description string associated with the $O.str$. Note that the same string can be associated with multiple objects (e.g., string "Starbucks" associated to all Starbucks stores).

2.2 Query Model

The TAS query interface is as follows: As the user types in the query, with every key stroke, the string typed so far is sent to the TAS system along with the user's location. In response, the TAS system returns the set of most *relevant k* spatial objects. The query Q therefore has two components: (i) the string typed so far; we denote it by $Q.str$, and (ii) the location $Q.loc$ of the user during query.

Furthermore, without loss of generality, we assume that the database $\mathcal{D}$ induces a global rectangular region $Global = \{ll, ur\}$ such that all objects in $\mathcal{D}$ are spatially contained in it. $Global.ll$ and $Global.ur$ denote the lower left and upper right corner respectively of $Global$. Furthermore, we assume $Q.loc$ is contained inside $Global$.

Note that the above query model corresponds to a stateless TAS system. In reality, the TAS system can be stateful: such a system can compute answers incrementally by using results of earlier queries, as the user types in more characters. For simplicity, we present our techniques in the context of a stateless TAS system. The general technique of materializing score bounds and using them to prune trie traversal applies to a stateful system as well. However, our query processing algorithm needs to be adapted to exploit cached results from earlier queries; we wish to explore that direction in future work.

2.3 Ranking Framework

Given a query Q and a spatial database $\mathcal{D}$, TAS should only return objects from $\mathcal{D}$ that are valid completions of $Q.str$. Any type-ahead search system must satisfy this basic property. We denote the set of such objects as $MatchSet(Q, \mathcal{D})$. Since $\mathcal{D}$ is fixed, for simplicity, we henceforth omit $\mathcal{D}$ from our notation. Formally,
$MatchSet(Q) = \{O | O \in \mathcal{D} \wedge Q.str \text{ is a prefix of } O.str\}$.

Among these objects, we want to return the k objects that are in *close proximity to the query location* and have high *static score*. We next describe such a ranking function.

Combination Model: Let $Dist(Q.loc, O.loc)$ denote the distance between the location of the query Q and object O. In this paper, we use the Euclidean distance function but any function that is monotone with respect to the distance along each dimension (i.e., x, y) can be used. In fact, all L_p distance functions satisfy this property.

The overall score of an object $O \in MatchSet(Q)$ for query Q is defined as $\mathcal{F}(Dist(Q.loc, O.loc), O.sscore)$, where $\mathcal{F}$ is a function monotone with respect to the two components. Although our techniques apply to the above class of scoring function, for simplicity, we describe our techniques in the context of the linear interpolation function proposed in [6, 10].

Specifically, the final score is a linear interpolation of the individual normalized scores of the two components:

$$\mathcal{F}(Q, O) = w_d \times (1 - \frac{Dist(Q.loc.O.loc)}{maxDist}) + w_s \times \frac{O.sscore}{maxSScore} \quad (1)$$

where w_d, w_s are the two parameters, s.t.$w_d + w_s = 1$. They allow the system designer to control the relative importance of the two components in the overall score. $maxDist$ is the maximum distance between any object and query and $maxSScore$ is the maximum static score of any object in the database. We compute $maxDist$ as $Dist(G.ll, G.ur)$. $maxDist$ (similarly $maxSScore$) is used to normalize the distance score (static score).

Note that our study focuses on efficient techniques for type-ahead search; hence, we adapt existing distance functions instead of developing new ones.

2.4 Problem Statement

Given a query $Q = (str, loc)$, a spatial database $\mathcal{D}$, and an integer k, identify the result set $Res(Q, k)$ of objects, such that,
(i) $|\text{Res(Q,k)}| = k$
(ii) $\forall O \in Res(Q, k), O \in MatchSet(Q)$, and
(iii) the objects in $Res(Q, k)$ have the highest scores among all objects in $MatchSet(Q)$, i.e.,for any object $O \in Res(Q, k)$ and any object $O' \in MatchSet(Q) - Res(Q, k)$, $\mathcal{F}(Q, O) \geq \mathcal{F}(Q, O')$.

3. ARCHITECTURES FOR LOCATION AWARE TAS

Standard (i.e., non location-aware) TAS systems require ordered tree data structure such as *trie*. Type ahead search is enabled by inserting all search strings into a trie $\mathcal{T}$. Let

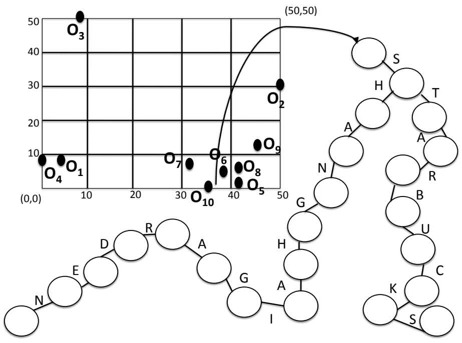


Figure 2: Partition Space Build Trie for Location-Aware TAS

$n_{\mathcal{T}}$ be a node in $\mathcal{T}$, and Subtree($n_{\mathcal{T}}$) is the entire subtree under it.

Note that each trie node $n_{\mathcal{T}}$ can be uniquely identified by a string (denoted by $str(n_{\mathcal{T}})$); that string corresponds to the labeled path from the root node to $n_{\mathcal{T}}$. For example, in the trie in Figure 1, the string identifying the left child of the root is "s", the one identifying its (only) child is "st", and so on. Searching is performed by first identifying the node $n_{\mathcal{T}}$ that matches the query string, i.e., $str(n_{\mathcal{T}}) = Q.str$. This node is henceforth referred to as the *query matched node*, denoted by QMN(Q). Subsequently, Subtree(QMN(Q)) is searched to obtain the results. Existing work offers several optimized version of trie, such as Patricia Tree [7], and we adopt those optimized versions. Now, to make the search location aware, these are the two possibilities.

3.1 Partition Space then Build Trie

One possibility is to first partition the global space *Global* into a set of spatial regions. Maintain a trie containing description strings of the objects inside each region. Given a query Q, the system first identifies the region that contains the query location. It then performs a standard trie search operation inside that region to identify $Res(Q,k)$. $Res(Q,k)$ is determined solely based on static scores of the objects, ignoring distance. Figure 2 corresponds to this architecture, where 25 spatial regions of $\mathcal{D}$ are created (each of dimension 10×10), and the trie corresponding to the region $\{(30,0),(40,10)\}$ is shown.

Although simple and similar in principle to standard TAS architecture, this architecture suffers from several potential demerits:

Missing cross-boundary objects: Recall the example "shan" in Section 1. This architecture first determines the $Q.loc$ specific spatial region $\{(30,0),(40,10)\}$, then performs trie search and returns O_6 as the top-1 completion. A more desirable result O_5 will be missed as it fails to consider cross-boundary objects.

Inability to support desired semantics: The ranking of an object inside a spatial region is oblivious to distance. Now recall example "star" in Section 1, the above architecture fails to distinguish between farther Starbucks O_7 and the nearer one O_{10}, and may return either of them. Observe that, a very fine-granularity partition (e.g., one partition per zip code) may mitigate such effects to some extent. However, very fine granularity partitions are extremely space redundant as they require to replicate the same string numerous times (in each spatial region specific trie). Limited availability of main memory makes this possibility unrealistic in practice.

3.2 Integrate Trie with Spatial Data Structure

An alternative possibility is to maintain a single trie over all description strings of objects in $\mathcal{D}$. Each trie leaf contains a list of objects (denoted as *object list*). Each object is described by $\langle O.id, O.loc, O.sscore\rangle$. Observe that, any object whose description string is the string identifying the node $l_{\mathcal{T}}$, is in the object list of $l_{\mathcal{T}}$. Figure 1 shows this architecture over a subset of objects O_1, O_7, O_9, and O_{10} from the database in Table 1.

Given a query Q, the search begins in $\mathcal{T}$ by identifying QMN(Q). $Res(Q,k)$ is computed by visiting each leaf $l_{\mathcal{T}}$ in Subtree(QMN(Q)) and resolving complete scores of all objects encountered there, and finally selecting k objects that contain the overall k-highest scores. The algorithm uses a global priority queue `ResultsPQ` to maintain the k-objects with highest scores encountered thus far. `ResultsPQ` is updated as the score of an object in a leaf node is computed (denoted as `LeafNodeSearch`). The pseudocode is listed in Algorithm 1.

Algorithm 1 Baseline-TAS -Baseline Algorithm for location-aware TAS

```
Require: Trie $\mathcal{T}$, Query $Q$, an integer $k$
1: ResultsPQ = $\phi$
2: Perform lookup in Trie $\mathcal{T}$ to determine QMN(Q)
3: for each leaf $l_{\mathcal{T}} \in Subtree(QMN(Q))$ do
4:     Execute LeafNodeSearch($l_{\mathcal{T}}$) and update ResultsPQ
5: return $Res(Q,k)$ from ResultsPQ
```

The benefit of this integrated architecture is various: first, this architecture is able to support any ranking function. Note that, for both example queries, it will be able to return the desired semantics. Second, this architecture is optimum in space requirement. Hence we adopt this integrated architecture.

Efficient Implementation of `LeafNodeSearch`: The above implementation of `LeafNodeSearch` scans the entire object list in the leaf node, and computes scores of all the objects. This can be expensive if the object list is large. This can be avoided by using the threshold family of algorithms [9]. NRA can be used if only sorted accesses on each individual ranking component (a getNext() interface on static score and distance[1]) is provided. A TA [9] based solution can be used and may terminate even earlier if random access is also available on static score table. The k-th largest score in `ResultsPQ` is used to determine the termination condition of TA. Baseline-TAS is used as our baseline algorithm with TA like implementation for `LeafNodeSearch`.

Limitations: Algorithm `Baseline-TAS` still visits each leaf in QMN(Q), and performs several score computations in each leaf.

4. QUERY PROCESSING USING INTEGRATED ARCHITECTURE

Given a query Q, the goal is to return the k best objects ($Res(Q,k)$) in $MatchSet(Q)$ according to a ranking function $\mathcal{F}$, as stated in Subsection 2.4.

4.1 Query Processing Cost Model

We begin our discussion by formally defining processing cost to determine $Res(Q,k)$ using integrated architecture.

[1]main memory based spatial data structures like KD-tree, quad tree can enable efficient getNext() interface on distance

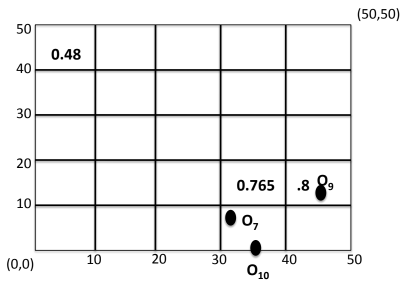


Figure 3: SGB at node "STA" of $\mathcal{T}$

Recall that, in order to determine $Res(Q, k)$, node QMN(Q) needs to be determined first. The next task is to visit the leaf nodes following the links of Subtree(QMN(Q)), and finally compute scores of the objects encountered there.

Let, $\mathcal{L}_{\{Q,k\}}$ be the total number of links traversed, and $Sc_{\{Q,k\}}$ be the total number of objects for which score is computed during query processing.

DEFINITION 1. **(Query Processing Cost Model)** `QPCost(Q,k)` = `Cost(QMN(Q))` + $c_l \times \mathcal{L}_{\{Q,k\}}$ + $c_{sc} \times Sc_{\{Q,k\}}$, *where* c_l *and* c_{sc} *are two constants, that denote unit link traversal, and score computation costs respectively.*

Observe that, any algorithm has to incur Cost(QMN(Q)) and is negligible compared to other two costs. Hence, we ignore Cost(QMN(Q)), and focus only upon the improvement of link traversal and score computation cost.

4.1.1 *Query Processing Cost of* `Baseline-TAS`

One may notice that during computation of the $Res(Q, k)$, Algorithm 1 needs to traverse entire Subtree(QMN(Q)). Let, $|Subtree(QMN(Q))|$ be the subtree size, and $Obj_{l_\mathcal{T}}$ be the number of objects processed at leaf $l_\mathcal{T}$ during leaf node search.

QPCost(Q,k) = $(c_l \times |Subtree(QMN(Q))|)$
$+ \Sigma_{\forall l_\mathcal{T} \in Subtree(QMN(Q))}(c_{sc} \times Obj_{l_\mathcal{T}})$.

We argue that returning top-k valid completions are most useful to a user during few initial query key strokes, especially in the case of mobile applications, where typing is cumbersome and error-prone. Unfortunately, the size of the Subtree(QMN(Q)) is prohibitively large in the beginning. `Baseline-TAS` must traverse that entire subtree, and hence incurs substantial query processing cost. Moreover, many of the link traversal and score computation get wasted as only a very small fraction of the leaf nodes of Subtree(QMN(Q)) eventually contributes to the set $Res(Q, k)$. We note that an efficient query processing technique must intelligently select those paths of Subtree(QMN(Q)) for traversal such that they eventually contributes to the final $Res(Q, k)$. Next, we discuss these possibilities in detail.

4.2 Techniques to Reduce Query Processing Cost

Note that, link traversal and score computation cost can be significantly reduced, if the actual $Res(Q, k)$ is pre-computed and materialized for each intermediate node of $\mathcal{T}$. Unfortunately, such techniques *immediately fail* in our case, since, i) $Res(Q, k)$ varies based on query location according to our ranking framework, and ii) Materializing $Res(Q, k)$ for every possible location at every intermediate node is impossible to follow in practice. An alternative possibility is to materialize at a very fine spatial granularity to support desirable ranking semantics. However, such granularity requires humongous space that our main memory based integrated architecture fails to support.

We therefore propose materialization of *score-bound* at a trie node, that denotes the *maximum-score* any object under that subtree may get, when a query matches that node. Note that, the challenge is to ensure that *score-bound* is correct *irrespective of query location.*

EXAMPLE 1. **(Single Score Bound)** *Consider the spatial database in Table 1, and the corresponding partial trie in Figure 1. Observe that, the object list under the trie node "STA" contains* O_7, O_9, *and* O_{10}. *The object with highest static score* (O_9) *determines the score-bound of that node. Note that, score-bound of an object needs to be computed considering the maximum-score for the distance component that occurs when* $Q.loc = O.loc$. *The score-bound of static score component of* O_9 *after normalization becomes* 0.3, *whereas, that of distance component is* 0.5, *with an overall score-bound of* 0.8 *after addition (assuming* $w_d = w_s = 0.5$ *in Equation 1).*

Query processing algorithm benefits from pre-computed score-bound in the following way: if the score-bound of a node in Subtree(QMN(Q)) is not larger than the k-th largest object score (computed thus far during query processing), the entire subtree under that node can be pruned, i.e., can be saved from traversal. Therefore, score-bound at a node needs to be *tight* for effective pruning.

However, the actual score of an object under a node may be significantly smaller than the pre-computed score-bound of that node. In fact, in the previous example, actual score of O_9 will never reach 0.8, unless $Q.loc = O_9.loc$. For any other $Q.loc$, the distance component score will be smaller than 0.5. Therefore, we argue that storing *only one* score-bound per trie node may not be *enough* since that may fail to generate sufficiently *tight* score-bounds. But also, storing score-bounds at the finest granularity is an impossible task, especially because the entire data structure needs to reside in the main memory during query processing.

In this work, we take up an intermediate approach by partitioning *Global* into a set of spatial regions and storing score-bounds for them.

The set of regions together must satisfy the *cover* property over *Global*, i.e., each point location in *Global* must be contained in one of the regions. We begin by considering a special case of cover at a node, where the regions are non-overlapping, equal size rectangles. Such a cover at node $n_\mathcal{T}$ with bounds is referred to as the *spatial grid of bounds* (henceforth denoted as SGB($n_\mathcal{T}$)). Each region of a spatial grid ($\mathcal{G}$) is referred to as a *grid cell* or simply a *cell* (g). The granularity of a spatial grid is determined by a domain expert, and we assign score-bounds for them. Next we define *Score-bound* of a grid cell g.

DEFINITION 2. **(Score-bound of a grid cell)** *It is the maximum-score of any object under* $n_\mathcal{T}$ *(for any Q.loc inside g).*
Score-bound$(g, n_\mathcal{T})$ = $max_{\forall O_i \in Subtree(n_\mathcal{T})}$ *Max-Score*(O_i), *s.t.* $g.ll.x \leq Q.loc.x \leq g.ur.x$ *and* $g.ll.y \leq Q.loc.y \leq g.ur.y$

EXAMPLE 2. **(SGB)** *Consider Figure 3 that shows an SGB of the node "STA" with granularity* 10×10. *Observe that only the grid cell*

$\{(40,10),(50,20)\}$ *has score-bound* 0.8*, whereas the grid-cell* $\{(0,40),(10,50)\}$ *contains a much smaller score-bound of* 0.48*. A query that is contained in the latter cell thus acquires tighter score-bound (as compared to a single score-bound) by the SGB.*

4.3 Main Insights

• **Judicious Node Selection:** It is *not possible* to materialize SGB at every trie node. Only a subset of trie nodes can be materialized. Different subset of nodes offers different *benefit* towards query processing. For example, materializing a parent and child node in a trie with similar score bounds is less beneficial compared to choosing two nodes with significantly different score bounds. Therefore, judicious node selection is important.

•**Adaptive Cover Computation:** Consider an example trie node that contains only one object with very high static score (all other objects have very low static scores), and assume $w_s \gg w_d$ in the ranking function. Most likely that object will influence the score-bounds of such a node for most $Q.loc$. Therefore, the score-bounds will exhibit very high spatial locality. Creating partitions of finer granularity is not useful for such nodes. However fine granularity partitions may be beneficial for a case, that shows substantial variation in score-bounds at different $Q.loc$. Therefore, cover computation needs to be adaptive.

4.4 Memory Distribution

Intuitively, our objective is to minimize the expected cost of top-k query processing on any query, using the materialized trie.

Let $\mathcal{N_T}$ be the set of intermediate nodes in a trie $\mathcal{T}$, and S be the space available for materialization. Let $p(Q)$ denote the probability or likelihood of Q being issued [2]. The formal problem is stated as follows.

Generalized Memory Distribution Problem:
Minimize $\Sigma_{\forall Q} p(Q) \times QPCost(Q)$
such that $(0 \leq s_{n_\mathcal{T}} \leq S)$, and $\forall_{n_\mathcal{T} \in \mathcal{N_T}} (\Sigma s_{n_\mathcal{T}} \leq S)$,
where $s_{n_\mathcal{T}}$ is the allocated space for materialization at the intermediate trie node $n_\mathcal{T} \in \mathcal{N_T}$.

Although clearly very important, this problem is unfortunately quite challenging to solve optimally, since it is exponential in both S and $\mathcal{N_T}$. As a first step toward addressing these challenges, we consider a restricted version, where the possible assignments for $s_{n_\mathcal{T}}$'s are either 0 (no materialization) or R (a predefined number designed by the domain expert), and our task is to select the set of M nodes such that each of them has R space allocation, s.t., $S = M \times R$. With this simplified assumption, we define the $\{M, R\}$ Memory Distribution Problem as follows.

Definition 3. **($\{M, R\}$ Memory Distribution Problem)**
Minimize $\Sigma_{\forall Q} p(Q) \times QPCost(Q)$
s.t., $s_{n_\mathcal{T}} = 0$ *or* R*, and* $|\{n_\mathcal{T} | (s_{n_\mathcal{T}=R})\}| = M$*, where* $s_{n_\mathcal{T}}$ *is the allocated space at the trie node* $n_\mathcal{T} \in \mathcal{N_T}$*.*

5. $\{M, R\}$ DISTRIBUTION PROBLEM SOLUTION

Note that, the optimal cover of size R (R cover in short) can be computed independently at a trie node, while the M node selection process needs to use R-cover to determine the set $\mathcal{M}$ (s.t., $|\mathcal{M}| = M$). A trie with M materialized nodes, where each materialized node has R-cover is known as bound materialized trie $\mathcal{T_{BM}}$.

We begin our discussion by illustrating the M Node Selection problem, then discuss R cover computation problem.

5.1 M Node Selection

We aim at modeling the *benefit* of a set $\mathcal{M}$ of materialized trie nodes for reducing $QPCost$, and then pose the optimization problem.

We begin by analyzing the *benefit* of a single materialized node $n_\mathcal{T}$ in a set of materialized nodes for a query Q. Materialization at a node $n_\mathcal{T}$ benefits only those queries whose QMN(Q) is an ancestor of $n_\mathcal{T}$. *Benefit* of materializing at $n_\mathcal{T}$[3] for an ancestor $n'_\mathcal{T}$ is modeled by considering the following two aspects.

Likelihood of Pruning - Suppose $Q.loc$ is known. Let, g and g' be the cells in SGB($n_\mathcal{T}$) and SGB($n'_\mathcal{T}$) respectively that contains $Q.loc$. Therefore, akin to actual pruning, $\frac{score-bound(g', n'_\mathcal{T})}{score-bound(g, n_\mathcal{T})}$ approximates the relative value of actual k-th largest score at $n'_\mathcal{T}$, and the score-bound at $n_\mathcal{T}$; a larger ratio increases the *likelihood* of pruning. Observe that, in this model, we are required to capture this *likelihood* for any $Q.loc$. Therefore, the likelihood of pruning is measured by considering the ratio of Expected-Score-bound($n'_\mathcal{T}$), and Expected-Score-bound($n_\mathcal{T}$), by leveraging location distribution information of past $n'_\mathcal{T}$ queries from query logs.[4]

Cost Save if Pruned - Next, we capture how much $QPCost$ node $n_\mathcal{T}$ *saves* upon materialization (irrespective of its ancestors), considering rest of the materialized nodes in set $\mathcal{M}$ that are $n_\mathcal{T}$'s descendants. We define the *materialized frontier* of a node $n_\mathcal{T}$ in that context.

Definition 4. **(Materialized Frontier)**
MFr($n_\mathcal{T}$) - The nearest *set of materialized descendant nodes of* $n_\mathcal{T}$ *together creates a materialized frontier.*

Figure 4 shows a materialized frontier of $\mathcal{T}$ root, and a materialized frontier of node "STA".

Note that, without $n_\mathcal{T}$ being materialized, query processing algorithm must proceed further down and traverse the entire Subtree($n_\mathcal{T}$), except the subtree of $n_\mathcal{T}$'s materialized frontier. We argue that $n_\mathcal{T}$ saves higher $QPCost$ upon materialization, if $QPCost(n_\mathcal{T})$ is much larger than the query processing cost of its materialized frontier[5].Therefore,

Definition 5. **(Cost-save)**
CostSave($n_\mathcal{T}$) = QPCost($n_\mathcal{T}$) - $\Sigma_{\forall n''_\mathcal{T} \in MFr(n_\mathcal{T})} QPCost(n''_\mathcal{T})$

As an example, in Figure 4 for node "STA" $QPCost(n_\mathcal{T})$ is 13 and $\Sigma_{\forall MFr(n_\mathcal{T})} QPCost(MFr(n_\mathcal{T}))$ is 9 (assuming $c_l = 1$ and $c_{sc} = 1$). A larger CostSave denotes higher worth of $n_\mathcal{T}$'s benefit.

Finally, given an ancestor $n'_\mathcal{T}$, likelihood of pruning at $n_\mathcal{T}$ and CostSave($n_\mathcal{T}$) are multiplied to denote $n_\mathcal{T}$'s *benefit* towards $QPCost(n'_\mathcal{T})$. Then, we sum it over each ancestor of $n_\mathcal{T}$ to compute $n_\mathcal{T}$'s total *benefit*.

Similarly, the *benefit* of a set of nodes is the sum of *benefit* of each node in that set.

Definition 6. **(Benefit)**
$Benefit(\mathcal{M}) = \Sigma_{\forall n_\mathcal{T} \in \mathcal{M}} Benefit(n_\mathcal{T}, \mathcal{M})$*, and*
$Benefit(n_\mathcal{T}, \mathcal{M}) = \Sigma_{\forall_{n'_\mathcal{T}}} \frac{Expected-Score-bound(n'_\mathcal{T})}{Expected-Score-bound(n_\mathcal{T})} \times CostSave(n_\mathcal{T})$
s.t., $n'_\mathcal{T}$ *is an ancestor of node* $n_\mathcal{T}$*.*

[2]can be derived from query log

[3]$n'_\mathcal{T}$ and Q will be used interchangeably in this discussion.
[4]We compute Average-Score-bound in the absence of a query log.
[5]Note that, Subtree $n_\mathcal{T}$ may not even have any materialized frontier.

Algorithm 2 `Randomized` M `Node Selection` : Algorithm to compute the best set of M nodes

```
Require:
    $\mathcal{N}_{\mathcal{T}}$ - set of intermediate trie nodes, NumReStart - number
    of random restart, $M$ - the number of selected nodes
 1: $\mathcal{M} = \{\}$, $PrevSet = \{\}$ , $PrevBenefit = 0$, and $i = 1$
 2: repeat
 3:    $\mathcal{M}$ = {a randomly chosen set of $M$ trie nodes}
 4:    while $AllNeighborSet(\mathcal{M})$ are not visited do
 5:       $\mathcal{M}'$ = Remove one node from $\mathcal{M}$ uniform randomly, and
          replace that with a node chosen uniform randomly from
          $\{\mathcal{N}_{\mathcal{T}}\} - \{\mathcal{M}\}$.
 6:       $\mathcal{M} = \mathcal{M}'$ if $Benefit(\mathcal{M}') > Benefit(\mathcal{M})$
 7:    if $PrevBenefit < Benefit(\mathcal{M})$ then
 8:       $PrevSet = \mathcal{M}$
 9:       $PrevBenefit = Benefit(\mathcal{M})$
10:    $i = i + 1$
11: until $\{i > NumReStart\}$
12: $\mathcal{M} = PrevSet$
13: return $\mathcal{M}$
```

Optimization Problem - M **Node Selection**:
Given a set of $\mathcal{N}_{\mathcal{T}}$ trie nodes, select a set $\mathcal{M}$ ($|\mathcal{M}| = M$), s.t. $Benefit(\mathcal{M})$ is maximized.
Next, we argue that identifying the best set $\mathcal{M}$ according to the optimization problem is provably challenging, since this requires to solve an NP-complete problem.

Theorem 1. **(Hardness Result)**
Finding the set $\mathcal{M}$ *which maximizes* $Benefit(\mathcal{M})$ *is NP-Complete.*

NP-completeness is proved using a reduction from the set cover problem [12], using a very simple benefit function. Due to lack of space, we omit the details of that proof from this version of the paper. Unfortunately, an approximation algorithm for set cover does not lend to our problem easily, since the actual benefit of a node is much complicated in reality.

Efficient Algorithm
As a reasonably efficient alternative of this hard problem, we propose a *randomized hill climbing* algorithm to search for a local optima, starting from a random $\mathcal{M}$ set and computing its associated benefit. At each step, the algorithm goes to a "neighboring set" of $\mathcal{M}$, by swapping one random node from the existing $\mathcal{M}$ set that results in higher $Benefit(\mathcal{M})$. One complete hill climbing process terminates when all swapping possibilities of a set $\mathcal{M}$ are explored. We design a random-restart approach (with a predefined number, `NumReStart`) on top of the hill-climbing method, that iteratively performs hill-climbing search, each time with a random initial set $\mathcal{M}$. The hill climbing results $\mathcal{M}$ that corresponds to the highest $Benefit(\mathcal{M})$ is retained after all iterations. Note that, random-restart enhances the chances of obtaining the global optima. The pseudo-code of this algorithm is listed in Algorithm 2.

5.2 R Cover Computation

We discuss the R cover computation problem at a trie node. We perform the following two tasks:
• Compute SGB of grid $\mathcal{G}$.
• If the number of cells in SGB is more than R, we compute an R cover on that.

5.2.1 SGB Computation

Recall that, score-bound of a grid cell at a trie node is the maximum *Max-Score* of each satisfying object under that node for any *Q.loc* in that cell. Max-Score of an object can be efficiently computed considering MINDIST [19][6].

[6]Although, MINDIST is defined using Euclidean distance here, our solution framework can be generalized to any L_p distance metric

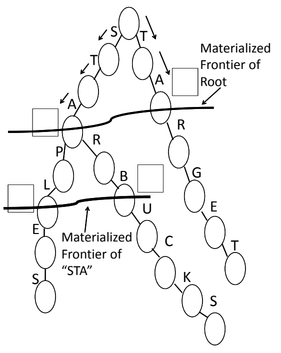


Figure 4: Query Processing Using Bound-Materialized Trie

$$\text{Max-Score}(O,g) = w_d \times \left(1 - \frac{MINDIST(O.loc,g)}{maxDist}\right) + w_s \frac{O.sscore}{maxSScore}$$

Definition 7. **(MINDIST)**
MINDIST$(O,g) = |O.x - g_{r.x}|^2 + |O.y - g_{r.y}|^2$, *where,* $g_{r.x}$ ($g_{r.y}$)
= g.ll.x (g.ll.y) if O.loc.x (O.loc.y) < g.ll.x (g.ll.y)
= g.ur.x (g.ur.y) if O.loc.x (O.loc.y) > g.ur.x (g.ur.y)
= O.loc.x (O.loc.y) otherwise.

Given a cell g at $n_{\mathcal{T}}$, Max-score needs to be computed for each object under $n_{\mathcal{T}}$ to compute Score-bound$(g, n_{\mathcal{T}})$. Then, score-bound needs to be computed for every g in the grid $\mathcal{G}$. Observe that, a naive computation is quadratic in the number of objects, and the number of cells. Although done in preprocessing, repeating this quadratic computation at every trie node is impractical.

We propose a novel solution at this juncture. Given a cell g under $n_{\mathcal{T}}$, the Score-bound$(g, n_{\mathcal{T}})$ is the *highest* (top-1) *Max-Score* of an object under $n_{\mathcal{T}}$. Observe that, the overall score of an object is a monotonic combination of distance score and static score, where distance score is again monotonic along x and y dimensions. Therefore, Threshold algorithm [9] (TA) style computation can be enabled, and Score-bound$(g, n_{\mathcal{T}})$ may be computed without computing Max-score of all objects under $n_{\mathcal{T}}$. Three lists are to be used for the objects under $n_{\mathcal{T}}$ during TA - `List.x` (sorted in increasing x-coordinate distance), `List.y` (sorted in increasing y-coordinate distance), `List.sscore` (sorted in decreasing static score). As in TA, each entry in the list is an object id, its complete score can be resolved using the object database $\mathcal{D}$.

To compute the score-bound of a cell g, we perform region specific TA, by identifying *interesting regions* of g, that we define next.

Definition 8. **(Interesting Regions)**
interesting regions(g) = Partitions of Global (Global.ll, Global.ur) wrt g(g.ll, g.ur).

Consider Figure 5, that shows interesting regions of a cell g for the root node of the trie $\mathcal{T}$.

First of all, the score-bounds of the internal regions of all cells in a grid can be computed efficiently - a single scan over $\mathcal{D}$ is sufficient to assign a score-bound in internal region of each cell in $\mathcal{G}$.

We explain the score-bound computation of g that has 9 interesting regions (8 external, 1 internal) next. For each external region of g, we perform region specific TA. We now

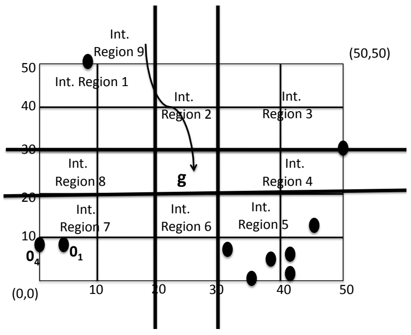


Figure 5: Computing Score-bound(g) at $\mathcal{T}$ root

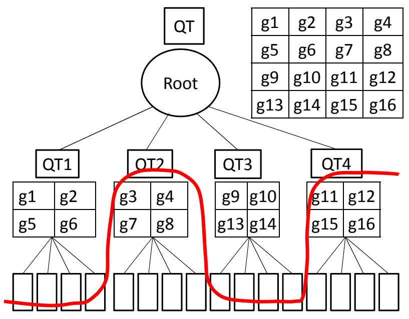
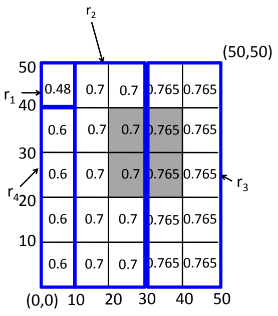


Figure 7: Quad-Tree of a SGB($n_{\mathcal{T}}$)



0.48	0.7	0.7	0.765	0.765
0.6	0.7	0.7	0.765	0.765
0.6	0.7	0.7	0.765	0.765
0.6	0.7	0.7	0.765	0.765
0.6	0.7	0.7	0.765	0.765

Figure 6: R Cover Creation

describe how score-bound of region specific TA at Int.Region 7 can be computed. Note that, only objects O_1 and O_4 are to be considered during `RegionTA` computation of Int.Region 7, and *MINDIST* needs to be computed considering the closest boundary point of g (which happens to be $g.ll$), and location of the objects inside Int.Region 7 (O_1, O_4). `RegionTA` in other interesting regions will follow similar argument. Finally, Score-bound(g) is the maximum of these 9 interesting region specific Max-Score.

5.2.2 R Cover Creation

Given an SGB($n_{\mathcal{T}}$), the task is to create a cover with R regions. A natural approach of creating R cover is by merging cells of SGB($n_{\mathcal{T}}$). An R cover must satisfy the following conditions: the score-bound of every region $r \in R$ (Score-bound($r, n_{\mathcal{T}}$)) is correct.

Observe that, in order to satisfy this condition, Score-bound($r, n_{\mathcal{T}}$) must be the *maximum* of the score-bounds of the cells inside it, as Lemma 1 suggests.

LEMMA 1. **(Score-bound of a region)** *Score-bound($r, n_{\mathcal{T}}$) = $max_{\forall g \in r}$ Score-bound($g, n_{\mathcal{T}}$).*

Due to lack of space, we omit the details of the proof from this version of the paper.

As an example, consider the SGB in Figure 6 (the bound values are imaginary numbers here). The correct score-bound of the region (grey filled) is 0.765 - hence, the original grid cells with 0.7 bounds now get a higher score bound of 0.765. In other words, merging introduces *looseness* in the score-bound and is detrimental to effective pruning. But note that, different merging induces different looseness. For example, if the cell with score-bound 0.48 is merged with the cell with bound 0.7, that induces more looseness for the former cell, as compared to merging 0.6 score-bound cell with 0.7.

We define *error* in that context. For a given g, that is merged inside a region r, the difference in score-bound between these two is denoted as g's *error*. Note that, error captures looseness in bounds, hence may affect the effectiveness of pruning, but not the correctness of results.

While it is mandatory to maintain the correct score-bound in r, it is also important to preserve *tight* bounds for effective pruning. Therefore, a cover creation algorithm must adaptively form R regions. Our next optimization problem intends to create R cover such that it minimizes the *maximum-error* of the regions in the cover.

DEFINITION 9. **(Maximum-Error of a region)** *Maximum-error($r, n_{\mathcal{T}}$) = $max_{\forall g \in r}$ Score-bound($g, n_{\mathcal{T}}$) - $min_{\forall g' \in r}$ Score-bound($g', n_{\mathcal{T}}$)*

Optimization Problem - R Cover Creation : Given an SGB($n_{\mathcal{T}}$), create R cover such that,

$max_{\forall_{r \in R}}$Maximum-error($r, n_{\mathcal{T}}$) is minimized.

Although seemingly practical and important, optimally solving R Cover Creation problem is unfortunately very challenging - since it requires us to solve an NP-complete problem. A direct mapping exists between this known NP-hard [15] problem and our problem. Furthermore, any reasonable approximation algorithm is also unknown. The difficulty comes from the fact that it needs to consider exponential number of possible region formations to compute optimal cover. Figure 6 shows an optimal cover of size 4 (r_1, r_2, r_3, r_4), where each cell has 0 error.

THEOREM 2. **(Hardness Result)** *Computing optimal cover of size R based on the maximum-error metric is NP-complete.*

We consider a viable alternative of this hard problem by restricting the choice of possible regions to the nodes of a quad tree `QT` of SGB. The benefit of quad-tree [11] is, it allows us to design an *optimal algorithm* to select R cover. In a nutshell, the basic idea is to create a `QT` of SGB($n_{\mathcal{T}}$) (the leaf nodes of `QT` are the actual cells), and select a R-size frontier of it that optimizes the maximum-error metric. Figure 7 depicts a quad-tree with 16 cells, and a size 10 frontier (with the wriggly line) of it.

Algorithm to Select R-size Frontier of `QT`: Algorithm 3 selects R-size Frontier - initially the cover (referred to as `Cover` in the pseudo-code) consists of the leaf nodes of `QT`, therefore has 0 error. The algorithm begins by sorting the intermediate nodes of `QT` in the increasing order of maximum-error. At a certain point, if the size of the current cover is larger than R, the algorithm selects the "best" (with lowest maximum-error) available intermediate node, and add that to the existing cover. Each time a new node is added in the cover, all its immediate children nodes are removed from there; thus this operation overall reduces

Algorithm 3 Select$R$Frontier : Algorithm to Select R-size Frontier of QT

```
Require:
        QT - a quad tree of SGB($n_{\mathcal{T}}$), $R$ - an integer
 1: Cover = { leaf nodes of QT}
 2: $\mathcal{I}$ = { set of intermediate QT nodes, sorted in increasing order
    of error }
 3: while |Cover| $\neq (R \pm 3)$ do
 4:     Consider the first node $\mathcal{I}_1 \in \mathcal{I}$
 5:     $\mathcal{I} = \{\mathcal{I}\} - \{\mathcal{I}_1\}$
 6:     Cover = {Cover} + $\{\mathcal{I}_1\}$ − {Children$_{\mathcal{I}_1}$}
 7: return Cover
```

the size of the cover by 3. Also, the maximum-error of the cover at that point is the maximum-error of that newly-added node. The algorithm terminates when cover has R regions[7].

It can be shown that Algorithm Select$R$Frontier is an optimal algorithm, as the Lemma 2 suggests.

LEMMA 2. **(Optimality)**
Algorithm Select$R$Frontier *is optimal for maximum-error metric.*

6. EFFICIENT QUERY PROCESSING

Finally, we discuss algorithm (BMT-TAS) that leverages T_{BM} to compute $Res(Q, k)$. The pseudo-code is written in Algorithm 4.

Consider the case, where each node in the trie contains materialized bounds. In such cases, the algorithm BMT-TAS can follow the best-first search [7] like execution fashion of nearest-neighbor queries in multi-dimensional index structure, that has been shown to be optimal [19].

However, T_{BM} is unique, since only a judiciously selected set of M nodes in the trie contains materialized information. Therefore, we employ a *hybrid-approach*, that performs *depth-first search* [7] traversal in a subtree rooted at a materialized node (SRM), but performs best-first like traversal across SRMs. Lines 7-15 in the pseudocode perform these two operations.

Algorithm BMT-TAS maintains two data-structures in this process - a stack for depth-first search, and a *priority queue* for best-first search. The current best-k results are maintained in a global priority queue ResultsPQ. During depth-first search (inner loop), if the algorithm encounters a materialized node, that node is pushed into the priority quere, with the priority as its score-bound [8]. Best-first search is performed in the outer loop across SRMs. During best-first search, the algorithm first obtains the top item in the priority queue, i.e., the unexplored SRM with the highest score bound. Next it determines if the termination condition (if the k-th largest object score is not smaller than the score bound of the top unexplored SRM) is satisfied. If satisfied, it terminates, otherwise, it continues with exploring the top unexplored SRM. This process terminates automatically when the entire $Subtree(QMN(Q))$ is traversed.

It can be proved that algorithm BMT-TAS is optimal in query processing cost.

LEMMA 3. **(Optimality)**
BMT-TAS is an optimal algorithm in query processing cost.

[7]We note that for certain cases, the resultant frontier size may vary by ±3 from R - this happens due to the restricted quad-tree structure and we allow such a negligible deviation

[8]Note that, the depth first search may encounter even leaf nodes in that process if there is no materialization.

Algorithm 4 BMT-TAS - Efficient Algorithm for location-aware TAS using T_{BM}

```
Require: Trie $T_{BM}$, Query $Q$, an integer $k$
 1: ResultsPQ = $\phi$, stackForDFS = $\phi$, pqforBFS = $\phi$
 2: Perform lookup in Trie $T_{BM}$ to determine QMN(Q)
 3: stackForDFS.Push(QMN(Q))
 4: while true do
 5:     while (!stackForDFS.empty()) do
 6:         $nextNodeToTraverse = stackForDFS.Pop()$
 7:         if ($nextNodeToTraverse$ == LeafNode) then
 8:             resultsPQ = LeafNodeSearch($nextNodeToTraverse$);
 9:         else
10:             for all $childNode \in nextNodeToTraverse.children$
                do
11:                 if ($childNode.IsMaterialized$) then
12:                     $bound = GetBound(childNode.RCover, Q.loc)$
13:                     pqForBFS.$Enqueue(ChildNode, bound)$
14:                 else
15:                     stackForDFS.$Push(ChildNode)$
16:     if (pqForBFS.$IsEmpty()$) then
17:         break
18:     $rootOfNextSubtreeToTraverse$ = pqForBFS.$Dequeue()$
19:     if (k-th largest score in resultsPQ $\geq$
        $rootOfNextSubtreeToTraverse.Priority$) then
20:         break
21:     else
22:         $stackForDFS.Push(topItemFromPQ.Value)$
23: return $Res(Q, k)$ from ResultsPQ
```

7. EXPERIMENTAL EVALUATION

We present an experimental evaluation of the techniques proposed in the paper. The goals of our study are:
• To measure the benefit of materializing score bounds using the $\{M, R\}$ distribution technique to query performance
• To evaluate the benefit of judicious node selection and adaptive cover creation using the $\{M, R\}$ distribution technique over arbitrary node and cover selection
• To study the sensitivity of query performance of BMT-TAS algorithm to M and R
• To study the scalability of BMT-TAS to database size
• To study the sensitivity of query performance of BMT-TAS to k
• To study the sensitivity of query performance of BMT-TAS to the ranking function, specifically the weights w_d and w_s of the two components.

7.1 Experimental Setting

Implementation: We implemented two algorithms: (i) the baseline algorithm Baseline-TAS and (ii) the proposed algorithm BMT-TAS using the bound materialized trie. For Baseline-TAS, we modify the standard trie as follows. For each leaf node $l_{\mathcal{T}}$, we store two pointers:
(i) **Pointer to a *kd*-tree**: We store the spatial locations of the objects in the object list of $l_{\mathcal{T}}$ (along with the object ids) in a *kd*-tree and store a pointer to the root of that *kd*-tree in $l_{\mathcal{T}}$.
(ii) **Pointer to static score list**: We store the ids, static stores and locations of those objects in a list sorted in decreasing order of static scores. We store a pointer to that list in $l_{\mathcal{T}}$ as well.
By exploiting the getNext() interface of *kd*-tree, getNext() interface of static score list as well as random access interface to the static score list, we implemented the LeafNodeSearch subroutine using the TA algorithm as discussed in Section 3.2. This implementation of LeafNodeSearch is used by both Baseline-TAS and BMT-TAS. The remaining implementation of Baseline-TAS follows the pseudocode of Algorithm 1.

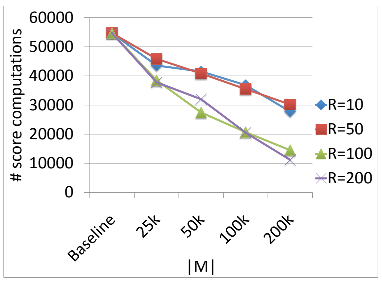


Figure 8: Num score computations w.r.t. M **and** R

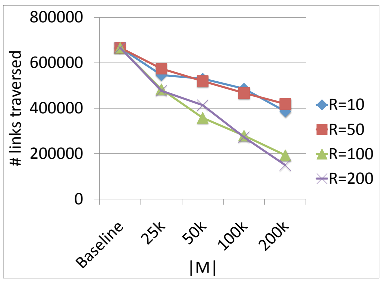


Figure 9: Num links traversed w.r.t. M **and** R

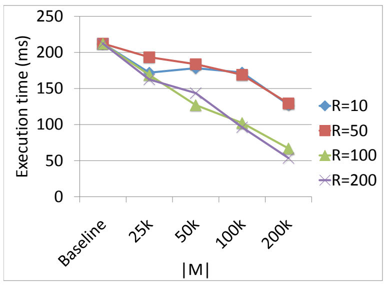


Figure 10: Execution time w.r.t. M **and** R

For BMT-TAS, we further modify the trie as follows. We first execute the node and cover selection process. Subsequently, for each trie node $n_{\mathcal{T}}$ selected, we store the regions in R-cover and the score bound for each region in $n_{\mathcal{T}}$. Since each region is a node in the quad-tree over the basic grid $\mathcal{G}$ of $n_{\mathcal{T}}$, we store a single number that encodes the quad-tree node [18]. At query time, we can use the encoding to determine the region that contains the query location efficiently.

Datasets: We report results over a real-world dataset and a synthetic dataset.

Yellow Page dataset: This dataset contains names, locations (in latitude and longtitude) and popularities of 100,000 business entities in the state of Washington, USA. We use the popularities of the businesses as their static scores. Each business name has an average of 20 characters.

Synthetic dataset: We synthetically generated spatial datasets that emulate real-life spatial data for scalability experiments. We make three observations about real-world spatial data: (i) the number of objects having the same description string follows the Zipf distribution, i.e., there are a few strings associated with many objects (e.g., "Starbucks", "Bank of America", "Mcdonald's") while most strings are associated with very few objects (ii) the popularities of objects also follow the Zipf distribution: some objects are extremely popular while most are modestly popular and (iii) locations of objects follow a clustered distribution. Based on these observations, we generate a synthetic dataset as follows. We have a large set of business names (but not their locations and popularities) from the customer database of a Fortune 500 company: we randomly select a string from that set. We then generate a number z (scaled by the desired size of the database) from a Zipf distribution and generate z objects associated with the string. We generate the locations of the objects following the clustered distribution and the popularities following the Zipf distribution. We iterate the process till we obtain the desired number of objects.

Queries: For each dataset, we generated 100 queries as follows. We randomly select 100 prefixes of object strings (of length 1, 2 and 3 characters) whose selectivities are between 1% and 10%. We associate the location of a randomly selected object from the database with each prefix. An example of a query for Yellow Page dataset is ("ma", (47.60, -122.33)). All our results are averaged over 100 such queries. We use a $k = 10$ unless otherwise specified. We use $w_d = 0.5$ and $w_s = 0.5$ unless otherwise specified.

All experiments were conducted on an Intel x64 machine with two 2.66GHz Intel Xeon processor and 8GB RAM, running Windows 2008 Server (R2 Enterprise x64 edition). In all experiments, the trie resides completely in main memory.

7.2 Experimental Results: Synthetic Dataset

Benefit of bound materialization: We first evaluate the benefit of materializing the bounds computed by the $\{M, R\}$-technique on a synthetic database of 1 million objects. We compare the query processing cost of BMT-TAS algorithm with that of Baseline-TAS. We measure the query processing cost not only by execution time but also the two main components: (i) number of score computations, and (ii) number of links traversed. Figure 8 shows the number of score computations for Baseline-TAS and BMT-TAS. The leftmost point in each chart shows the number for Baseline-TAS (labeled Baseline in all charts) while the remaining points show the number for BMT-TAS for various values of M and R. For all values of M and R, BMT-TAS performs fewer score computations compared to baseline. BMT-TAS is most beneficial when bounds are materialized in 100,000 or more nodes and the number of regions is at 100 or more. For example, for $M = 100,000$ and $R = 100$, BMT-TAS *performs 3 times fewer score computations* compared with Baseline-TAS. For $M = 200,000$ and $R = 200$, BMT-TAS outperforms Baseline-TAS *by a factor of 5.* [9] This implies that the bounds help in avoiding most score computations.

We next compare the performance of the two algorithms in terms of the second component of the cost: the number of links traversed. Figure 9 plots the results. The results are quite similar to the case of score computations: BMT-TAS performs fewer score computations compared to Baseline-TAS for all values of M and R. Again, BMT-TAS is most beneficial for $M \geq 100000$ and $R \geq 100$. For example, for $M = 100,000$ and $R = 100$, BMT-TAS *traverses 2.5 times fewer links* compared with Baseline-TAS. For $M = 200,000$ and $R = 200$, BMT-TAS outperforms Baseline-TAS *by a factor of 4.5.* This implies that the bounds help in avoiding traversing most of the links.

Finally, we compare the two algorithms in terms of the execution time. Figure 10 shows the results. The results follow the same pattern as the previous two figures. First, this validates our model of query processing cost: the execution time is a weighted combination of the above two costs. Second, BMT-TAS outperforms Baseline-TAS for all values of M and R, For $M = 100,000$ and $R = 100$, BMT-TAS is *2 times faster* than Baseline-TAS. For $M = 200,000$ and $R = 200$, BMT-TAS is *4 times faster* than Baseline-TAS. As discussed in Section 1, it is crucial to keep the trie search time well below 100 ms: BMT-TAS achieves the above goal. While Baseline-TAS takes an average of 212 ms, BMT-TAS with $M = 200,000$ and $R = 200$ takes an average of only

[9] Assuming average object size of 36 bytes (20 bytes for string, 4 bytes for id, 8 bytes for location and 4 bytes for static score), the trie along with the objects have a size of 36MB. With $M = 100,000$ and $R = 100$, the space to materialize the bounds is roughly $100000 \times 100 \times 8 = 80MB$ (assuming 8 bytes per region) Hence, $M = 100,000$ and $R = 100$ or even $M = 200,000$ and $R = 200$ is feasible in modern hardware.

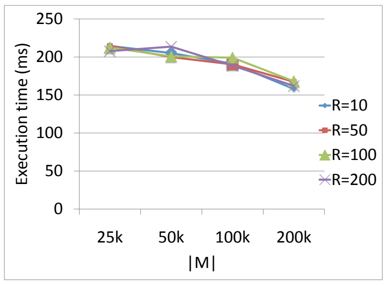


Figure 11: Execution time for Rand-Uni materialization

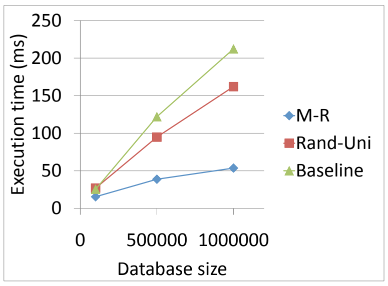


Figure 12: Scalability to database size

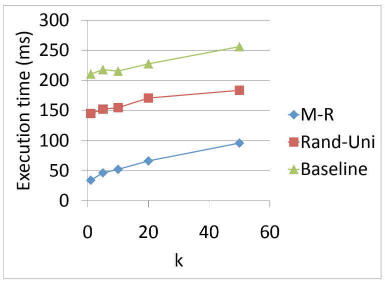


Figure 13: Execution time w.r.t. k

54ms. In summary, our techniques make location-aware TAS feasible.

Comparison with Random Node Selection: The above charts show that bound materialization saves the query processing cost. However, do we really need judicious node selection and adaptive cover creation using the $\{M, R\}$ distribution technique? Or would randomly selecting M nodes and an uniform grid of R cells in each of those nodes would have been equally beneficial (referred to as *Rand-Uni* materialization)? We conducted experiments to answer the above questions. We selected the nodes and the covers using the Rand-Uni technique and augmented the trie with it. We then executed queries using `BMT-TAS` on such a trie. Figures 11 shows the performance in terms execution time for various values of M and R. Rand-Uni materialization do have some benefit over `Baseline-TAS` but our $\{M, R\}$ distribution technique yields much more benefit for the same space overhead (i.e., $M \times R$). For example, for $M = 200,000$ and $R = 200$, the number of score computations, number of links traversed and execution time for Rand-Uni materialization are $31k$, $424k$ and 162ms respectively; for the $\{M, R\}$ technique, these numbers are $11k$, $149k$ and $53ms$. This implies that the benefit modeling and node selection based on benefit are important; similarly, the adaptive choice of regions is also important.

Scalability to database size: We evaluate the scalability of `Baseline-TAS` and `BMT-TAS` using $\{M, R\}$-based materialization and Rand-Uni materialization. We generated synthetic databases of size 100k, 500k and 1 million. For the materialization approaches, we chose M to be 20% of the database size (i.e., 20k, 100k and 200k respectively) and $R = 200$. Figure 12 the performance in terms of execution time. $\{M, R\}$ technique significantly outperforms `Baseline-TAS` and Rand-Uni for all database sizes. The gap increases with increase in database size. We observed similar behavior for the two components of the cost; we omit those charts to avoid repetition.

Sensitivity to k: We evaluate the sensitivity of the algorithms to the number k of results desired. Figure 13 plots the execution time of the 3 techniques for various values of k. For all values of k, $\{M, R\}$ technique significantly outperforms the other two techniques.

7.3 Experimental Results: Real Dataset

Benefit of bound materialization: We first evaluate the benefit of bound materialization for the real world dataset. Figure 14 shows the average execution time for `Baseline-TAS` (Baseline) and `BMT-TAS`. As in the synthetic dataset, `BMT-TAS` is most beneficial when the bounds are materialized in enough nodes. In this case, that number is 10000. For $M = 20000$ and $R = 10$, `BMT-TAS` outperforms `Baseline-TAS` *by a factor of 3*. We observe that covers consisting a few regions (e.g., 10) suffices for this dataset; further increasing R does not provide additional benefit. We investigated this behavior in depth. A highly popular business (that is a valid completion) typically has the highest overall score for queries originating from the area surrounding it, i.e., it dominates over less popular businesses in the surrounding area. Since the real dataset is spatially sparser compared with the synthetic dataset (10 times fewer objects distributed over the same geographical space), the highly popular businesses are spread farther apart and each of them dominates in even larger surrounding areas. Hence, many regions around it will have the same upper bound score, i.e., the bounds will have more spatial locality. In such cases, our adaptive R cover algorithm can cover the geographic space with a small number of regions without introducing much error.

Sensitivity to ranking function: Finally, we evaluate the sensitivity of the $\{M, R\}$ technique to the ranking function. Specifically, we study how sensitive is the query performance to the weights w_d and w_s. All previous experiments used $w_d = 0.5$ and $w_s = 0.5$. Figures 15 and 16 show the execution times for $w_d = 0.25, w_s = 0.75$ and $w_d = 0.75, w_s = 0.25$. `BMT-TAS` outperforms `Baseline-TAS` for these values of weights as well. Note that when the weight on spatial proximity increases to 0.75, the domination of highly popular businesses in surrounding areas reduces. This reduces the spatial locality of the bounds. Hence, increasing R does provide additional benefit.

8. RELATED WORK

To the best of our knowledge, this is the first work on TAS over spatial databases. Our semantics and architecture build upon prior works on keyword search on spatial databases, standard TAS, and autocompletion. However, our principal technical contributions remain in the novel formulations and solutions of the memory distribution problem.

Keyword search on spatial databases: Keyword search on spatial data has been recently studied in [10, 6]. Existing work proposes ranking metric that considers both proximity and relevance of the object with the query. Our ranking framework uses similar intuition by combining distance and static score in location-aware TAS. However, location-aware TAS is required to return top-k valid completions with every key stroke, making our problem and solution tangentially different from [10, 6].

Autocompletion and TAS: The TAS problem is related to autocompletion [8, 16, 5]. In autocompletion, as the user types in her query, the system displays the set of likely completions of the query. While autocompletion reduces the amount of typing, it still requires a large amount of clicking before the user gets to see the final results.

Our work differs from prior work on TAS as prior approaches are *not location-aware* [14, 3]. TAS on spatial data needs to be location-aware; hence, prior semantics, architecture, and algorithms are inadequate. Another example of TAS is the recently released *Google Instant (GI)*. With a query being typed, GI selects the most popular completion

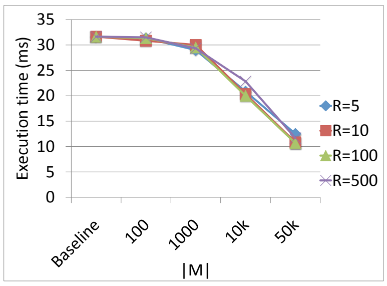


Figure 14: Execution time for real dataset, $w_d = 0.5, w_s = 0.5$

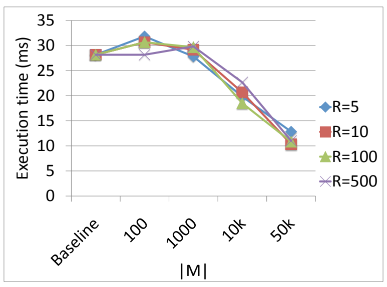


Figure 15: Execution time for real dataset, $w_d = 0.25, w_s = 0.75$

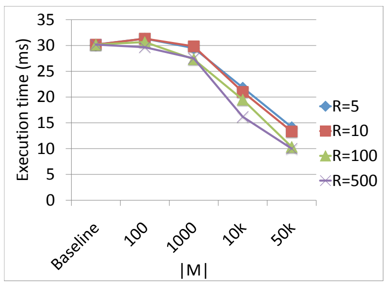


Figure 16: Execution time for real dataset, $w_d = 0.75, w_s = 0.25$

and returns the results for it. However, in the cases where the most popular completion is not the intended query, the results will be undesirable. We, on the other hand, return the *"best" set of results among all the likely completions of the query* and use location as a strong signal to select the "best" results. Due to the diversity of the results, our results are much more likely to include the desired result even with just a few letters typed.

Spatial index structures and other data structures: We use spatial data structures as components of our location aware TAS system [4, 11]. In the integrated architecture proposed in this paper, we index the spatial locations of the objects in each leaf node using a main memory spatial index structure (e.g., kdtree). Subsequently, we can use the threshold algorithm to efficiently implement `LeafNodeSearch`. Spatial grids have been used for indexing spatial objects [2, 17]. In this paper, we use grids for storing the bounds instead of indexing objects. Furthermore, our main contribution is the adaptive cover computed using the spatial grid of bounds. Our R-cover selection algorithm uses a quad-tree as a basis [11]; our main contribution here is the optimal cover selection algorithm. Furthermore, `BMT-TAS` employs a best-first search [7] approach to search in T_{BM}, leveraging priority queue [7] data structure.

Optimization problems: We formalize two optimization problems to pre-compute T_{BM}. Unfortunately, both of them are computationally hard. We show the NP-Completeness of M Node Selection problem by reducing classical set cover [12] problem to it, even for a much simpler benefit function. In the absence of a reasonable approximation algorithm, applicable for our actual benefit function, we design a hill climbing based heuristic solution.

Our R Cover Creation problem aims to optimize maximum-error [13] of individual grid cells, and is also NP-hard. It directly maps to the existing work [15], that shows the hardness of rectangular partitioning in two-dimensional space. Although efficient approximation algorithm is designed for cases where the objective is to minimize the number of regions that is required to satisfy an error bound, those results do not extend to our case where the number of regions(R) is given and the task is to determine the partitioning that minimizes the error.

9. CONCLUSION

In this work, we introduce the problem of location-aware TAS on spatial databases. We show that standard TAS techniques can not be adapted effectively on spatial data. We propose an integrated architecture for location-aware TAS. Furthermore, we suggest novel techniques to augment that architecture with materialized information for efficient query processing. The challenges mainly surfaces due to the limited availability of main memory. We formalize two optimization problems in that contexts by modeling query processing cost. We demonstrate the hardness of both problems, and design efficient algorithmic solutions. Finally, we devise an optimal query processing algorithm that uses that augmented architecture for efficient query processing. We perform extensive experiments on real and synthetic data that corroborate the efficiency of our proposed solution.

10. REFERENCES

[1] *http://www.marketwire.com/press-release/Study-Shows-Double-Digit-Growth-Local-Mobile-Usage-Unlocking-Access-Younger-Wealthier-1297436.htm.*

[2] *Spatial databases with application to GIS.* Morgan Kaufmann Publishers Inc., 2002.

[3] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.

[4] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Software Eng.*, 5(4):333–340, 1979.

[5] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.

[6] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[8] J. J. Darragh and I. H. Witten. Adaptive predictive text generation and the reactive keyboard. *Interacting with Computers*, 3(1):27–50, 1991.

[9] R. Fagin and et. al. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[10] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[11] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[13] M. N. Garofalakis and A. Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *PODS*, pages 166–176, 2004.

[14] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, pages 695–706, 2009.

[15] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *ICDT*, pages 236–256, 1999.

[16] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD Conference*, pages 1156–1158, 2007.

[17] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 1984.

[18] M. Prez, X. Benavent, and R. Olanda. Efficient coding of quadtree nodes. In *Computational Science ICCS 2006.* 2006.

[19] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, 1995.