Scalable Top-K Spatial Keyword Search

Dongxiang Zhang, Kian-Lee Tan, Anthony K. H. Tung
School of Computing, National University of Singapore
13 Computing Drive, 117417, Singapore
{zhangdo,tankl,atung}@comp.nus.edu.sg

ABSTRACT

In this big data era, huge amounts of spatial documents have been generated everyday through various location based services. Top-*k* spatial keyword search is an important approach to exploring useful information from a spatial database. It retrieves *k* documents based on a ranking function that takes into account both textual relevance (similarity between the query and document keywords) and spatial relevance (distance between the query and document locations). Various hybrid indexes have been proposed in recent years which mainly combine the R-tree and the inverted index so that spatial pruning and textual pruning can be executed simultaneously. However, the rapid growth in data volume poses significant challenges to existing methods in terms of the index maintenance cost and query processing time.

In this paper, we propose a scalable integrated inverted index, named I^3, which adopts the Quadtree structure to hierarchically partition the data space into cells. The basic unit of I^3 is the `keyword cell`, which captures the spatial locality of a keyword. Moreover, we design a new storage mechanism for efficient retrieval of keyword cell and preserve additional summary information to facilitate pruning. Experiments conducted on real spatial datasets (Twitter and Wikipedia) demonstrate the superiority of I^3 over existing schemes such as IR-tree and S2I in various aspects: it incurs shorter construction time to build the index, it has lower index storage cost, it is order of magnitude faster in updates, and it is highly scalable and answers top-*k* spatial keyword queries efficiently.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Retrieval models

General Terms

Algorithm, Performance

Keywords

Top-*k* spatial keyword, Quadtree, Inverted index

1. INTRODUCTION

The great success of iPhone and Android has led to an increase in the market share of smartphones in recent years. Such smartphones are equipped with GPS devices that allow a user to conveniently capture and associate his/her current location with published content. Users can take a photo, mark its geo-location and annotate it with tags on Flickr so that the photo is associated with both spatial and textual information. They can also publish news and events on the spot using Twitter. Consequently, more and more *spatial documents*[1] have been created and become publicly accessible. For example, Flickr has hosted more than 5 billion photos, impacted by the iPhone's existence[2]. Twitter nowadays delivers almost 250 million tweets a day[3]. The scale of these enormous location-based databases and the demand for real-time response make it very critical to develop efficient query processing mechanisms.

Top-*k* spatial keyword search is an important tool in exploring useful information from a spatial database and has been well studied for years [20, 4, 10, 8, 6, 12, 14, 17]. The query consists of a spatial location, a set of keywords and a parameter *k*. The answer is *k* top-most relevant documents which are ranked based on a combination of both the spatial and textual attributes. The spatial relevance is measured by the distance between the location associated with the candidate document to the query location, and the textual relevance is computed in the same way as in traditional search engines. In this paper, we study both `AND` semantics and `OR` semantics for the spatial keyword query.

- The `AND` semantics has a textual constraint that each returned document should contain `all` the query keywords. Such semantics is widely used in many location based applications where the query keywords indicate a user's preference. For example, queries like "spicy Chinese restaurant" have clear user intention and require all the keywords to be matched. The collective spatial keyword query recently proposed in [3] is another interesting application of `AND` semantics.

- The `OR` semantics allows the query keywords to partially match a document. This is a more general case. In the above example, when a user does not have a strong preference, non-spicy Chinese restaurants can also be recommended if they are close to the user's location. Compared to `AND` semantics, more candidates will be examined in the query processing of `OR` semantics.

[1]We use the term `spatial document` to refer to a document that has a geo-location annotation.
[2]http://techcrunch.com/2011/09/02/instagram-adds-50-million-photos-in-august-now-over-200-million-total/
[3]http://techcrunch.com/2011/10/17/twitter-is-at-250-million-tweets-per-day/

Ideally, a scalable solution should satisfy three requirements: (i) The index requires low maintenance cost and should be efficient for highly frequent insert operations. (ii) The index shows effective storage utilization. (iii) The index answers queries efficiently. In state-of-the-art solutions, the most popular one is to embed the inverted index into the R-tree, named the IR-tree family [10, 6, 14]. In these indexes, a centralized R-tree is maintained to capture the global spatial information. All the documents are inserted into the R-tree using the spatial attribute. Each node in the tree is augmented with an inverted index, which refers to a pseudo-document representing all the documents whose locations fall inside the node's MBR. The IR-tree can take advantage of spatial and textual information simultaneously in the pruning stage. When examining a tree node, we can calculate its upper-bound score using the maximum query term weight and the minimum spatial distance. If the upper bound value is smaller than the current best score, the node can be pruned. However, when handling a large dataset, the IR-tree suffers from two main drawbacks. First, the centralized R-tree mechanism requires high update cost. Each node has to maintain an inverted index for all the keywords of documents associated with this node's MBR. When a node is full and split into two new nodes, all the textual information in the node has to be re-organized. Second, the processing cost to examine whether a node is relevant to the query keywords is not negligible [17]. A large number of pseudo-documents have to be accessed when the query keywords are frequent. In the worst case, all the tree nodes have to be accessed for a query, incurring very high I/O cost. Thus, it is difficult for IR-tree to scale to a huge dataset.

Recently, Rocha-Junior et. al. proposed the S2I index and showed that it offers better query processing performance [17]. S2I uses textual-first partition and splits the database into inverted lists. If a keyword is frequent, an aggregated R-tree [16] is built to further improve spatial pruning. Otherwise, the infrequent keywords are stored in a flat file. S2I is scalable in terms of the number of keywords in the database because given a set of query keywords, only the related inverted lists are accessed. Compared to IR-tree, it greatly reduces the amount of I/O access in the worst case. However, it is costly to perform partial aggregation across different R-trees, especially when the keywords are frequent. It results in a large number of random accesses on tree nodes. In addition, S2I maintains inverted lists and R-trees at the same time. When a keyword becomes frequent, its related documents were stored in the flat file and now these documents have to be taken out and inserted into a new R-tree. It takes considerable overhead for the data transfer between the flat file and the R-tree index.

In this paper, we use the Quadtree [9] to decompose the data space into a hierarchy of cells and propose a scalable integrated inverted index, named I^3, to manipulate spatio-textual information. Quadtree is selected because it is recommended for update-intensive applications [11]. Our index stores `keyword cell` as the basic unit, which captures spatial locality for a keyword. A `keyword cell`, denoted by $\langle w_i, C_j \rangle$, refers to a list of documents containing keyword w_i and having their associated locations in cell C_j. Moreover, I^3 stores summary information of keyword cell for effective pruning. The summary information includes a signature file [7] which aggregates document id in the keyword cell and the upper bound score of keyword relevance. Based on the summary information, we propose different pruning strategies for `AND` and `OR` semantics. To sum up, we propose a scalable index structure I^3 for top-*k* spatial keyword query and the index has the following advantages:

- I^3 greatly reduces the index construction time.
- I^3 takes much less update cost than existing methods and is more suitable for big data scenarios.
- I^3 is effective in storage utilization.
- I^3 significantly outperforms state-of-the-art schemes when answering top-*k* spatial keyword queries in terms of `AND` semantics and `OR` semantics.

The rest of the paper is organized as follows. In Section 2, we review related work in the area of top-*k* spatial keyword search. In Section 3, we address the problem of spatial keyword search based on `AND` semantics and `OR` semantics. The index overview of I^3 and detailed explanation of data operations are presented in Section 4. The query processing strategy is proposed in Sections 5. Results of an extensive experimental study on real datasets are reported in Section 6. Section 7 concludes the paper.

2. RELATED WORK

Spatial keyword search has been well studied for years due to its importance to commercial search engines. Various types of spatial keyword queries have been proposed. These related works can be categorized from two dimensions. The first dimension indicates whether the query contains a region as a spatial constraint and the second dimension specifies whether `AND` or `OR` semantics is used.

First, we introduce the works with spatial constraint which requires the returned documents' associated location to be intersecting with or contained in a query region [4, 10, 20, 12, 5, 14]. Among these works, [4] and [10] adopt `AND` semantics while [20], [12], [5] and [14] use `OR` semantics. This type of query is especially useful in web search engine. The query keywords contain a gazetteer term which can be represented by an approximate region. The result documents have spatial annotations that overlap with the query region, and are ranked based on certain criteria. Early works [20, 4] use two separate indexes, an R-tree to capture spatial information and inverted index for textual pruning. The query processing contains two stages, either spatial-first or textual-first, depending on the spatial and textual selectivity. Such a two-phrase pruning approach is not efficient because many false positive candidates, which are only spatially or textually relevant, still need to be accessed. To improve the efficiency, various hybrid indexes have been proposed. The main idea is to extend R-tree to embed textual information in tree nodes. The most common solution is to augment a tree node with inverted index for the spatial documents within that MBR [10, 14].

The remaining works [8, 6, 17] do not have the query region overlapping constraint. The candidates can be located anywhere in the space and ranked by spatial relevance and textual relevance. In [8], Felipe et al. considered `AND` semantics and ranked the results by their distance to the query location. They proposed IR^2, a hybrid index of the R-tree and signature file, for query processing. Cong et al. [14] proposed a more general top-*k* spatial keyword query. It uses `OR` semantics and retrieves spatial documents ranked by both spatial relevance and textual relevance. The spatial relevance is measured by the distance from the document's associated location to the query location and the textual relevance is defined using tf-idf. To support top-k spatial keyword queries, IR-tree was proposed to integrate R-tree with inverted index. It augments the nodes of an R-tree with a pseudo-document vector to store the textual information of that node. Spatial relevance is calculated from the MBR and an upper bound of textual relevance is derived from the pseudo-document for pruning. As a variant of IR-tree, DIR-tree takes into account both spatial and textual proximity when inserting an object. It shows better performance when the spatial keywords are correlated locally. However, the IR-tree family takes non-negligible processing cost when examining whether a node is relevant to query keywords. A large number of pseudo-documents have to be accessed when the query keywords are frequent, incurring high I/O cost.

Recently, S2I index [17] was proposed for more efficient spatial keyword search. It partitions the spatial database first by the textual attribute. If a keyword is infrequent, all the elements in the inverted list are stored sequentially for efficient retrieval to save I/O cost. Otherwise, an aggregated R-tree [16] is built for spatial pruning. S2I is scalable to the number of keywords in the database because given a set of query keywords, only the related inverted lists will be accessed. However, it is difficult to do spatial aggregation across different R-trees.

Our proposed I^3 index adopts textual partition first just like the S2I index. We discard R-tree and use Quadtree to split the space into a hierarchy of cells. The basic unit in our index is named `keyword cell` which captures spatial locality for a keyword. We design a new storage mechanism to store keyword cell. We also augment dense keyword cell with signature file and other summary information for effective pruning.

Besides the traditional spatial keyword search problem, variants of the topic have been proposed. One is to allow the query keywords to appear in multiple documents, which is referred as *m*CK query[18, 19] or collective spatial keyword [3]. Another interesting extension is to add the user's driving or walking direction as a constraint [13].

3. PROBLEM STATEMENT

In our data model, a spatial document $\mathscr{D}$ is associated with both spatial and textual attributes. Since we assume the documents are mainly generated from smart phones, we can use a two dimensional point in the form of latitude and longitude to represent spatial information. The textual attribute is represented by a list of keywords, each with a term weight which can be customized in different applications. We use the classic tf-idf measure [2, 15] to evaluate the term weight or textual relevance score s_i of a keyword w_i with respect to a document $\mathscr{D}$:

$$\mathscr{D} = <\mathscr{D}.id, \quad \mathscr{D}.lat, \quad \mathscr{D}.lng, \quad \mathscr{D}.terms>$$

$$\mathscr{D}.terms = \{<w_i, \quad s_i>\}$$

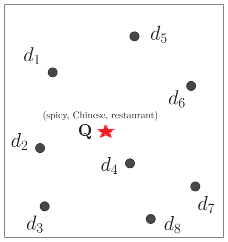
Figure 1 illustrates an example of a spatial database with 8 documents, each associated with a location and a set of keywords. The term weight of each keyword in the document in also provided. For the problem definition, we adopt the same top-*k* spatial keyword query in [6] except that we consider both `AND` semantics and `OR` semantics. In the definition, a query Q is a spatial point with several keywords. For example, in Figure 1, the query contains three keywords and its location is marked as a five-pointed star. Since we consider top-*k* query, Q also contains a parameter k to determine the number of matching documents to return.

$$Q = <Q.lat, \quad Q.lng, \quad Q.terms, \quad Q.k>$$

When `AND` semantics is used, $\mathscr{D}$ is a candidate only if it contains all the query keywords, i.e.,

$$\forall w \in Q.terms, \quad w \in \{\mathscr{D}.terms.w_i\}$$

When `OR` semantics is used, $\mathscr{D}$ is a candidate as long as it contains at least one query keyword to indicate the textual relevance, i.e.,



Spatial documents and a query Q

d_1	(Chinese 0.6), (restaurant,0.4)
d_2	(Korean 0.7), (restaurant 0.3)
d_3	(spicy 0.2), (Chinese 0.2), (restaurant 0.5)
d_4	(spicy 0.7), (restaurant 0.7)
d_5	(spicy 0.8), (Korean 0.5), (restaurant 0.6)
d_6	(spicy 0.4), (restaurant 0.5)
d_7	(Chinese 0.1), (restaurant 0.3)
d_8	(restaurant 0.2)

Document keywords

Figure 1: A simple example of a spatial database

$$\exists w \in Q.terms, \quad w \in \{\mathscr{D}.terms.w_i\}$$

The ranking function needs to take into account both spatial and textual proximity. We define the score of a document $\mathscr{D}$ as the linear combination of its spatial score and textual score, using a parameter α.

$$\mathscr{D}.score = \alpha \cdot \phi_s + (1-\alpha) \cdot \phi_t$$

Here, ϕ_s refers to spatial proximity, measured inversely proportional to the distance from the query location to the candidate's location. ϕ_t is tf-idf based textual relevance score, measured directly proportional to the query keyword frequency in the candidate document. Finally, the result set contains k tuples with the highest scores.

Note that in our data model we consider a spatial database in which each document is associated with a two dimensional point. However, our proposed index is general enough to handle arbitrary geometry shape.

4. INTEGRATED INVERTED INDEX

An efficient index for spatial keyword search is required to support both spatial pruning and textual pruning simultaneously. Existing solutions prefer the combination of R-tree and inverted list. However, those hybrid indexes are not scalable and require high maintenance cost. In this section, we introduce our new index I^3 to manipulate spatial textual data.

4.1 Textual Partition

Like S2I, I^3 adopts textual-first partition. In other words, given a spatial document $\mathscr{D}$, we split it into a set of small tuples. Each tuple is associated with only one keyword w. The location information is inherited from $\mathscr{D}$. We also preserve the document id and term weight s in $\mathscr{D}$. Hence, a tuple $\mathscr{T}$ has the following fields:

$$\mathscr{T} = <\mathscr{T}.id, \quad \mathscr{T}.w, \quad \mathscr{D}.id, \quad \mathscr{D}.lat, \quad \mathscr{D}.lng, \quad \mathscr{T}.s>$$

Tuples with the same $\mathscr{T}.w$ are grouped together. Such a textual-first partition provides a better performance bound than IR-tree in the worst case because only the related query keywords will be accessed, i.e., $\mathscr{T}$ is candidate only if $\mathscr{T}.w \in Q.terms$.

4.2 Keyword Cell

After the textual partition, commercial search engines build an inverted list for each partition and the tuples are inserted in the descending order of their term weight. Without spatial information embedded, spatial pruning cannot be applied via the inverted index. To improve performance, S2I distinguishes between frequent and infrequent keywords. Given a threshold T, if a keyword's frequency exceeds T, it is considered frequent and all the tuples for the keyword will be inserted into a separate R-tree. Otherwise, the keyword is infrequent and its tuples are stored in a flat file sequentially. Such a design has the following drawbacks:

- The threshold T is difficult to tune. If it is set too small, there will be a large number of small R-tree index files generated. The restriction of open file limit in the operating system may lead to degradation in the index performance. If T is set too large, many keywords become infrequent and lack spatial pruning in the query processing.
- When the query keywords are frequent, it requires considerable random access cost to aggregate the final score for a document whose keywords appear across different R-trees. It lacks global summary information among the R-trees for pruning and the overlap of MBRs in different trees is also not good for score aggregation.

In this paper, we discard R-tree and use Quadtree [9] for space decomposition. We maintain a Quadtree for each keyword. There are two reasons for this selection:

1. Quadtree is easier to maintain and is recommended in update intensive applications [11].
2. Quadtree provides a uniform space decomposition mechanism for all the keywords. It is suitable for join queries among different keywords in query processing.

A Quadtree has a maximum capacity for the cells. When the maximum capacity is reached, the cell is split into four smaller cells and the data points in the parent cell are inserted into child cells. We treat the tuples that are associated with the same keyword and located in the same cell as a basic unit, denoted by `keyword cell` $\langle w_i, C_j \rangle$. In this way, the inverted list for w_i becomes a list of keyword cells. Since each keyword cell has a maximum capacity, we set the capacity to be the same as the page size. If the page size is P and a tuple takes B bytes for storage, the capacity is set $\lfloor P/B \rfloor$. Thereafter, all the tuples $\mathscr{T}$ with the same keyword and located in the same cell will be stored in the same disk page. These tuples can be retrieved with one disk I/O. If the number of tuples in $\langle w_i, C_j \rangle$ exceeds $\lfloor P/B \rfloor$, we say $\langle w_i, C_j \rangle$ is **dense** or keyword w_i is **dense** in cell C_j. The keyword cell will be split into smaller ones that are not dense.

Compared to S2I, keyword cell provides a uniform storage and retrieval interface for all the keywords as they follow the same space decomposition mechanism. I^3 does not need the threshold T to indicate whether a keyword is frequent or infrequent.

Compared to inverted index, I^3 further partitions the tuples in the list based on their cell id. With the embedded spatial information, spatial pruning can be utilized to facilitate query processing. In addition, we do not need to guarantee that all the elements in an inverted list are stored contiguously because guaranteeing the contiguity requires intensive update cost to manage variable-length fragments[21]. Instead, we allow the keyword cells in the same list to be stored in any order and in different pages to effectively save maintenance cost.

Figure 2 shows two inverted lists for keywords "spicy" and "restaurant" derived from the example spatial database in Figure 1. Each inverted list consists of a set of keyword cells based on the space decomposition. In this example, we set $\lfloor P/B \rfloor$ to be 2 for simplicity, meaning each page contains at most 2 tuples. A keyword cell $\langle w_i, C_j \rangle$ is stored in the inverted list of w_i only if the keyword w_i is neither empty nor dense in cell C_j. If $\langle w_i, C_j \rangle$ is dense, it contains more than 2 tuples and will be split into smaller keyword cells. For example, "restaurant" is contained in three documents $\{d_4, d_8, d_7\}$ in cell C_4. Thus, the keyword cell $\langle restaurant, C_4 \rangle$ is dense and split into three smaller keyword cells: $\langle restaurant, C_{4\_1} \rangle$, $\langle restaurant, C_{4\_2} \rangle$ and $\langle restaurant, C_{4\_4} \rangle$.

Table 1: Notation table

$\mathscr{D}$	A spatial document
$\mathscr{T}$	A spatial tuple with only one keyword
α	The weight of spatial relevance in ranking function
$\mathbb{C}$	The whole space region
w_i	A keyword
C_j	A cell in Quadtree
P	Page size
B	Tuple size
$\langle w_i, C_j \rangle$	Keyword cell of keyword w_i in cell C_j
S_i	Summary node
$\mathscr{E}$	Summary information of dense keyword cell
H	Hash function for the signature file
$\mathbb{P}$	A disk page
η	Signature length
δ	Score of the k-th result

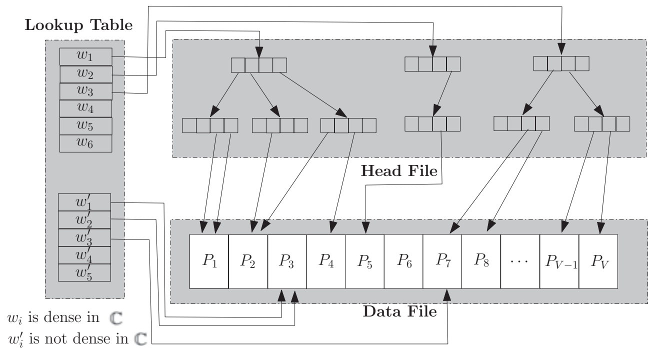


Figure 3: Index structure of I^3

4.3 Index Overview

An overview of I^3 is shown in Figure 3. The index consists of three main components: a lookup table serving as the portal, a head file containing summary information of **dense** keyword cells and a data file storing the tuples of keyword cell in all the inverted lists. To facilitate understanding of the index, a notation table is provided in Table 1.

4.3.1 Lookup Table

The lookup table is essentially a map whose key is a keyword w_i and the value contains a pair of fields. Suppose that the whole space region is denoted by $\mathbb{C}$, which is the root cell in Quadtree. The first field is a boolean flag indicating whether w_i is dense in $\mathbb{C}$. The other field is a file offset for the head file or data file. If keyword cell $\langle w_i, \mathbb{C} \rangle$ is not dense, it means w_i appears in fewer than $\lfloor P/B \rfloor$ tuples. These tuples can be stored in one disk page and fetched with one I/O. In this case, the entry in the map will point to the start position of the disk page in the data file. Otherwise, a summary node will be created in the head file and the offset field in the map is used to locate this summary node in the head file.

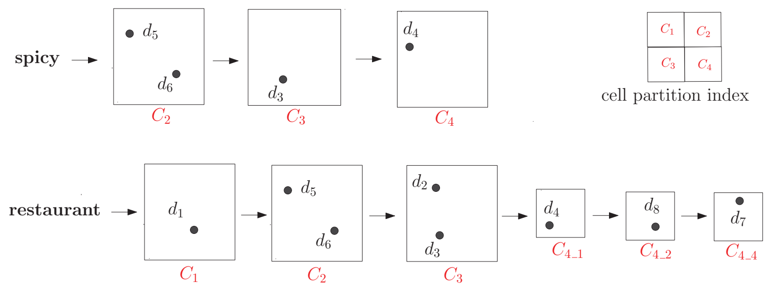


Figure 2: keyword cell examples for "spicy" and "restaurant"

4.3.2 *Head File*

A head file contains summary information of dense keyword cells for effective pruning. Given a keyword cell $\langle w_i, C_j \rangle$, the summary information $\mathscr{E}$ include a signature file *sig* and an upper bound textual score *max_s*:

$$\mathscr{E} = < \mathscr{E}.sig, \quad \mathscr{E}.max\_s >$$

The signature file is a bitmap of length η and associated with a hash function H based on the document id. When a tuple $\mathscr{T}$ is inserted, it is mapped to the $H(\mathscr{D}.id)$-th bit and that bit is set to 1. The signature file is used to aggregate all the documents containing keyword w_i and located in cell C_j. When AND semantics is used, a result is required to contain all the query keywords. We can intersect the signatures of different keywords in the same cell. If there is no intersection, it means we cannot find a spatial document associated with all the keywords. We can prune the cell without examining the objects within the cell. The field *max_s* is the upper bound textual relevance of keyword w_i. Since we can calculate the spatial relevance using the minimum distance from the query location to cell C_j, we can get a final upper bound score of keyword w_i for aggregation. If the aggregation score is smaller than the k-score in top-k results, we can prune the cell as well.

We organize the summary information similar to R-tree. In R-tree implementation, each tree node has an MBR for itself as well as a list of child MBR. The MBR is summary information and used as an approximate representation for the data points or shapes within that node. In our implementation, we create a summary node S_i for each dense keyword cell. The summary node contains its own summary information $\mathscr{E}$. Since this is a dense keyword cell, we uniformly decompose the cell into four child cells. The summary information of these four child keyword cells are also stored in S_i. Besides the summary information, S_i contains a list of child pointers. If a child keyword cell is still dense, we create another summary node for it in the head file and the pointer points to the newly created node. Otherwise, the pointer points to a disk page storing the documents in that child keyword cell. This implementation is illustrated in Figure 3. Note that in this figure, some child pointers have no outgoing arrows. This is because there are no keywords appearing in that cell.

4.3.3 *Data File*

Our data file contains a sequence of fixed-size pages. Each page is split into a fixed number of slots, one slot for one spatial tuple $\mathscr{T}$. As mentioned, the number of slots in a page is $\lfloor P/B \rfloor$ if a tuple takes B bytes. We discard the requirements of ranking order and page contiguity in inverted index so that different keyword cells in the same list can be stored in different pages and updated concurrently. We only require that all the spatial tuples from the same keyword cell are stored in the same page so that they can be accessed with one page I/O. To improve storage utilization, we allow different keyword cells to be stored at the same page. However, when a page is loaded into memory, it may contain objects from different keyword cells. To identify objects from the same keyword cell in a page, we attach a new field, named source id, to each tuple. Each keyword cell has a unique source id so that we can scan the page and fetch the valid tuples.

In short, our data file provides a flexible and uniform access interface for different keywords in different cells. All the keyword cells from different inverted lists are integrated and stored in one data file. Compared to S2I, our index is more elegant and demonstrates higher storage utilization. In S2I, it needs to set a threshold to determine whether a keyword is frequent or not. Infrequent keywords are stored in one flat file which contains their inverted lists while frequent keywords are stored in different R-trees. In I^3, we use keyword cell to provide a uniform storage and access strategy for frequent and infrequent keywords. Hence, I^3 can avoid the data movement between inverted file and R-trees when the status of a keyword turns from infrequent to frequent or vice versa.

4.4 Data Operation

Now we introduce how to maintain the index. More specifically, we explain three most basic data operations on I^3, including data insertion, deletion and update.

4.4.1 *Data Insertion*

In Algorithm 1, we show the sketch of inserting a tuple $\mathscr{T}$ into I^3. We first check whether $\mathscr{T}.w$ exists in the lookup table. If this is a new keyword, we select any page $\mathbb{P}$ in the data file with an empty slot and insert $\mathscr{T}$ into $\mathbb{P}$. The lookup table is also updated with a new entry for this non-dense keyword (lines 1 - 4). If the keyword appears in the lookup table, we check whether $\mathscr{T}.w$ is dense in the whole region $\mathbb{C}$. If it is not dense, we know that its entry in the lookup table points to the disk page $\mathbb{P}$ where the related documents are stored. We call function **insertNonDenseKwd** to insert $\mathscr{T}$ into $\mathbb{P}$ (lines 6 - 8). Otherwise, the entry in the lookup table points to a summary node in the head file. We start from this root cell $\mathbb{C}$ and recursively check whether $\mathscr{T}.w$ is dense in the child cell containing $\mathscr{T}$. There would be only one child cell containing tuple $\mathscr{T}$. In this procedure, all summary nodes that are accessed are updated to include information of tuple $\mathscr{T}$. The process stops at a non-dense cell C_u (lines 10 - 15). We can get the disk page for $\langle \mathscr{T}.w, C_u \rangle$ in the child pointers of S_i and call function **insertDenseKwd** to insert the tuple (line 16).

Algorithm 2 shows how to insert a tuple $\mathscr{T}$ with a non-dense keyword into a page $\mathbb{P}$. If $\mathbb{P}$ is not full, we can insert $\mathscr{T}$ into any empty slot and the procedure is finished (lines 1 - 2). Otherwise, the page is full and we need to scan all the tuples in $\mathbb{P}$ to see if they are associated with the same source id. If all of them are from the same keyword cell, the number of tuples containing $\mathscr{T}.w$ exceeds the

Algorithm 1 **Data Insertion**

```
1.  if $\mathscr{T}.w$ is a new keyword then
2.      select any page $\mathbb{P}$ with an empty slot
3.      insert $\mathscr{T}$ into $\mathbb{P}$
4.      insert $\mathscr{T}.w$ into lookup table
5.  else
6.      if $\mathscr{T}.w$ is not dense in cell $\mathbb{C}$ then
7.          find the disk page $\mathbb{P}$ containing tuples in $\langle \mathscr{T}.w, \mathbb{C} \rangle$
8.          insertNonDenseKwd($\mathscr{T}$, $\mathbb{P}$)
9.      else
10.         $C \leftarrow \mathbb{C}$
11.         while $\mathscr{T}.w$ is dense in $C$ do
12.             update the summary node $S_i$ for $\langle \mathscr{T}.w, C \rangle$
13.             find the child cell $C_u$ where $\mathscr{T}$ is located
14.             $C \leftarrow C_u$
15.         find the disk page $\mathbb{P}$ containing tuples in $\langle \mathscr{T}.w, C_u \rangle$
16.         insertDenseKwd($\mathscr{T}$, $S_i$, $C_u$, $\mathbb{P}$)
```

maximum capacity and makes it a dense keyword. In other words, $\mathscr{T}.w$ is now dense in the root cell $\mathbb{C}$. We split the dense keyword cell and allocate a summary node S' in the head file to store the summary information. All the tuples in $\mathbb{P}$ are scanned to get the signature file and upper bound textual relevance. Meanwhile, the entry of $\mathscr{T}.w$ in the lookup table is updated to be dense and points to the summary node S' in the head file. Finally, we find the child cell C_u containing $\mathscr{T}$ and call **insertDenseKwd** to insert the tuple into C_u (lines 4 - 9). If page $\mathbb{P}$ is full and the tuples in $\mathbb{P}$ are from multiple keyword cells, we scan the page and group the tuples by source id. Let O denote the set of tuples from $\langle \mathscr{T}.w, \mathbb{C} \rangle$. We find a page $\mathbb{P}'$ with at least $|O|+1$ empty slots and move the tuples in O together with $\mathscr{T}$ into $\mathbb{P}'$. The entry for $\mathscr{T}.w$ in the lookup table is also updated as we have changed its disk page in the data file (lines 11 - 15).

Algorithm 2 **insertNonDenseKwd**($\mathscr{T}$, $\mathbb{P}$)

```
1.  if $\mathbb{P}$ is not full then
2.      insert $\mathscr{T}$ into $\mathbb{P}$
3.  else
4.      if all the tuples in $\mathbb{P}$ are from the same source then
5.          allocate a new summary node $S'$ in the head file
6.          scan tuples in $\mathbb{P}$ and update $S'$
7.          update lookup table for $\mathscr{T}.w$
8.          find the child cell $C_u$ where $\mathscr{T}$ is located
9.          insertDenseKwd($\mathscr{T}$, $S'$, $\mathbb{C}_u$, $\mathbb{P}$)
10.     else
11.         fetch all the tuples $O$ from keyword cell $\langle \mathscr{T}.w, \mathbb{C} \rangle$
12.         find a page $\mathbb{P}'$ has at least $|O|+1$ empty slots
13.         move $O$ from $\mathbb{P}$ to $\mathbb{P}'$
14.         insert $\mathscr{T}$ into $\mathbb{P}'$
15.         update the entry of $\mathscr{T}.w$ in the lookup table
```

Now we introduce how **insertDenseKwd** works, as shown in Algorithm 3. The algorithm works similarly to Algorithm 2 except that it needs to update the summary node S_i. First, we update the *signature* and *weight* field in $\mathscr{E}$ for S_i and $\mathscr{E}_u$ for the child cell to incorporate the new tuple (line 1). After that, if $\mathbb{P}$ has an empty slot, we simply insert $\mathscr{T}$ into $\mathbb{P}$ (lines 2 - 3). Otherwise, we scan the page and check whether all the tuples are associated with the same source id. If all the tuples are from the same keyword cell, this child keyword cell $\langle \mathscr{T}.w, C_u \rangle$ will become dense. A new summary node S' is allocated in the head file. We update the u-th child pointer and make it point to the new summary node S'. All the tuples in $\mathbb{P}$ are scanned to update summary information in S'. Since we know $\mathscr{T}.w$ is dense in C_u, we find the child cell C'_u in C_u which contains $\mathscr{T}$ and call **insertDenseKwd** again to insert $\mathscr{T}$ into C'_u (lines 5 - 10). If the tuples in the full page $\mathbb{P}$ are from different sources, we fetch all the tuples in $\mathbb{P}$ which have the same source id with $\mathscr{T}$ and find a page $\mathbb{P}'$ with empty slots to host all these tuples. We move these tuples to $\mathbb{P}'$ and update the u-th child pointer to reflect the modification of disk page (lines 12 - 16).

Algorithm 3 **insertDenseKwd**($\mathscr{T}$, S_i, C_u, $\mathbb{P}$)

```
1.  update the summary information for $S_i$ and $u$-th child keyword
    cell to incorporate tuple $\mathscr{T}$
2.  if $\mathbb{P}$ is not full then
3.      insert $\mathscr{T}$ into $\mathbb{P}$
4.  else
5.      if all the tuples in $\mathbb{P}$ are from the same source then
6.          allocate a new summary node $S'$ in the head file
7.          update the $u$-th child pointer in $S_i$ to point to $S'$
8.          scan tuples in $\mathbb{P}$ and update $S'$
9.          find the child cell $C'_u$ of $C_u$ where $\mathscr{T}$ is located
10.         insertDenseKwd($\mathscr{T}$, $S'$, $C'_u$, $\mathbb{P}$)
11.     else
12.         fetch all the tuples $O$ from keyword cell $\langle \mathscr{T}.w, \mathbb{C} \rangle$
13.         find a page $\mathbb{P}'$ has at least $|O|+1$ empty slots
14.         move $O$ from $\mathbb{P}$ to $\mathbb{P}'$
15.         insert $\mathscr{T}$ into $\mathbb{P}'$
16.         update the $u$-th child pointer in $S_i$ to point to $\mathbb{P}'$
```

4.5 Data Deletion and Update

The delete operation is simpler than the insert operation. If $\mathscr{T}.w$ is not dense in the root cell $\mathbb{C}$, we follow the lookup table to find the disk page. Then, we scan the items in the page, find the tuple and delete it. If this is the last tuple associated with the keyword, we also remove the entry in the lookup table. If the page becomes empty after deletion, we do not delete it because this page can be easily reused by other insertion or split of keyword cells. If the tuple is associated with a dense keyword, we also need to scan the tuples with the same source id in the page to update the *signature* and *weight* field in its summary node. The update of summary information is then propagated upwards the summary nodes until we reach the root of Quadtree. An update operation is treated as a deletion followed by an insertion because its location information or keyword information could be changed and the tuple belongs to another keyword cell.

5. QUERY PROCESSING

In this section, we introduce our search algorithm based on I^3. We first present the overview of the query processing algorithm. Then, we explain different pruning techniques based on `AND` semantics and `OR` semantics.

5.1 Algorithm Overview

Since all the keywords follow the same space decomposition mechanism, our query processing algorithm starts from the root cell $\mathbb{C}$ and adopts a top-down search strategy to access child cells. Each cell is a candidate search space and its upper bound score can be calculated by summing the spatial relevance score and textual relevance score. The spatial relevance score is measured by the minimum distance from the cell to the query point while the textual relevance is the aggregation score of different keywords. Let

δ denote the k-th score of the current top-k results; if the upper bound score of a cell is smaller than δ, the cell can be pruned. For the candidate cells, we access them in decreasing order of their upper bound scores. This is similar to a best-first k-nearest neighbour search in R-tree. The algorithm terminates if the upper bound scores of all the remaining cells are smaller than δ.

Algorithm 4 shows the sketch of the search algorithm for top-k spatial keyword queries. For each candidate cell, we maintain four fields used for pruning.

$$\mathscr{C} =< \mathscr{C}.C, \quad \mathscr{C}.denseKwds, \quad \mathscr{C}.docs, \quad \mathscr{C}.upperScore >$$

$\mathscr{C}.C$ represents cell C which is the current search region. The query keywords are divided into two categories. If a query keyword w is dense in C, we put the keyword in $\mathscr{C}.denseKwds$. Otherwise, w is not dense and we can directly access its related tuples by loading the disk page. These candidate documents are inserted into $\mathscr{C}.docs$. Finally, $\mathscr{C}.upperScore$ means the upper bound relevance score in the current cell. If $\mathscr{C}.upperScore \leq \delta$, the cell can be pruned.

In Algorithm 4, a priority queue PQ is maintained, which contains candidate cells ordered by their upper bound score. Initially a candidate for the root cell $\mathbb{C}$ is pushed into PQ (line 1). In the following query processing stage, we pop a candidate $\mathscr{C}$ with the maximum upper bound score in each iteration (line 3). If $\mathscr{C}.upperScore \leq \delta$, we terminate the algorithm as all the remaining candidates can be pruned (lines 4 - 5). Otherwise, we check if $\mathscr{C}.denseKwds$ is empty. If all the keywords in the current cell are not dense, we have retrieved all the related tuples in disk pages and store them in $\mathscr{C}'.docs$. We can calculate the final relevance score of these documents and update δ (lines 5 - 10). If there are still query keywords that are dense in $\mathscr{C}.C$, we need to zoom into the child cells and create a new candidate $\mathscr{C}'$ for each child cell. $\mathscr{C}'.C$ is set to the child cell C_i (line 13). For each dense keyword in $\mathscr{C}'.denseKwds$, if it is no longer dense in the child cell C_i, we remove it and fetch its related tuples in the disk page. For each tuple $\mathscr{T}$, if its document $\mathscr{D}$ exists in $\mathscr{C}.docs$, it means $\mathscr{D}$ contains another query keyword which is also not dense in cell C_i. We update $\mathscr{D}$'s textual relevance score to include $\mathscr{T}.s$. Otherwise, we insert a new document into $\mathscr{C}.docs$ (lines 14 - 20). After that, we check whether we can prune the new candidate $\mathscr{C}'$. If not, we update its upper bound score and push it into the priority queue (lines 21 - 24).

For query processing, there is a need to distinguish between the AND semantics and OR semantics. There are two key differences: how to check whether a cell can be pruned and how the upper bound score for a candidate cell is computed. In the following subsections, we introduce how the pruning works in the two types of semantics.

5.2 Query Processing for AND Semantics

First, we introduce how to prune a candidate $\mathscr{C}$ when the query model follows the AND semantics in Algorithm 5. Since a result is required to contain all the query keywords, $\mathscr{C}$ can be pruned if there is no document in the cell $\mathscr{C}.C$ containing all the query keywords. We check whether there is any intersection among the signature files of dense keywords in the current cell. If no intersection is found, the cell can be pruned (lines 1 - 6). Otherwise, we store the intersection in a variable sig and continue to check if there is intersection between the document id set $\mathscr{C}.docs$ and sig. Again, if there is no intersection, we can prune the candidate search space (lines 7 - 12).

Next, we present how to update the upper bound score for a candidate $\mathscr{C}$ for the AND semantics. The textual relevance score of a document $\mathscr{D}$ is the aggregation of relevance score from all the query keywords. If $\mathscr{C}.denseKwds$ contains all the query keywords, the aggregation score is the sum of $\mathscr{E}.s$ stored in the summary node.

Algorithm 4 Query Processing

```
1.  initialize a root candidate and push it into a priority queue $PQ$
2.  **while** $PQ$ is not empty **do**
3.    pop the first candidate $\mathscr{C}$
4.    **if** $\mathscr{C}.upperScore \leq \delta$ **then**
5.      **break**
6.    **if** $\mathscr{C}.denseKwds$ is empty **then**
7.      **for** $doc \in \mathscr{C}.docs$ **do**
8.        calculate the relevance score $s$ for $doc$
9.        **if** $s > \delta$ **then**
10.         $\delta \leftarrow s$
11.   **else**
12.     **for** child cell $C_i$ in $\mathscr{C}.C$ **do**
13.       create a new candidate $\mathscr{C}'$
14.       **for** keyword $w$ in $\mathscr{C}.denseKwds$ **do**
15.         **if** $w$ is dense in $C_i$ **then**
16.           insert $w$ into $\mathscr{C}'.denseKwds$
17.         **else**
18.           retrieve tuples $\{\mathscr{T}\}$ in $\langle w, C_i \rangle$
19.           **for** $\mathscr{T} \in \{\mathscr{T}\}$ **do**
20.             update $\mathscr{C}'.docs$
21.       **if** prune($\mathscr{C}'$) = TRUE **then**
22.         **continue**
23.       updateUpperScore($\mathscr{C}$)
24.       $PQ$.add($\mathscr{C}'$)
```

If $\mathscr{C}.denseKwds$ only contains part of the query keywords, the aggregation score is contributed from two sources: *score.dense* and *score.non_dense*. The contribution from dense keywords is calculated in the same way. The *score.non_dense* is the maximum score from $\mathscr{C}.docs$. Finally, by adding the spatial relevance, we can get $\mathscr{C}.upperScore$. The algorithm is shown in Algorithm 6.

As an illustrated example, suppose we are examining cell C_4 in Figure 2 for two query keywords "spicy" and "restaurant". We know that "spice" is not dense in C_4 while "restaurant" is dense in this cell. Thus, $\mathscr{C}.denseKwds = \{restaurant\}$ and $\mathscr{C}.docs = \{d_4\}$. In this case, *score.dense* is the maximum score of "restaurant" in C_4, which is 0.7. *score.non_dense* is the relevance score of "spicy" in d_4. So the upper bound score of textual relevance for C_4 equals to $0.7 + 0.7 = 1.4$.

Algorithm 5 prune($\mathscr{C}$) in AND semantics

```
1.  set all bits of $sig$ to be 1
2.  **for** $w \in \mathscr{C}.denseKwds$ **do**
3.    get the summary information $\mathscr{E}$ for $\langle w, \mathscr{C}.C \rangle$
4.    $sig \leftarrow sig \& \mathscr{E}.sig$
5.  **if** $sig = 0$ **then**
6.    **return TRUE**
7.  **if** $\mathscr{C}.docs$ is not empty **then**
8.    **for** $\mathscr{D} \in \mathscr{C}.docs$ **do**
9.      **if** $sig[\mathscr{D}.id] = 0$ **then**
10.       remove $\mathscr{D}$ from $\mathscr{C}.docs$
11.   **if** $\mathscr{C}.docs$ is empty **then**
12.     **return TRUE**
13. **return FALSE**
```

5.3 Query Processing for OR Semantics

Query processing for OR semantics differs from that of AND semantics in that a cell can be pruned only if it does not contain any query keywords. It requires $\mathscr{C}.docs$ to be empty and the intersection of the signatures from the dense keywords be 0. This is a

Algorithm 6 updateUpperScore($\mathscr{C}$) in AND semantics

```
1. calculate the spatial relevance $s.spatial$ of cell $\mathscr{C}.C$
2. $s.dense \leftarrow 0$
3. $s.non\_dense \leftarrow 0$
4. for $w \in \mathscr{C}.denseKwds$ do
5.     Get the summary entry $\mathscr{E}$ for $\langle w, \mathscr{C}.C\rangle$
6.     $s.dense \leftarrow s.dense + \mathscr{E}.s$
7. for $\mathscr{D} \in \mathscr{C}.docs$ do
8.     if $\mathscr{D}.score > s.non\_dense$ then
9.         $s.non\_dense \leftarrow \mathscr{D}.score$
10. $\mathscr{C}.upperScore \leftarrow s.spatial + s.dense + s.non\_dense$
```

stricter condition than that in AND semantics. Therefore, we need to examine much more candidates in OR semantics.

The calculation of the upper bound score in OR semantics is a bit tricky. Since any document containing a subset of query keywords is considered a candidate, we need to check the upper bound scores for all the possible subsets. The one with the maximum score is used to calculate $\mathscr{C}.upperScore$. We use the Apriori [1] algorithm to solve the problem. In the bottom level of the lattice, the subsets contain only one keyword. If this keyword is dense, its score is represented by $\mathscr{E}.s$. If not, we check any document in $\mathscr{C}.docs$ containing the keyword and use the real relevance score. Two query subsets in the same level can be merged only if we can find a common document id in these two sets. After the expansion finishes, we get the maximum score of all the candidate subsets as our upper bound score for textual relevance.

We still take cell C_4 as an example to show how to calculate the textual relevance. This time we assume the query keywords are "spicy Chinese restaurant". We know that "restaurant" is a dense keyword in C_4 while both "spicy" and "Chinese" are not dense. Suppose η is 4 and the hash function $H(id) = id \% \eta$, the signature for "restaurant" in C_4 is 1001 as it contains document $\{d_4, d_7, d_8\}$. Keyword "spicy" is contained in documents $\{d_4\}$ and "Chinese" in $\{d_7\}$. There is no document in C_4 containing all the query keywords. To get the upper bound score, we need to check the subsets of these three keywords. The upper bound score for each valid subset of query keywords is shown in Figure 4. For example, the score for {Chinese,restaurant} is $0.1 + 0.7 = 0.8$ and the score for {spicy,restaurant} is $0.7 + 0.7 = 1.4$. The final upper bound for textual relevance would be 1.4.

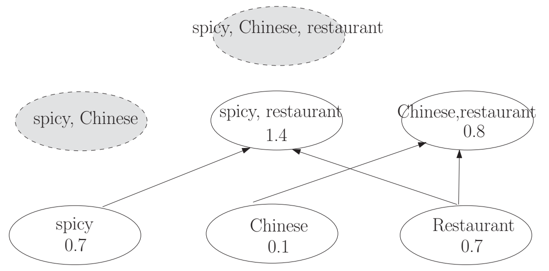


Figure 4: An example of calculating upper bound score in OR semantics

6. EXPERIMENT EVALUATION

In this section, we study the performance of I^3 and compare it with state-of-the-art approaches, including IR-tree and S2I, using real datasets Twitter and Wikipedia and real query log from AOL. The implementations of IR-tree and S2I were generously provided by the authors. We slightly modified the code to support both AND and OR semantics. We do not show experiment results of other variants of IR-Tree like DIR-tree and CDIR-tree as proposed in [6] because we found that these variants showed little improvement in query processing performance but took much longer time to build the index. All of the indexes were implemented in Java and experiments were conducted on a server with Quad-Core AMD Opteron(tm) Processor 8356, 64GB memory, running Centos 5.6.

Table 3: REST query sample

1	restaurant	2	restaurant equipment
3	Italian restaurant	4	restaurant supplies
5	used restaurant equipment	6	restaurant coupons
7	golden corrall restaurant	8	restaurant recipes
9	restaurant chairs	10	restaurant nyc

6.1 Dataset

We select Twitter and Wikipedia as two representative datasets. In the Twitter datasets, there are a large number of spatial tweets publicly accessible but each tweet contains only very few keywords. A tweet is designed to contain at most 140 characters and most of the keywords appear only once in one tweet. Thus, tweet is not textually abundant and the textual relevance may not play an important role in the ranking function. We extracted 15 million tweets which are associated with latitude and longitude. Wikipedia, on the other hand, is abundant in textual information. Each article contains hundreds of keywords. However, compared to Twitter, there are much fewer available articles associated with spatial attribute. Some articles related to gazetteer terms may be associated with coordinates. For example, in the article about "Forbidden_City"[4], there is a segment formatted in `{{coord|39|54|53|N|116|2 3|26|E}}`. We extracted around 400K articles with such format from the latest Wikipedia dataset.

Given the 15 million Twitter dataset, we also sampled 3 smaller ones for scalability test, which were named Twitter1M, Twitter5M and Twitter10M. The statistics of these datasets is shown in Table 2, including dataset cardinality, total number of unique keywords and the average number of keywords for each spatial document.

6.2 Query Set

In order for a comprehensive performance study, we generated two types of queries from a real AOL query log [5]: FREQ and REST. In FREQ, the queries are selected to contain frequent keywords. First, we picked queries that contain only two keywords. Then, we sorted these queries based on the frequency of keywords. The first 100 queries are selected as FREQ_2. We repeated the same procedure to generate FREQ_3, FREQ_4 and FREQ_5, in which each query contains 3, 4 and 5 keywords respectively. The other type REST is about restaurant, which is a common type of query in location based services. We selected 100 commonest queries containing keyword "restaurant". The top-10 query keywords are shown in Table 3. The location information in the above query sets is sampled from the spatial distribution of the Twitter data set.

6.3 Setup

We set $P = 4$KB in all the three indexes. For other parameters in IR-tree and S2I, we follow the same setting as reported in their ex-

[4]http://en.wikipedia.org/wiki/Forbidden_City
[5]http://www.gregsadetsky.com/aol-data/

Table 2: Dataset description

DataSets	Number of tuples	Number of unique keywords	Average number of keywords per document $\mathscr{D}$
Twitter1M	$1,000,000$	$441,457$	6.56
Twitter5M	$5,000,000$	$1,249,999$	6.54584
Twitter10M	$10,000,000$	$1,964,267$	6.5442
Twitter15M	$15,000,000$	$2,557,752$	6.54324
Wikipedia	$401,892$	$866,307$	129.941

Table 4: Parameter Setting

Number of query keywords *qn*	2, **3**, 4, 5
α	0.1, **0.3**, 0.5, 0.7, 0.9
k	10, **50**, 100, 150, 200

periments. During the query processing stage, we clear the system cache before we execute a query set so that the following query processing is not affected by the cache. Since each query set contains 100 queries, the average processing time and I/O cost are reported. For query processing of I^3, we load the lookup table into memory as it is quite small. The access to the head file and disk file are disk-based. The query parameters in the experiment setup are shown in Table 4. We are interested to evaluate query performance in terms of varying number of query keywords *qn* for `FREQ` query, spatial relevance weight α in the ranking function and the number of query results *k*. The value in *bold* represents the default setting.

Since in our implementation, each tuple in the datafile takes $B = 32$ bytes. Each page can hold at most $\lfloor P/B \rfloor = 128$ tuples. I^3 only has one parameter that we need to tune, which is the signature length η in the head file. Figure 5 shows the top-*k* query performance using AOL query set and the head file size in histogram based on Twitter1M dataset. As η increases, the pruning for both `AND` and `OR` semantics becomes better but more disk space is required. In the following experiments, we set $\eta = 300$.

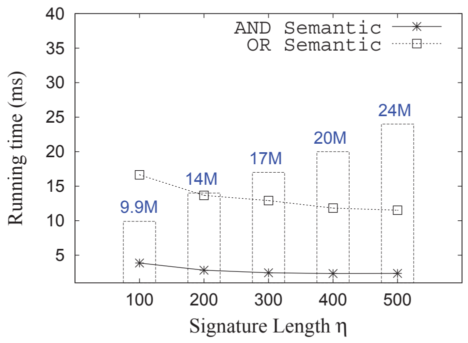


Figure 5: Performance tuning for η

6.4 Index Construction Time and Storage Cost

In this part, we examine the index construction time and storage cost. The time to build each index is shown in Figure 6. Note that building time is affected by various factors, such as cache size, index design and even code optimization. It is difficult to provide a thoroughly fair comparison. Here, we simply report the building time obtained based on the experiment setup discussed previously. As shown, I^3 takes the least time to build the index for the Twitter datasets as it provides a uniform storage and access interface for keyword cell. The construction cost of IR-tree is sensitive to the number of objects in the dataset. As we can see, the building time increases dramatically as the number of objects in Twitter grows from 1M to 15M. However, it performs very well in the Wikipedia dataset for two reasons. One is that Wikipedia only contains 400K objects which is much smaller than Twitter. The other is that the implementation is based on a static dataset. An R-tree is first built and then the inverted lists are i njected into the tree nodes. Hence, when a tree node is split, it does not need to re-organize the inverted lists in that node.

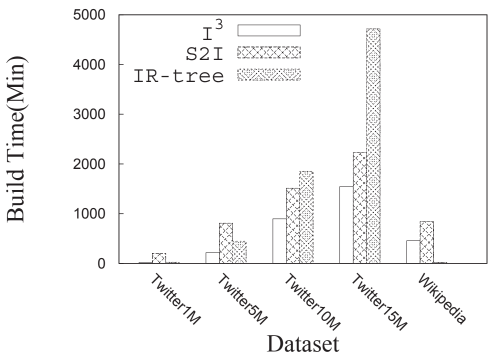


Figure 6: Index construction time

The storage size of different indexes is reported in Table 5. We report the head file and data file size for I^3. Among all the schemes, I^3 is the most storage efficient as it allows different keyword cells to be stored in the same disk page. It takes 2-3 times less storage than S2I. Moreover, S2I generates a large number of small index files. In the Twitter5M dataset, $111,702$ small files were created as part of the spatial tree index. IR-tree stores the pseudo-documents or inverted index in a separate file. Thus, we report both the R-tree size and the inverted index size. Unfortunately, the storage of inverted index is not optimized in the implementation. The experiment result shows it takes 623GB in the Twitter15M dataset.

6.5 Query Processing

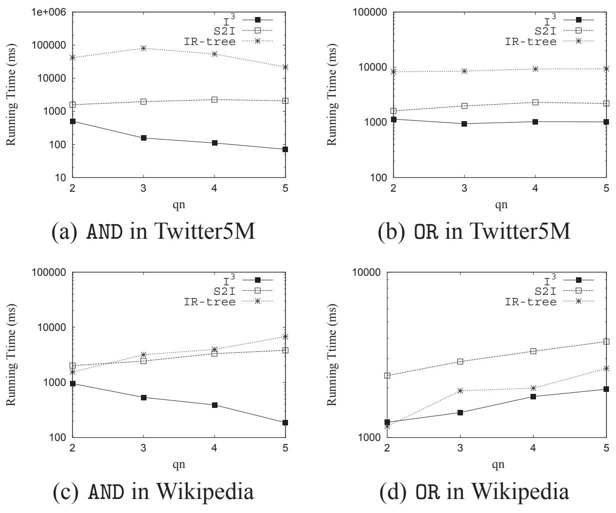
We study the index performance in both `AND` and `OR` semantics using the `FREQ` and `REST` query sets. We report the query processing performance in datasets Twitter5M and Wikipedia in terms of different experiment settings, including varying number of query keywords *qn*, number of top-*k* results and spatial relevance weight α in the ranking function. We also test the scalability of the index with respect to an increasing data cardinality in the Twitter dataset.

6.5.1 *Increasing number of query keywords*

In this experiment, we increase the number of query keywords

Table 5: Index size(GB)

	I^3		S2I	IR-tree	
	Data File	Head File	Index	Inv Index	R-tree
Twitter1M	0.21	0.01	0.6	3.4	0.06
Twitter5M	1.2	0.09	2.8	71	0.3
Twitter10M	2.2	0.16	4.2	287	0.6
Twitter15M	3.3	0.26	6.9	623	0.9
Wikipedia	1.8	0.12	4.1	8.1	0.26



(a) AND in Twitter5M (b) OR in Twitter5M

(c) AND in Wikipedia (d) OR in Wikipedia

Figure 7: Running time of increasing *qn*

qn from 2 to 5 using FREQ query set. Figure 7 shows the query processing time in both AND and OR semantics in the two datasets. We can see that I^3 demonstrates the best performance when handling frequent query keywords. This confirms that the index design and query processing algorithm are effective. The summary information of keyword cell can be retrieved efficiently and facilitate the pruning of search space. If the query is based on AND semantics, the running time even reduces as *qn* becomes larger. There are fewer candidates containing all the query keywords and most of the search space can be pruned using the intersection of signature file. The query processing time of I^3 can be more than an order of magnitude faster than IR-tree and S2I when *qn* is 4 and 5. When the query is based on OR semantics, the advantage is not so significant as that for AND semantics. However, I^3 still takes the least time to answer a query and shows scalable performance in terms of increasing query keywords.

The processing time of S2I is not sensitive to the query semantics. The reason is that it simply aggregates partial scores from different R-trees. It does not have any summary information like I^3 to detect whether a document id appears in all of these R-trees, which can be used for pruning in AND semantics. Moreover, the performance of S2I degrades as *qn* increases. When the keywords are frequent, it takes longer time to perform partial aggregation among different R-trees. Such join operations are expensive. It is worse than IR-tree when handling Wikipedia dataset in which textual information is abundant.

IR-tree shows the worst performance in the Twitter5M dataset. It takes more than 10s to answer a query with frequent keywords, which is in line with our analysis on its performance in the worst case. When the tree size is large and the query keywords are frequent, the cost to access tree nodes and inverted lists becomes very expensive. In Wikipedia, IR-tree performs better than S2I in OR semantics as the tree is quite small, containing around 100 internal nodes. Note that IR-tree demonstrates better performance in OR than in AND semantics. We found that the score of 50-th document in top-*k* results is much larger than that in AND semantics. This makes the pruning more effective in OR semantics. Many nodes in IR-tree are pruned because their upper bound scores are smaller than the 50-th score.

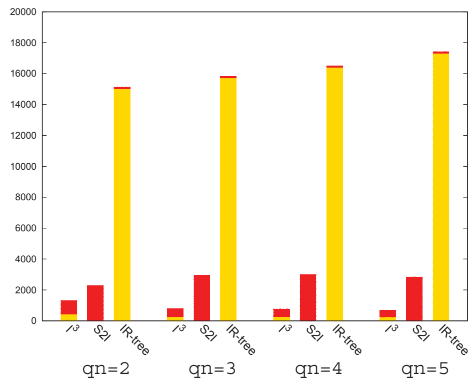


Figure 8: I/O cost of increasing *qn* in OR semantics in Twitter5M

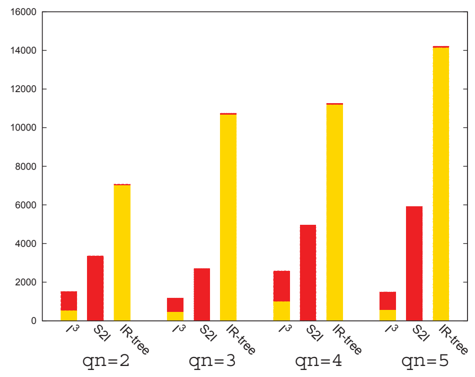
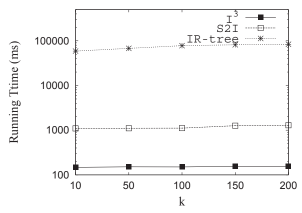


Figure 9: I/O cost of increasing *qn* in OR semantics in Wikipedia

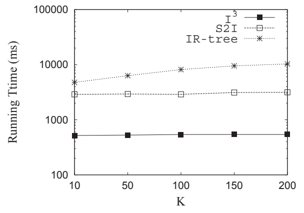
The IO cost results of different indexes answering queries in OR semantics in Twitter5M and Wikipedia are shown in Figures 8 and 9. For I^3, we report IO cost caused by accessing both summary nodes in the head file (the gold part in the bottom of the histogram in the figure) and the disk pages in the data file (the red part on the top of the histogram in the figure). For S2I, since all the keywords are frequent, S2I builds an R-tree for each query keyword and only the tree nodes are accessed. Hence, only the number of tree IOs is shown in the figure. There is no sequential access to the infrequent keywords stored in the flat file. In IR-tree, the IO cost consists of the access to tree nodes (the red part on the top of the histogram) and their associated inverted file (the gold part in the bottom of the histogram). As we can see, it is incredibly expensive to access the inverted file associated with tree nodes in IR-tree. The reason is that in the implementation, a B-tree is built for each inverted file to facilitate the retrieval of relevant document lists for the query keywords, which leads to a large number of I/O access. I^3 shows the best performance as *qn* increases.

6.5.2 *Increasing number of query results*

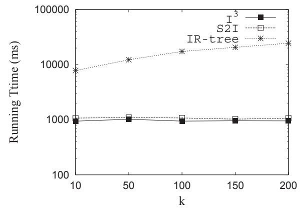
The results of increasing k are shown in Figure 10. The caption of each sub-figure represents the query semantics, dataset and query type used in the experiment. We can see that IR-tree is not scalable in terms of k. As k increases, the k-th score becomes larger. IR-tree needs to examine much more tree nodes as the pruning becomes less effective. Since the tree node is augmented with inverted lists, the cost to access tree nodes is expensive. S2I shows stable performance in the Twitter dataset but is sensitive to k in the Wikipedia dataset. I^3 is scalable to k in the two datasets in both `AND` and `OR` semantics.



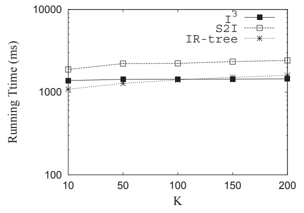
(a) `AND` / Twitter5M / `REST`



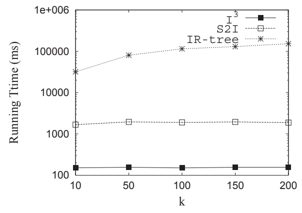
(b) `AND` / Wikipedia / `REST`



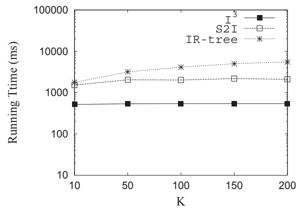
(c) `OR` / Twitter5M / `REST`



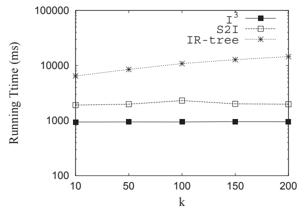
(d) `OR` / Wikipedia / `REST`



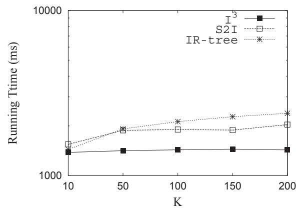
(e) `AND` / Twitter5M / `FREQ`



(f) `AND` / Wikipedia / `REST`



(g) `OR` / Twitter5M / `FREQ`



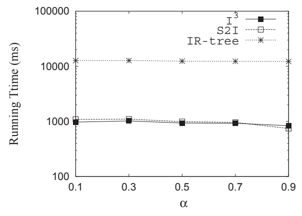
(h) `OR` / Wikipedia / `REST`

Figure 10: Increasing number of query results

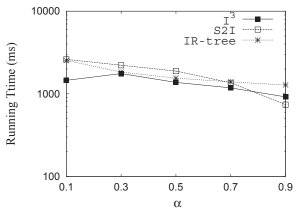
6.5.3 *Varying α in the ranking function*

In Figure 11, we study the impact of spatial relevance weight α. The results demonstrate interesting patterns in two different datasets. In Twitter, most of the keywords in a tweet appear only once. For documents containing keyword w_i, the term weight of w_i is basically the same in these documents. In other words, the ranking function is mainly determined by the spatial relevance. It is difficult for a document whose associated location is far away from the query location to be ranked in the top results. Thus, the performance is irrelevant to α in Twitter. No matter how α varies, the documents whose associated locations are near the query location are returned.

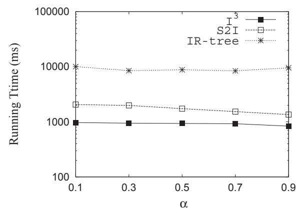
In Wikipedia, we can see that the effect of α in the ranking function. Among the three types of index, S2I is the most sensitive to α. When α is small, spatial relevance becomes not important. The spatial pruning of R-tree in S2I is not useful and most of the tree nodes have to be accessed. However, when α is close to 1, the ranking function is mainly determined by the spatial proximity. S2I performs very well in this case. For IR-tree and I^3, their performance gets better as α increases but not so significantly as S2I.



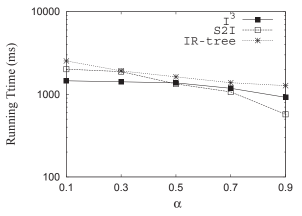
(a) `OR` / Twitter5M / `REST`



(b) `OR` / Wikipedia / `REST`



(c) `OR` / Twitter5M / `FREQ`



(d) `OR` / Wikipedia / `REST`

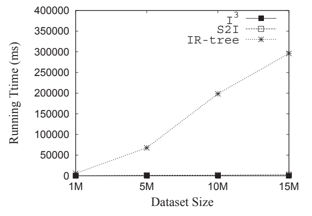
Figure 11: Varying α

6.5.4 *Increasing cardinality of Twitter dataset*

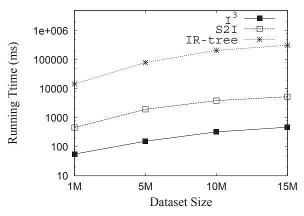
In this experiment, we examine the performance in terms of increasing cardinality of dataset. We tested on four datasets: Twitter1M, Twitter5M, Twitter10M and Twitter15M. The running time is shown in Figure 12. We plot the time result in `REST` queries without log scale. As we can see, I^3 and S2I demonstrate better scalability than IR-tree in the Twitter dataset.

6.6 Update Cost

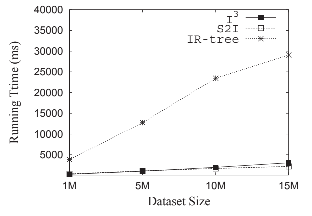
We also compare the update cost in I^3 with S2I in Twitter and Wikipedia. We did not compare with IR-tree as the update implementation was not provided. Moreover, it has been reported that S2I is more update efficient than IR-tree [17]. To compare the update performance, we first build the index to a moderate size, then execute $4,000$ randomly generated data operations, including insertion and deletion of spatial documents, and finally flush the update back to disk index. Table 13 shows the update cost for different data size. I^3 has a clear advantage over S2I as it has more flexible and efficient I/O access mechanism. The reasons are twofold. First, S2I maintains an inverted file for all the infrequent keywords and builds an R-tree file for each frequent keyword. When a keyword becomes frequent, its inverted list needs to be removed and all the items in the list are re-inserted into an R-tree. This causes considerable I/O cost. Second, it is difficult for S2I to take advantage of disk locality to save disk seek time and rational latency as spatial objects are more likely to be stored in different R-tree files. Compared to S2I, I^3 provides uniform storage mechanism to frequent



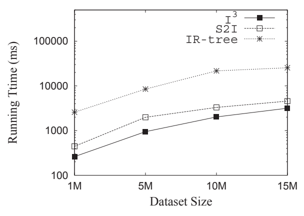
(a) AND using REST



(b) AND using FREQ



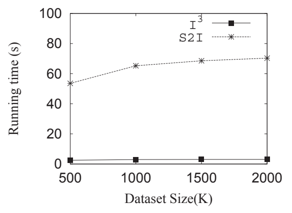
(c) OR using REST



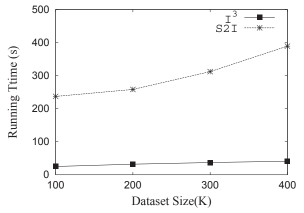
(d) OR using FREQ

Figure 12: Increasing cardinality of Twitter

keywords and infrequent keywords. Thus, its update cost is much cheaper.



(a) Twitter



(b) Wikipedia

Figure 13: Index update time

7. CONCLUSION

In this paper, we presented a scalable index named I^3 for efficient top-k spatial keyword search. We used Quadtree for space decomposition and proposed the concept of keyword cell as the basic storage unit. The keyword cell captures spatial locality for a keyword. We designed a uniform storage mechanism for frequent and infrequent keywords. For effective pruning, we built an associated head file to store summary information for dense keyword cell. Our index supports both AND and OR semantics. Experiment results verified the advantages of I^3. It not only took less building time, update overhead and storage cost, but also outperformed existing methods in answering queries in large-scale datasets.

8. ACKNOWLEDGEMENT

This work is funded by the NExT Search Centre, which is supported by the Singapore National Research Foundation and Interactive Digital Media R&D Program Office, MDA under research grant R-252-300-001-490.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.

[2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD Conference*, pages 373–384, 2011.

[4] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD Conference*, pages 277–288, 2006.

[5] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[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] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4):267–288, 1984.

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

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

[10] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In *SSDBM*, page 16, 2007.

[11] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *SIGMOD Conference*, pages 546–557, 2002.

[12] A. Khodaei, C. Shahabi, and C. Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA (1)*, pages 450–466, 2010.

[13] G. Li, J. Feng, and J. Xu. Desks: Direction-aware spatial keyword search. In *ICDE*, pages 474–485, 2012.

[14] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. Ir-tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[15] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, pages 563–574, 2006.

[16] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.

[17] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.

[18] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.

[19] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532, 2010.

[20] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.

[21] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.