Trajbase: Searching Trajectories in Multi-region

Renjie Zheng Qin Liu Hanjun Mao Hongming Zhu Weixiong Rao

School of Software Engineering
Tongji University, China
Corresponding Authors: {qin.liu, zhu_hongming, wxrao}@tongji.edu.cn

ABSTRACT

The recent years witnessed popular use of smart phones and wearable devices. Such mobile devices are widely equipped with GPS sensors to record their geographical positions, generating massive spatial trajectory data. Many real applications, such as urban computing and intelligent transportation systems, search the trajectory data. To process such queries, it is important to meet two objectives including fast running time and low space cost. Previous work frequently only meets one objective but compromises another. In this work, we design a compact data structure TrajBase to index trajectory data. The novelty of such an indexing structure is a carefully designed encoding approach. The approach can achieve comparable compression ratio compared with the state of the art approach and fast building time. Extensive evaluation with two real trajectory datasets can successfully verify that TrajBase outperforms previous works in terms of query processing time and indexing space cost.

1. INTRODUCTION

The recent years witnessed popular use of smart phones and wearable devices. Such devices are widely equipped with GPS sensors to record their geographical positions, generating massive spatial trajectory data. The trajectory data has been widely used to understand the mobility of various moving objects such as people and vehicles, and to build various application including urban planning and intelligent transportation systems. One task is to find the trajectory data that passes multiple input regions of interest. For example, in urban computing applications, it is required to discover city residents' living and working locations in order to understand their movement flow. Both the living and working locations are the input regions of the discovery query. In transportation systems, casual analysis can identify the source area leading to traffic jam and the victim area suffering from the heavy traffic jam. The source and victim areas are as the input regions of the casual analysis.

Given the task above, we want to design an efficient query processing approach satisfying two objectives: (i) fast running time to process such queries and meanwhile (ii) low space cost to index the trajectory database. Given the massive trajectory data, low space cost of the index is important. More specifically, when the size of massive trajectory is out of main memory capacity, we have to store the trajectories on disk and instead maintain the index structure in main memory as much as possible. Then, the query running time depends upon how the index structure processes the query and minimizes the I/O operations to retrieve the final query result (i.e., needed trajectories) on disk. It is not hard to find that as the lower layer, our work is the fundamental to support more complex queries, such as k-nearest neighbor (KNN) queries [3], in spatial database. Thus, it is important to design an efficient algorithm to answer the defined query in this paper.

Unfortunately, it is a nontrivial problem to meet two objectives above. Some approaches (such as a classic Zip algorithm) can compress an indexing structure of trajectory data for low space cost, but requiring decompression before processing queries. The decompression heavily harms the query processing time.

In this paper, we design a compact trajectory indexing structure, namely TrajBase, to achieve the above two objectives. TrajBase consists of two level of structures, a first-level classic spatial index (we use quad-tree as an example) and a secondary level index (consisting of lists of integer numbers). The integer numbers are to encode trajectory IDs for low space cost. Meanwhile, by very fast bitwise operation `AND` and `OR` directly on the encoded integer numbers, we can perform query processing but without decompression of such encoded IDs for fast searching time. As a summary, this paper makes the following contributions:

1. First, we design an efficient encoding algorithm to build TrajBase by using fast running time and meanwhile achieving a compression ratio comparable to the state of art approach [14].

2. Second, we propose an algorithm to efficiently process the query to find those trajectories passing multiple input regions. The key of this algorithm is to use the bitwise operation `AND` and `OR` directly on the encoded numbers in TrajBase, but without decoding such numbers. Thus, the query processing algorithm can achieve very fast query processing time.

3. Finally, based on two real data sets, we conduct extensive experiments to compare TrajBase with the state of the arts. The experimental results verify that TrajBase outperforms previous work in terms of space and running time.

The rest of the paper is organized as follows. Section 2 gives the problem definition and introduces TrajBase. Next, Section 3 designs the proposed encoding algorithm. After that, Section 4 presents the query processing algorithm. Section 5 evaluates the performance of TrajBase, and Section 6 investigates related work. Finally, Section 7 concludes the paper.

UrbanGIS 16, October 31-November 03 2016, Burlingame, CA, USA

DOI: http://dx.doi.org/10.1145/3007540.3007543

2. PROBLEM DEFINITION AND TRAJBASE OVERVIEW

2.1 Problem Definition

We consider a trajectory database T, where each trajectory $t \in T$ consists of a set of quadruples $qr = \langle tid, x, y, ts \rangle$. Here, tid is a unique trajectory ID of t, meanwhile x and y represent the longitude and latitude of a specific location at a timestamp ts. Typically the quadruples in t are ordered by ascending order of timestamp ts.

Next, we define an input query q consisting of a set of regions of interest R. Without loss of generality, we represent a *region* in form of rectangle. With the query q and database T, we want to find a subset of trajectories $S \sqsubseteq T$: for each trajectory $t' \in S$, there exist quadruples $qr' \in t'$ with the locations x' and y' in the quadruples qr' falling within *every* region $r \in R$. For each trajectory $t' \in S$, we return all quadruples $qr' \in t'$.

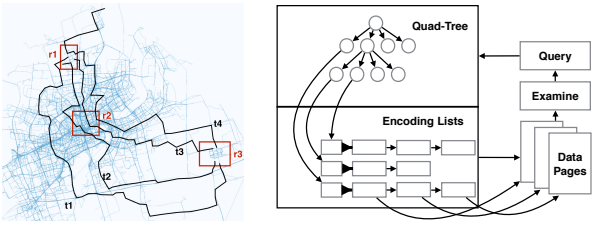


Figure 1: Query example and architecture (a) Query $q = \{r_1 ... r_3\}$ and Query result $S = \{t_1 ... t_4\}$ (b) Architecture

Intuitively we need to find those trajectories going through *all* regions R and next return the quadruples of such found trajectories. Figure 1 (a) plots one of our real trajectory data (i.e., Shanghai data set) used in our experiment. Consider a query q consisting of 3 input regions r_1, r_2, r_3. Among those trajectories $T = t_1 ... t_4$, only three trajectories $S = t_2, t_3, t_4$ have passed through all three regions.

2.2 Solution Overview

In this section, we give an overview of the proposed index structure and next highlight the query processing algorithm.

2.2.1 *Indexing Structure*

Fig. 1 (b) shows the basic architecture of Trajbase to index the trajectories on disk. First, we use the traditional spatial index (e.g., quadtree [6]) to index those trajectories. Based on the quadtree, we further maintain a list of encoded numbers (in short *encoding lists*) in each leaf node of the quadtree. Here, we do NOT simply maintain a sorted list of trajectory IDs. Instead, as the *novelty* of our work, the encoding lists maintain a list of integer numbers to encode the IDs of those trajectory passing through leaf cells.

Maintaining such encoded numbers offers twofold benefits: (i) When the number of trajectories passing through leaf nodes is large, the space cost associated with trajectory IDs is high. Thus, we propose to encode such trajectory IDs to compress several IDs into one encoded number and maintain much *shorter* encoding lists of these IDs, rather than the list of original trajectory IDs, for less space cost. (ii) The encoding lists of encoded numbers can speedup the query processing by performing intersection on encoded numbers, when query q involves multiple input regions. Moreover, the proposed query processing algorithm avoids the scan of entire encoding lists, leading to much faster running time.

We define a threshold M in order to balance the number of trajectory IDs in each leaf node. In this way, the threshold M can limit the size of each encoding lists L_i which is used to reduce the number of trajectory IDs in each leaf node. It is not hard to find that the threshold M balances the number of trajectories among all leaf nodes and thus achieve faster running time to process query q.

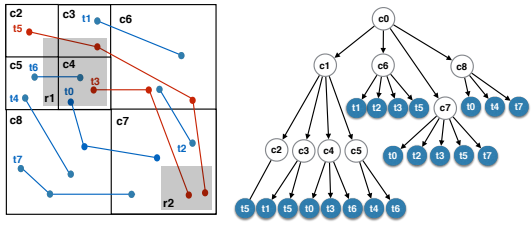


Figure 2: Quadtree with threshold $M = 5$: (a) Extent (b) Quadtree

Fig. 2 shows an example to index 8 trajectories (with threshold $M = 5$). The spatial extent is split into 7 leaf nodes as shown in Fig. 2(a) and the quadtree is built as shown in Fig. 2(b). In this quadtree, the root is divided into four cells c_1, c_6, c_7 and c_8. There exists 6 trajectories passing through cell c_1. Since such a number is larger than the threshold $M = 5$, cell c_1 is further divided to four smaller ones c_2, c_3, c_4 and c_5.

2.2.2 *Query Processing*

With the indexing structure TrajBase above (consisting of quadtree and encoding lists), we show how an input query q is processed by the following three steps: (i) lookup of quadtrees to find valid leaf nodes, (ii) search of encoding lists to retrieve the IDs of candidate trajectories, and (iii) final validation of the trajectory candidates on disk in order to find final results.

In more detail, for every input region $r \in R$ in query q, we find those leaf nodes in quadtree Q which overlap r. Denote all such leaf nodes to be N_r. For each leaf node $n \in N_r$, we retrieve encoding lists (denoted by L_n) with respect to leaf node n. Then, we can find the candidate results of q by

$$\bigcap_{r \in R} \left[\bigcup_{n \in N_r} L_n \right] \tag{1}$$

In the equation above, for a specific region $r \in R$, we have multiple leaf nodes N_r overlapped by r; the subitem $\bigcup_{n \in N_r} L_n$ performs a union operation over the encoding lists L_n of those leaf nodes $n \in N_r$. Thus, the subitem is to find the trajectory IDs appearing in any of such leaf nodes $n \in N_r$. After that, the item $\bigcap_{r \in R} \left[\bigcup_{n \in N_r} L_n \right]$ next performs an intersection over the union results with respect to all regions $r \in R$.

Fig. 2(a) shows two query regions on the extent. Region r_1 overlaps four leaf nodes of the quadtree Q, and thus we have leaf nodes $N_{r_1} = \{c_2, c_3, c_4, c_5\}$. Then we retrieve the associated encoding lists $L_{c_2} = \{t_5\}, L_{c_3} = \{t_1, t_5\}, L_{c_4} = \{t_0, t_3, t_6\}, L_{c_5} = \{t_4, t_6\}$. After the union operation on these trajectories, we have the candidate trajectories overlapped by region r_1: $\bigcup_{n \in N_{r_1}} L_n = \{t_0, t_1, t_3, t_4, t_5, t_6\}$. Similarly, for region r_2, we have the candidate trajectories $\bigcup_{n \in N_{r_2}} L_n = \{t_0, t_2, t_3, t_5, t_7\}$. Then, we perform an intersection operation on those intermediate candidate trajectories and have the final candidate trajectories $\{t_0, t_3, t_5\}$.

Finally, with the intersection result above given by Equation 1, we need the 3rd step to make the final validation. It is because a leaf node, though overlapping an input region r, may not contain the quadruples qr falling within the region r. As the example shown

in Fig. 2(a), the candidate trajectory t_0 fails passing through r_2 and will be ruled out in this final step. Thus, the final validation step can correctly find the final result.

3. DESIGN OF ENCODING LISTS

3.1 Overview

Recall that an encoding list w.r.t a leaf node is to encode the IDs of those trajectories passing through the leaf node. Here, as the novelty idea of our work, we treat the entire encoding lists of all leaf nodes as a single data structure, and make an optimization for least space cost.

In general, each leaf node contains multiple trajectory IDs, and a trajectory ID might appear in multiple leaf nodes. In order to represent the membership between leaf nodes and trajectory IDs, we model a membership matrix $X_{|N|\times|T|}$, where $|N|$ is the number of leaf nodes and $|T|$ is the number of trajectories. In the matrix $X_{|N|\times|T|}$, each row (resp. column) represents a leaf node (resp. trajectory). Then, if the i-th leaf node contains the j-th trajectory ID (with $1 \leq i \leq |N|$ and $1 \leq j \leq |T|$), we have the binary element $X_{i,j} = 1$ and otherwise $X_{i,j} = 0$.

With the binary matrix $X_{|N|\times|T|}$ above, it could be very large and sparse. If we simply maintain the full matrix $X_{|N|\times|T|}$, the associated space cost is very high, at the scale of $O(|N| \cdot |T|)$. We will propose to compress the matrix for low space cost.

3.2 Baseline Approach

In this section, following the basic idea of inverted lists, we propose a baseline approach to maintain matrix $X_{|N|\times|T|}$. Specifically, for each row in $X_{|N|\times|T|}$, we sort the column IDs with those non-zero elements, and then maintain a sorted list of such column IDs. Given all rows in $X_{|N|\times|T|}$, we then maintain a directory of such rows. Each element in the directory refers to the associated sorted list of column IDs.

Fig. 3(a) gives an example to maintain 8 trajectory IDs by the inverted lists-based baseline approach. For example, in the leaf node of c_3, 2 trajectories (t_1 and t_5) go through c_3, and thus the posting list of c_3 is with t_1 and t_5; in leaf node of c_6, the associated posting list is with 4 sorted trajectory IDs (t_1, t_2, t_3 and t_5).

Compared with the naive approach to fully maintain matrix $X_{|N|\times|T|}$, the baseline approach leads to much less space cost, at the scale of the number non-zero elements in the matrix. Nevertheless, the baseline approach does not help the speedup of query processing. In the next section, we propose to further reduce the space cost of maintain $X_{|N|\times|T|}$ by compressing inverted lists and speedup the query processing based on the compression approach.

3.3 Encoding the Matrix

Our basic idea to encode matrix $X_{|N|\times|T|}$ essentially is a hybrid of the inverted list and matrix. It first compresses the matrix and next maintains an inverted list-like structure. More specifically, by a base number B, we divide the $|T|$ columns of matrix $X_{|N|\times|T|}$ into $\lceil |T|/B \rceil$ groups. Each group is with at most B columns. Next, for each row of $X_{|N|\times|T|}$, the associated $|T|$ binary elements are divided to $\lceil |T|/B \rceil$ groups. Given each group of binary elements, we then encode the binary elements into an integer number, by treating such elements as the binary bits of the encoded integer number. We denote such encoded number *eid*, and each row is with at most the number $\lceil |T|/B \rceil$ of *eid*s.

Consider that *eid* could be zero. For saving space cost, we do not maintain those zero *eid*s. Next, with each of those non-zero *eid*s, we record the column ID of the left-most element encoded by such an *eid*. We denote such an ID to be *cid*. Then, we use the pair $\langle cid, eid \rangle$ to maintain the group of binary element. For each row of $X_{|N|\times|T|}$, we have at most $\lceil |T|/B \rceil$ integer pairs $\langle cid, eid \rangle$, where *eid* is a non-zero integer number. Based on such pairs, we maintain a sorted list of such pairs in ascending order of *cid* values.

We use Fig. 3(b-c) as an example. With base number $B = 2$, we divide the 8 columns into 4 groups. For each row of the matrix, we have 4 groups of binary elements and encode them into 4 integer pairs. For example, for c_6, the original bits 01110100 are divided into 4 groups 01, 11, 01 and 00, which are encoded into four integers 1, 3, 1 and 0. To save space cost, we only maintain non-zero encoded numbers, and thus we maintain 3 integer pairs $\langle 0, 1 \rangle$, $\langle 2, 3 \rangle$ and $\langle 4, 1 \rangle$.

It is not hard to find that the space cost of the proposed approach linearly depends upon the number of non-zero *eid*s. To optimize the number of such non-zero *eid*s, we introduce an optimization approach by re-ordering the columns of matrix $X_{|N|\times|T|}$ (Section 3.4 will give the detail to re-order the columns). After the re-ordering, the zero-elements in $X_{|N|\times|T|}$ are clustered together (or equivalently non-zero elements are also clustered). After that, we can adopt the encoding approach above onto the re-ordered matrix again with the purpose to maintain encoding lists consisting of fewer pairs.

Fig. 3(d) shows an example to re-order columns of the matrix in Fig. 3(b). After that, Fig. 3(e) is with 14 integers pairs, smaller than the 18 ones in Fig. 3(c).

3.4 Re-ordering Matrix Columns

Based on the basic idea of re-ordering matrix columns, in this section, we formally define the re-ordering problem and then propose a solution.

Problem 1 *Given a binary matrix $X_{|N|\times|T|}$, we want to re-order the matrix columns, such that the new matrix $X'_{|N|\times|T|}$ after the re-ordering is with the least space cost.*

Depending upon various applications, the cost of re-ordering $X_{|N|\times|T|}$ can be measured by associated metrics. For example, [10] is interested in maximal sequence of non-zero entries in a row and then measures the cost by the sum of the number of such sequence in all rows. [14] instead measures the cost by the number of non-zero encoded numbers *eid*s. [12] uses a much complex function to compute the cost. Nevertheless, all of such previous work have proved that Problem 1 is NP-hard and there is no efficient solution. Therefore, we have the following lemma.

Lemma 1 *Problem 1 is NP-hard and there is no efficient solution.*

Due to the NP-hard Problem 1, the previous work have proposed either approximation or heuristic solutions to solve it. For example, [10] first built a graph to represent the matrix and used traveling salesman heuristics, such as nearest neighbor (NN), 2-OPT, 3-OPT and Lin-Kernighan. [14] needed to compute a pairwise similarity of matrix columns and repeatedly selected a best column among remaining ones by extending the classic threshold algorithm (TA)-based top-k algorithm. [12] instead focused on a graph-based approach where a key step is to repeatedly compute a maximal connected component. We note that all such previous work suffer from the weakness of slow running time, caused by either non-trivial preprocessing overhead or inefficiency of the proposed re-ordering approaches themselves.

To tackle the above inefficiency issue, we propose a novel algorithm to re-order the columns of matrix $X_{|N|\times|T|}$ by the following basic idea. Given the quadtree, we would like to first perform breath-first-search (BFS) on the tree. Based on the sequence order of traversing quadtree's leaf nodes, we re-assign a new column ID

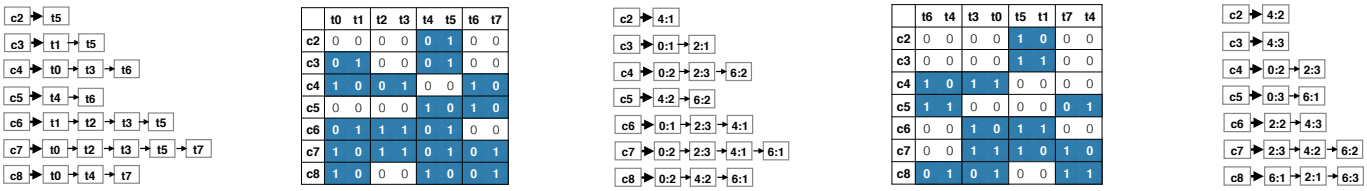


Figure 3: From left to right: (a)Inverted list; (b)Original Matrix; (c) Original Encoding Lists; (d) Re-ordered Matrix; (e) Optimized Encoding Lists

for the latest met trajectory in leaf nodes. In this way, those trajectory in the leaf nodes with the highest depth are re-assigned with smallest IDs.

In order to understand the idea above, we take the quadtree in Fig. 2(Right) as a running example. After the BFS on the quadtree, we have a following order to traverse the trajectories in leaf nodes:

$\mathcal{T} = \{t_1, t_2, t_3, t_5, t_0, t_2, t_3, t_5, t_7, t_0, t_4, t_7, t_5, t_1, t_5, t_0, t_3, t_6, t_4, t_6\}$

Given such ordered trajectories IDs, we care about the trajectories that are last met. Thus, we put them into a stack and skip the duplicate trajectory IDs and have the following new order:

$\mathcal{T}' = \{t_6, t_4, t_3, t_0, t_5, t_1, t_7, t_4\}$

With the new order, we re-assign new IDs for the trajectories.

$t_6 \rightarrow 0$, $t_4 \rightarrow 1$, $t_3 \rightarrow 2$, $t_0 \rightarrow 3$,$t_5 \rightarrow 4$, $t_1 \rightarrow 5$, $t_7 \rightarrow 6$, $t_4 \rightarrow 7$.

With the running example above, we explore the behind rationale of the BFS-based re-ordering approach. First, following the idea of BFS traversal, the trajectories in the same depth are at physically close locations. Thus, the BFS-based approach guarantees that the trajectories in the same depth are re-assigned with consecutive IDs. Second, following the definition of quadtree, we divide the area with more trajectories into smaller subareas, leading to leaf nodes of higher depth. It indicates that the leaf nodes with higher depth are clustered with more trajectories. In this way, for trajectories in the leaf nodes with higher depth, we then would like to put higher priority to re-assign such trajectories with consecutive IDs than those with lower depth. Consequently, the numerically consecutive IDs truly indicate the physical locality of associated trajectories.

The basic idea above looks quite simple, but *unifies* the numerically consecutive IDs with the physical locality. Thus, it is not hard to find that the proposed re-assignment can lead to a surprisingly high compression ratio that is comparable to the classic work [10, 14, 12]. Fig. 3(d) shows the matrix with the re-ordered columns, leading to the space cost of 14 pairs, less than the original cost of 18 pairs before re-ordering the matrix columns. With a larger base number $B > 2$, the space cost will be further reduced and smaller than the space cost of inverted list. Our evaluation will further verify that our re-ordering approach can achieve comparable compression ratio and meanwhile leads to much faster running time.

The pseudocode in Alg. 1 describes the detail of our proposed BFS-based re-ordering approach. The running time of Alg. 1 depends upon the size of quadtree Q and number of trajectories. Since we need to traverse the quadtree and re-order the trajectories, we have the complexity $O(|T| + |Q|)$, where $|T|$ is the number of trajectories and $|Q|$ is the total number of nodes in the quadtree Q. Compared with previous work [14, 10, 12] requiring at least the complexity of $O(|T|^2)$, Alg. 1 is much more efficient.

4. QUERY PROCESSING

In this section, we describe the details of processing a query $q(Q, R)$, where Q is a quadtree and R is a set of input regions. Here each leaf node in Q refers to an element in encoding lists. To process a query $q(Q, R)$, TrajBase involves two steps. First, TrajBase searches the quadtree Q to select trajectory candidates. Next, we validate such candidates to find the correct results. It is not hard to identify that the 1st step is the key of processing the query $q(Q, R)$, and we focus on the optimization of 1st step. Once the candidates are ready, it is comfortably easy to find the final result by checking whether or not each of the candidates is the correct result of q.

Algorithm 1: BFS-based reassignment

Input: Quadtree Q
Output: New $ID[t]$ reassigned for trajectories t indexed by Q

```
1  initiate a binary indicator I[t] for every trajectory t in T to be 0: I[t] ← 0;
2  initiate an empty queue P, an empty stack S, and an integer i to be 0;
3  for each node u in Q, set visited[u] to be 0;
4  push the root of Q to P;
5  while P is not empty do
6      pop a node u from P;
7      foreach child w of u do push w to P;
8      if u is a leaf node then
9          foreach trajectory t in the leaf node u do push t to S ;
10 while S is not empty do
11     pop a node t from S ;
12     if visited[t] == 0 then visited[t] ← 1; ID[t] ← i; i ← i + 1 ;
```

4.1 Framework of Candidate Selection

Alg. 2 gives the pseudocode of the 1st step (namely candidate selection). The algorithm still follows the BFS approach (lines 3-11) to search the quadtree Q. For each region $r_i \in R$, we use a set array $N[]$ to maintain those leaf nodes such that the associated areas overlap region r_i (lines 7). When a member $N[i]$ in the array maintains a set of leaf nodes, we perform a union operation on encoding lists in such leaf nodes (line 13). The set $U[i]$ maintains the union result w.r.t member $N[i]$. After that, we perform the intersection operation on all sets $U[i]$ w.r.t all regions (line 14). Lines 12-14 exactly follows the Eq. (1).

In the selection algorithm above, the BFS requires the complexity of $O(\log |Q|)$ to traverse the quadtree Q because we only need to traverse the nodes which is overlapped with the given regions $r \in R$. Instead, when the total number of encoding lists in the set $U[]$ is large, the cost to perform union and intersection operation (lines 12-14) is non-trivial. In the rest of this section, we propose the algorithms to optimize the union and intersection of encoding lists.

4.2 Union and Intersection

In this section, we design an algorithm to optimize the union and intersection of encoding lists in a set N of leaf nodes.

Recall that each leaf node $n \in N$ maintains an associated encoding lists and each encoding list consists of a sorted list of $\langle cid, eid \rangle$ pairs in ascending order by cid values. Given a set L of leaf nodes,

Algorithm 2: Candidate Selection

Input: query regions R, Quadtree Q
Output: Set C of trajectory candidates
1 initiate an empty sets C and two set arrays $U[i]$ and $N[i]$ for each region $r_i \in R$;
2 initiate an empty queue P; push the root of Q to P;
3 **while** *P is not empty* **do**
4 pop a node n from P;
5 **if** *n is leaf node* **then**
6 **foreach** *region $r_i \in R$* **do**
7 **if** *the area of node n overlaps region r_i* **then** add n to $N[i]$;
8 **else**
9 **foreach** *child w of n* **do**
10 **foreach** *region $r_i \in R$* **do**
11 **if** *the area of node w overlaps region r_i* **then** push w to P ;
12 **foreach** *region $r_i \in R$* **do**
13 **foreach** *leaf node $n \in N[i]$* **do** $U_i \leftarrow \{U_i \bigcup$ encoding lists in $n\}$;
14 $C \leftarrow \{U \bigcap U[i]\}$;

we now have sorted lists of $\langle cid, eid\rangle$ pairs w.r.t such leaf nodes. Among such sorted lists, we note that an encoded trajectory ID might duplicately appear in multiple lists. Thus, the purpose of the union operation is to remove such duplicate IDs.

Algorithm 3: Union of Encoding Lists

Input: $|L|$ encoding lists $L_1 ... L_{|L|}$
Output: Union result
1 initiate an empty encoding list $\mathcal{U}$;
2 use min. heap $H_{[1:|L|]}$ to maintain head pairs of all encoding lists $L_1 ... L_{|L|}$;
3 **while** *at least one encoding list not reach end position AND H is not empty* **do**
4 $cid \leftarrow H_{[1]}.cid$ and $eid \leftarrow H_{[1]}.eid$;
5 remove 1st item $H_{[1]}$ from heap H;
6 move encoding list of $H_{[1]}$ to next position, re-insert the current pair to H;
7 **while** *H contains at least one item AND $H_{[1]}.cid == cid$* **do**
8 $eid \leftarrow eid \sqcup H_{[1]}.eid$;
9 **if** *$H_{[1]}$ does not reach end position* **then**
10 remove 1st item $H_{[1]}$ from heap H;
11 move $H_{[1]}$ to next position, add the current pair of $H_{[1]}$ to H;
12 add the pair $\langle cid, eid\rangle$ to $\mathcal{U}$;
13 **return** $\mathcal{U}$;

Alg. 3 gives the pseudocode to union $|L|$ encoding lists. It uses a minimal heap to maintain the number $|L|$ of $\langle cid, eid\rangle$ pairs at current positions of all encoding lists. With the heap, we maintain such sorted pairs by ascending order of their *cid* values. In this way, all pairs in the encoding lists are sorted and each of them is accessed by only one time. Thus, the runtime complexity of the algorithm is $O(\sum_{i=1...|L|} |L_i|)$, i.e., the total number of pairs in all encoding lists.

The key part of the algorithm is the `while` loop (lines 7-11). Its main purpose is to perform the union on the pairs with same *cid* values and then add the union result of associated *eid* values to the result set $\mathcal{U}$.

Alg. 4 gives the pseudocode of intersection operation. It is similar to the union algorithm, except the two lines 8 and 13. Based on the definition of intersection, we require that there exist the number $|L|$ pairs with the same *cid* values. Thus, in line 13, the condition $counter == |L|$ guarantees the semantical correctness of the intersection result. In addition, another condition $eid > 0$ indicates that at least one trajectory ID is encoded by all $|L|$ encoding lists. The running time complexity of Alg. 4 is $O(\sum_{i=1...|L|} |L_i|)$.

5. EVALUATION

In this section, we conduct experiments on real data sets to evaluate the performance of TrajBase.

Algorithm 4: Intersection of encoding lists

Input: $|L|$ encoding lists $L_1 ... L_{|L|}$
Output: Intersection result
1 initiate an empty encoding list $\mathcal{I}$;
2 use min. heap $H_{[1:|L|]}$ to maintain head pairs of all encoding lists $L_1 ... L_{|L|}$;
3 **while** *at least one encoding list not reach end position AND H is not empty* **do**
4 $cid \leftarrow H_{[1]}.cid$, $eid \leftarrow H_{[1]}.eid$, and $counter \leftarrow 1$;
5 remove 1st item $H_{[1]}$ from heap H;
6 move encoding list of $H_{[1]}$ to next position, re-insert the current pair to H;
7 **while** *H contains at least one item AND $H_{[1]}.cid == cid$* **do**
8 $eid \leftarrow eid \sqcap H_{[1]}.eid$;
9 $counter \leftarrow counter + 1$;
10 **if** *$H_{[1]}$ does not reach end position* **then**
11 remove 1st item $H_{[1]}$ from heap H;
12 move $H_{[1]}$ to next position, add the current pair of $H_{[1]}$ to H;
13 **if** *$counter == |L|$ AND $eid > 0$* **then** add the pair $\langle cid, eid\rangle$ to $\mathcal{I}$;
14 **return** $\mathcal{I}$;

Table 1: Datasets Information with 100000 trajectories

	Shanghai	**Zhenjiang**
Total number of GPS points	7,509,760	16,343,629
Max/min/mean num. of GPS points per trajectory	1892/5/75.1	2629/5/163.4
Max/min/mean time interval (Secs)	500/1/45.59	500/1/22.7

5.1 Settings

5.1.1 Datasets andeps Preprocessing

We implemented the core system of TrajBase (including encoding lists and IO library) by C++ and others by Python 2.7. We conduct our experiments on two real GPS trajectory datasets collected in the two cities in China: Shanghai and Zhenjiang (a middle-level city of eastern China). The Shanghai data is generated by 4373 taxicabs from Jan. 31 to Feb. 4 2007, and the Zhenjiang data is generated by 1222 taxicabs from Sept. 1 to Sept. 10 2014 (except without Sept. 7 and Sept. 8). Table 1 summarizes the data sets.

Note that the datasets contain noisy data (caused by GPS error information). Thus, we preprocess the datasets as follows. First, we delete duplicate GPS records. Next, we compute the taxi speed based on every two continuous GPS points. If the speed exceeds 200 meters per second, we treat the latter GPS point as an outlier, and remove these outliers. In addition, we split a trajectory of one taxicab when the timestamp gap of two continuous GPS points is larger than 10 minutes because the taxi is highly possible starting a new trip.

Table 1 indicates that the mean time interval of trajectories in Zhenjiang dataset is shorter than the time interval in Shanghai dataset, while the area of Shanghai is much larger than Zhenjiang. Thus the GPS points in Zhenjiang dataset are much denser than those in Shanghai dataset.

5.2 Query Processing Time

First we are interested in how TrajBase performs in terms of query processing time by comparing four following approaches.

TrajBase: The *TrajBase* uses quadtree and encoding lists to index trajectories. It stores complete trajectories into different data pages on disk. The trajectories are partitioned and retrieved by the methods in section 4.1. After the selection of candidate trajectories, TrajBase directly retrieves the complete candidates from disk and gives the final answer after validation.

Quadtree: This approach only uses quadtree to index trajectories and is derived from traditional trajectory index [4]. Just like [4], it splits trajectories into different sized cells and stores the GPS points of the same cell as a data page on disk. When processing

Table 2: Space cost (MB) and Build Time (Sec)

	Shanghai			Zhenjiang		
	Mem. Cost	Disk Cost	Build Time	Mem. Cost	Disk Cost	Build Time
TrajBase	47.5	373	1948.25	138.04	920	3990.46
Replica-QT	9.4	733	1979.38	11.13	1705	4117.05
Quadtree	30.6	360	1934.24	89.97	785	3906.46
Grid	53.6	373	2618.34	132.26	920	4325.69

a query, this approach reads all data pages of the cells overlapped by query regions. To give the complete trajectories, it will then read the data pages those trajectories passing through again after validation.

Replica-QT: This approach indexes trajectories by quadtree like previous method *Quadtree*. To reduce the number of disk I/O caused by the second time data retrieval from disk like *Quadtree*, we store two versions of trajectories on disk (both complete trajectories like *TrajBase* and trajectory segments which split by cells like *Quadtree*). When processing queries, *Replica-QT* examines the trajectories by those trajectory segments in cells first and then it retrieves final results from disk.

Grid: This approach splits trajectories according to even sized grids like SETI [1] and builds a quadtree and encoding lists to index those grids like *TrajBase*. The trajectories are stored completely on disk like *TrajBase*.

In the following experiments, we use 100000 trajectories with maximum leaf node threshold $M = 4000$, base number $B = 64$ in both Shanghai and Zhenjiang datasets as the parameters for building quadtree and encoding lists. Obviously, the setting of M needs to balance the trade-off between the space cost of the index and efficiency of query processing. Although the space cost of the index will be lower with a larger M because of a lower depth of quadtree, the running time will be sacrificed because the larger cell size will cause higher false positive rate when selecting candidate trajectory. The setting of $M = 4000$ can best balance the trade-off in the both datasets in our experiment. Moreover, to avoid splitting the map too sparse in traffic hub, we set minimum cell size as $0.02km^2$ in both datasets. For grid approach we use cell size approximate to $0.13km^2$ in Shanghai dataset and $0.75km^2$ in Zhenjiang dataset.

Table 2 shows the space and time cost of different index approach using Shanghai and Zhenjiang datasets. TrajBase uses slightly more memory space cost than Replica-QT but much less disk cost. When compared with quadtree itself, TrajBase has more space cost but we will show that it can achieve much faster query processing time very soon.

5.2.1 Number of Regions

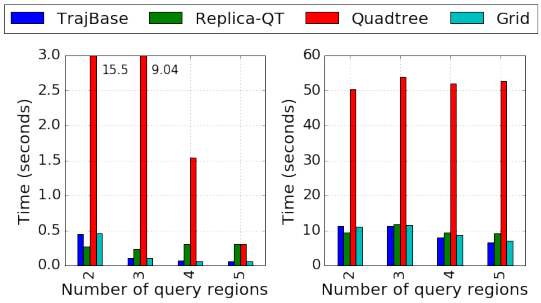


Figure 4: Effect of $|R|$: (a) Shanghai; (b) Zhenjiang

First, we vary the number $|R|$ of regions in query q and Figure 4 shows the average query processing time. In every test we choose regions randomly and set every region to be a square with an extent about $0.1km^2$. Overall, a larger $|R|$ does not necessarily lead to higher running time because few trajectories might satisfy the query q with a larger number $|R|$ regions.

When $|R| = 2$, Replica-QT achieves faster time than TrajBase. However, as shown in Table 4, Replica-QT also uses more disk space. Moreover, when $|R| > 2$, TrajBase has the best performance. Instead, Quadtree has the worst performance in most cases because the trajectory data are split into several data pages on disk and after computing candidate trajectories, it will read all the data pages on disk which the trajectories belongs to to give the final result. Thus, Quadtree costs much more time than other index strategy by combining those split trajectories and incurring high running time. Grid also uses more request time than TrajBase because the cell size is not adaptive for the number of trajectories passed the cell.

5.2.2 Size of Region

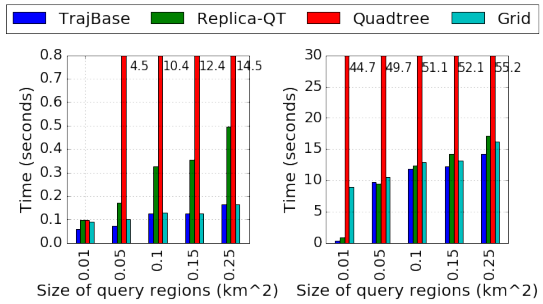


Figure 5: Effect of Size of Region: (a) Shanghai; (b) Zhenjiang

In the second experiment, by fixing the number $|R| = 3$ and instead varying the size of each region from $0.01km^2$ to $0.25km^2$, we plot the running time in Figure 5. Overall, a larger size leads to higher running time for all four approaches. Moreover, our approach TrajBase achieves the fastest running time. The result is consistent with the one in Figure 4.

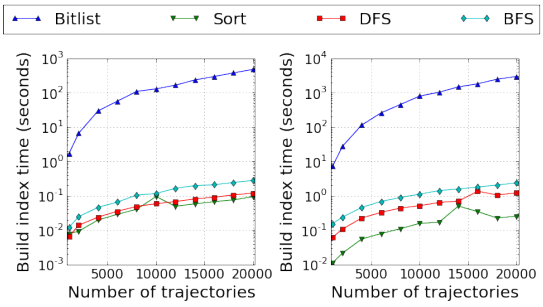
5.3 Encoding Lists

In this section, we are interested in how our approach (i.e., Algorithm 1) to build encoding lists in terms of both running time and compression ratio (measured by the ratio between the space cost of encoding lists and original inverted lists). For comparison, we use four following approaches.

- Random approach: the simplest method to randomly choose columns for re-ordering the matrix.
- BFS approach: Alg. 1 uses the BFS idea to re-order matrix columns.
- DFS approach: We slightly change Alg. 1 by using the uses the DFS (depth-first-search) idea to re-order matrix columns.
- Bitlist approach: Following the previous work [14], we carefully select a column by a top-k query algorithm.

5.3.1 Running Time to Build EncodingLists

First we measure the running time to build encoding lists used by four approaches (we do not plot the time of random approach because its running time is trivial low). In this experiment, we fix



(a) Shanghai (b) Zhenjiang

Figure 6: Time of Building Encoding Lists

the base number $B = 64$ such that each number *eid* encodes 64 trajectories. By varying the number of trajectories, we plot the time in Figure 6. From this figure, we find that the three approaches (*BFS*, *DFS* and *Sort*) use the approximately same time, much less than the one by Bitlist. This is because Bitlist needs to compute the goodness score of a lot of unselected trajectory in every iteration when deciding which trajectory to choose. Thus the complexity is $O(|T|^2)$. Comparatively, *Sort* only need to sort the trajectories with the number of passing cells and the complexity is $O(|T|log|T|)$. Meanwhile, *BFS* and *DFS* only need to traverse the quadtree and the complexity is $O(|Q|)$ where $|Q|$ is the number of nodes in quadtree.

5.3.2 *Number of trajectories*

By varying the number of trajectories, we plot the space cost of encoding lists built by five different approaches in Figure 7. When the number of trajectories grows, the space cost of encoding lists grows, too. Meanwhile, the compression ratio approximately maintain stable. Among the five approaches, *BFS* adopted by our work can achieve much better result than three others (such as *Random*, *DFS* and *Sort*). Note that *BFS* outperforms *DFS* due to the following reason: *BFS* tends to re-assign those trajectory IDs in neighbor cells with continuously re-ordered IDs. More specially, due to the spatial continuation of trajectories, those trajectories which passing the same small cells are more likely to be also passing the nearby bigger cells. In addition, though *Bitlist* achieves slight worse space cost and compression ratio then *BFS*, our previous experiment has already verified that the running time of *BFS* is significantly faster than the one used by *Bistlist*.

5.3.3 *Base B*

Finally, by varying the base number B and the threshold M of trajectory IDs per leaf node, we plot the space size of encoding lists in Figure 8. As shown in this figure, when the threshold M becomes larger, the compression ratio of all algorithms becomes higher. It is because a larger M indicates more trajectory IDs per leaf node and less leaf nodes in quadtree. Thus, the matrix $X_{|N|\times|T|}$ is much smaller and denser and those approaches adopted to build encoding lists can benefit from more options to re-order trajectory IDs, and lead to lower space cost.

In terms of the effect of B, a larger $B = 64$ leads higher space cost than $B = 8$ when the threshold M is very small, such as $M = 100$. It is because there is no good optional re-ordered trajectory ID for better compression result (for example only 1-binary element among 64 elements). As a result, we have to use one 64-bit length integer to encode only 1-element, leading to higher space cost than the cost used by $B = 8$ (using 8-bit length char to encode 1-element). Consistent to Figure 7, this experiment indicates that BFS has the second best compression ratio among all five approaches. However, given a large $M = 2000$, we have much more options to choose the best re-ordered trajectory IDs, such that there are more 1-elements inside an encoded integer. Thus, this experiment further verifies that a larger M is helpful to achieve the best space cost. Nevertheless, a larger typically requires a larger region size and alternatively leads to higher running time.

6. RELATED WORK

There has been plenty of work on trajectory indexing structure. The most classic data structure for trajectory index is R-Tree [7]. It is a height-balanced data structure and each node in it represents a region which is the minimum bounding box (MBB) of all its children nodes. As R-tree allows its children nodes to be overlapped each other, it will cause a large number of overlapping if some regions are visited by too many trajectories, resulting in many I/Os to answer queries. A lot of variations of R-Tree have been studied for indexing spatio-temporal data, including 3D R-Tree [17], TB-Trees [5] and STR-Trees [13]. 3D R-Tree simply modifies R-Tree to model the temporal aspect of the trajectory as an additional dimension. The structure of the TB-tree is a set of leaf nodes, each containing a partial trajectory, organized in a tree hierarchy and connected through a linked list. STR-Tree is an extension of the 3D R- tree to support efficient query processing with different insertion and split strategy. However, all those variations of R-tree can not solve the overlapping problem like R-tree.

Grid-based trajectory index is a trajectory indexing approach differing from R-tree and its variants. In such an index, the spatial extent is partitioned into non-overlapping spatial grids. SETI [1] is the first grid-based index which maintain a logically partition of the spatial extent into a number of non-overlapping spatial grids with the same size. Each cell contains only those trajectory segments that are completely within the cell. Each trajectory segment is stored in data pages of the cell it belongs to. TrajStore [4] is another grid-based trajectory index like SETI. Different from SETI, TrajStore store trajectory segments with quad-tree [6] in different size of grids. It describes an adaptive storage scheme to choose the size of the spatial grids. TrajStore adopts a lossy compression approach on trajectory points. Instead, we adopt an encoding approach on the trajectory index not on raw trajectory points. In addition, Some studies [11] [20] use distributed computing frameworks with conventional trajectory index techniques to process spatial queries and is able to handle more data. Those grid-based index approaches have better scalability, all existing trajectory databases are designed for one region range query and have problems dealing with multi-region range query.

KNN query is another major types of query. KNN queries retrieve the top K trajectories with the minimum aggregate distance to a few points (KNN point query [3], [16], [15]) or a specific trajectory (KNN trajectory query). KNN point query retrieves top-K nearest trajectories when given multiple input points and is more similar to our multi-region range query. Besides, there are some study focusing on similarity search ([19], [18], [2])over trajectory databases that uses a sample query trajectory for full matching according to shape or other criteria. Different from such KNN queries (consisting of multiple input points), we generally allow multiple regions, not just points.

Inverted list have been extensively used in commercial Web engines to index terms of documents and allow fast full text searches. We use this data structure into trajectory index by treating regions as terms and trajectories as text to provide fast multi-regions trajectory search. The space cost of original inverted list will be very

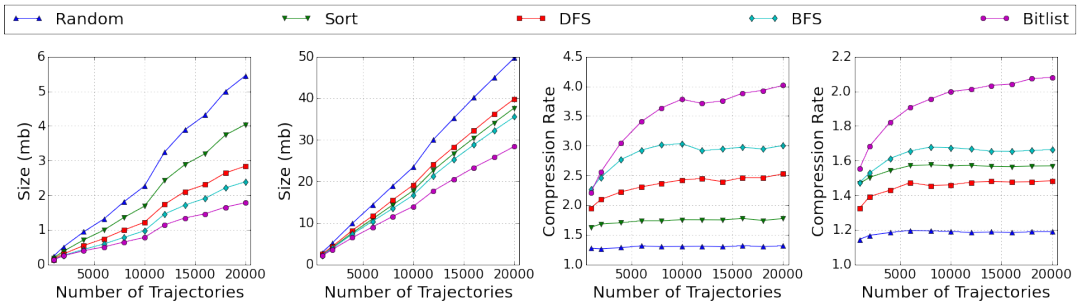
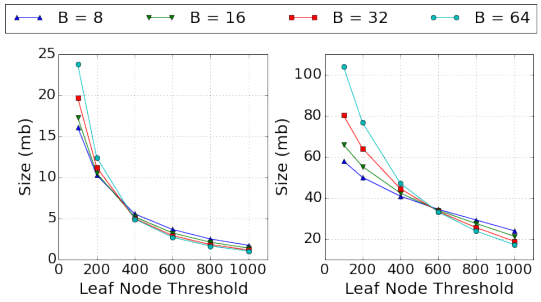


Figure 7: Num. of Trajectories (from left to right): (a-b) space cost on Shanghai and Zhenjiang; (c-d) compression ratio of Shanghai and Zhenjiang



(a) Shanghai (b) Zhenjiang

Figure 8: Size in different Bitlist Base B and Maximum Leaf Node Threshold M with 10000 Trajectories

large when dealing with a huge amount of documents. Most existing inverted list compression techniques ([8] [9])greatly reduce the space cost, but compromise the search performance caused by decompression. Bitlist [14] proposes a full-text indexing structure to achieve space optimization as well as search improvement. However, the main problem of Bitlist is the high running time of building the indexing structure. TrajBase will achieve the same aim with far less time cost by making use of the internal structure of trajectories.

7. CONCLUSION

In this paper, we present a compact trajectory indexing structure TrajBase which to answer the trajectory queries consisting of multiple input regions queries. TrajBase contains two level of structures, a first level classic spatial index (we use quadtree as an example) and a secondary level index called encoding lists. We design an efficient algorithm to build encoding lists with much faster running time than previous work, and meanwhile achieve comparable compression ratio. After that, we describe an algorithm to efficiently find those trajectories going through all multiple input regions. This algorithm can directly perform the queries on top of the encoded encoding lists for much faster running time. Based on two real data sets, we conduct extensive experiments to compare TrajBase with the state of arts. The experimental results verify that Trajbase outperforms previous work in terms of space and running time.

Acknowledgment: This work is partially supported by National Natural Science Foundation of China (Grant No. 61572365) and Science and Technology Commission of Shanghai Municipality (Grant No. 15ZR1443000). We also would like to thank the anonymous reviewers for their valuable comments that helped improve this paper.

8. REFERENCES

[1] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets With SETI. *CIDR*, 2003.

[2] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.

[3] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD*, pages 255–266, 2010.

[4] P. Cudre-Mauroux, E. Wu, and S. Madden. TrajStore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.

[5] C. S. J. D. Pfoser and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In *VLDB*, 2000.

[6] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. In *ACTA*, page 4(1), 1974.

[7] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 14(2):47–57, 1984.

[8] S. D. H. Yan and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, page 401ąğC410, 2009.

[9] X. L. J. Zhang and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, page 387ąğC396, 2008.

[10] D. S. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *VLDB*, pages 13–23, 2004.

[11] K. Lee, R. K. Ganti, M. Srivatsa, and L. Liu. Efficient spatial query processing for big data. In *SIGSPATIAL*, pages 469–472, 2014.

[12] Y. Lim, U. Kang, and C. Faloutsos. SlashBurn: Graph compression and mining beyond caveman communities. *TKDE, volume = 26, number = 12, pages = 3077–3089, year = 2014*.

[13] J. C. T. Y. Pfoser, D. Novel approaches to the indexing of moving object trajectories. In *VLDB*, 2000.

[14] W. Rao, L. Chen, P. Hui, and S. Tarkoma. Bitlist: new full-text index for low space cost and efficient keyword search. In *Proceedings of the VLDB Endowment*, pages 1522–1533. VLDB Endowment, Aug. 2013.

[15] Y. Sun, J. Qi, Y. Zheng, and R. Zhang. K-nearest neighbor temporal aggregate queries. In *EDBT*, pages 493–504, 2015.

[16] L. A. Tang, Y. Zheng, X. Xie, J. Yuan, X. Yu, and J. Han. Retrieving k-nearest neighboring trajectories by a set of point locations. In *SSTD*, pages 223–241, 2011.

[17] M. Vazirgiannis, Y. Theodoridis, and T. K. Sellis. Spatio-temporal composition and indexing for large multimedia applications. *Multimedia System*, 6(4):284–298, 1998.

[18] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[19] B. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.

[20] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In *Eighth International Conference on Grid and Cooperative Computing, GCC 2009, Lanzhou, Gansu, China, August 27-29, 2009*, pages 287–292, 2009.