The VLDB Journal (2016) 25:473–493
DOI 10.1007/s00778-016-0425-6

CrossMark

REGULAR PAPER

# Elite: an elastic infrastructure for big spatiotemporal trajectories

**Xike Xie[1] · Benjin Mei[2] · Jinchuan Chen[3] · Xiaoyong Du[3] · Christian S. Jensen[1]**

Received: 2 July 2015 / Revised: 25 January 2016 / Accepted: 29 January 2016 / Published online: 17 February 2016
© Springer-Verlag Berlin Heidelberg 2016

**Abstract** As the volumes of spatiotemporal trajectory data continue to grow at a rapid pace; a new generation of data management techniques is needed in order to be able to utilize these data to provide a range of data-driven services, including geographic-type services. Key challenges posed by spatiotemporal data include the massive data volumes, the high velocity with which the data are captured, the need for interactive response times, and the inherent inaccuracy of the data. We propose an infrastructure, Elite, that leverages peer-to-peer and parallel computing techniques to address these challenges. The infrastructure offers efficient, parallel update and query processing by organizing the data into a layered index structure that is logically centralized, but physically distributed among computing nodes. The infrastructure is elastic with respect to storage, meaning that it adapts to fluctuations in the storage volume, and with respect to computation, meaning that the degree of parallelism can be adapted to best match the computational requirements. Further, the infrastructure offers advanced functionality, including probabilistic simulations, for contending with the inaccuracy of the underlying data in query processing. Extensive empirical studies offer insight into properties of the infrastructure and indicate that it meets its design goals, thus enabling the effective management of big spatiotemporal data.

**Keywords** Elasticity · Spatiotemporal data · Trajectories

✉ Jinchuan Chen
jcchen@ruc.edu.cn

Xike Xie
xkxie@cs.aau.dk

Benjin Mei
meibenjin@ruc.edu.cn

Xiaoyong Du
duyong@ruc.edu.cn

Christian S. Jensen
csj@cs.aau.dk

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

[2] School of Information, Renmin University of China, Beijing, China

[3] Key Lab of Data Engineering and Knowledge Engineering, Renmin University of China, Beijing, China

## 1 Introduction

The current decade is called the digital universe decade because the digital universe, all data available in digital form, is predicted to grow exponentially during the entire decade. We consider one type of data, namely spatial data and, in particular, spatial-temporal trajectory data. Such data capture the movement of objects, e.g., of individuals, by means of their smartphones or other devices they carry. With the proliferation of location sensing technologies and networking, dramatically increasing volumes of trajectory data are becoming available at a rapid rate.

These data hold the potential to be used in a wide range of services. For example, trajectories capture the time-varying states of a transportation system and of the behaviors of the system users. These data can be used for next-generation, data-driven routing services such as eco-routing and personalized routing [1–3]. And it can be used for urban planning and other smart city purposes. While we consider geographic spatial data, we note that spatiotemporal trajectories also play a role in studies of the human brain in neuroscience [4].

The proliferation of spatiotemporal trajectory data presents us with three key challenges: the data volume, the velocity of the data, including the update rate and the query latency

requirements, and the inherent inaccuracy of the data, often referred to as veracity.

McKinsey reports that the volume of spatiotemporal data from smartphone users is on the order of petabytes per year, and they find that the volume can be 400 times larger if location information inferred using techniques such as station triangulation is included [5]. Likewise, in neuroscience, where a neuron fiber can be modeled as a trajectory, a brain simulation creates petabytes of data [4].

Due to the different location sensing technologies used for obtaining spatiotemporal data, these data are inherently inaccurate and thus come with veracity challenges. Specifically, different positioning technologies such as GPS-based, communication network-based (2G–4G and Wi-Fi based), and proximity-based (e.g., RFID-based) technologies, work differently in different settings and yield data with different accuracies. It is thus generally not possible to obtain accurate spatiotemporal information on moving objects [6–10]. For example, investigators may need to query a large volume of historical trajectories stored in a traffic monitoring system for finding witnesses around the scene of an accident. The relevance of trajectories could be measured by the spatiotemporal closeness, i.e., trajectories within a specified spatial region plus a time interval. Meanwhile, the quality of positioned information needs to be considered for the query evaluation [6–8].

Existing data management techniques fall short in fully addressing these challenges inherent to spatiotemporal data. Specifically, centralized storage and indexing techniques are not well equipped to address the challenges. While a number of distributed storage and indexing techniques have been proposed recently, these offer only partial solutions to the specific challenges of spatiotemporal data [11–16]. Some studies [11–14] propose a master-slave architecture, where the master node serves as a global index that captures the system namespace and the partitioning of the data among slaves. This approach results in a single-node bottleneck that limits index scalability and renders the processing of large current data requests inefficient. RT-CAN [15] and MIDAS [16] aim to enable scalability by applying peer-to-peer overlays. However, the proposed techniques fail to address aspects specific to spatiotemporal data. First, they assume a fixed domain space, while time expands continuously for spatiotemporal data. Second, they do not address complex data types such as trajectories and data inaccuracy. Third, their support for parallelism in query processing can be improved.

To address the challenges of spatiotemporal data, we provide techniques that enable a data management infrastructure that uses a peer-to-peer overlay (CAN [17]) over a cluster of shared-nothing computational nodes, where each such node is a virtual machine. The data are partitioned among nodes based on its spatiotemporal locality. The overlay serves as the global index, and each node has a local index on its data. By utilizing both data locality and parallelism, efficient query processing can be achieved.

The infrastructure includes techniques that model the inaccuracy in the data and enable inference of the confidence in query results, thus enabling trustworthy query results. Parameterized representation of the inaccuracy in the data is used in order to reduce the size and storage costs of inaccuracy descriptions. In query processing, we use sampling techniques that recover the inaccuracies of the data from their parameterized representations and enable confidence calculation. Both inter- and intra-query parallelism are supported in query processing.

The infrastructure matches the available computational resources with the computational needs, thus avoiding cases of over- and under-provisioning. Elasticity is achieved with respect to storage and computation. To achieve storage elasticity, new nodes can be assigned dynamically to active regions where incoming data increase the storage needs; and regions with under-utilized storage can be migrated and condensed to maximize resource utilization. To achieve computational elasticity, the infrastructure is equipped with novel workload estimators that identify a suitable degree of parallelism for querying.

To examine the design properties of the infrastructure, we consider the two arguably most fundamental queries in spatiotemporal databases, the spatiotemporal range query (STRQ) and the spatiotemporal nearest neighbor query (STNNQ). These are used widely and constitute building blocks for many other queries. For example, data-driven routing [18] can be supported by aggregating historical trajectories that pass through both a given origin and destination, and such trajectories can be found by intersecting the results of STRQs around the origin and destination. In the blue brain project [4], neuroscientists also adopt the range query as a building block for advanced data analyses. Likewise, recent work on the earthquake prediction [19] uses the STNNQ as a building block for the clustering of noisy seismic data.

Our contributions can be summarized as follows:

– We investigate a distributed peer-to-peer storage and indexing scheme that is well suited for elastic storage and query processing (Sect. 3).
– We study the evaluation of two fundamental query types, STRQ and STNNQ (Sects. 4.1 and 4.2).
– We present an elastic approach that enables on-demand provisioning of computational resources in order to accelerate query processing (Sect. 4.3).
– We offer empirically based insight into the design properties of the proposed infrastructure by means of experiments with both synthetic and real datasets (Sect. 5).

To the best of our knowledge, this work is the first study on storing and indexing trajectory data in real time and scal-

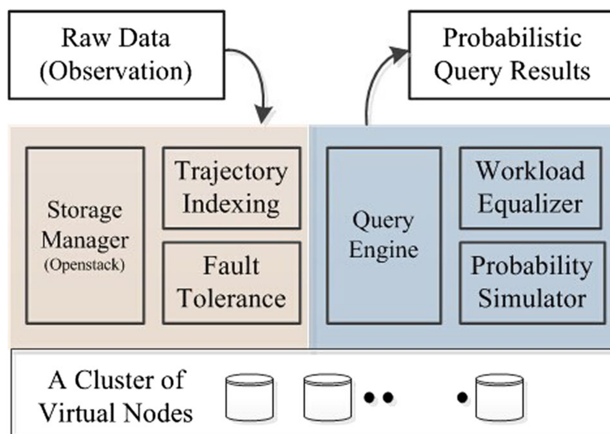


**Fig. 1** Infrastructure overview

ably, and the first elastic solution for handling the veracity challenge associated with the data.

The rest of the paper is organized as follows. As a precursor to the content sections covered above, Sect. 2 makes an overview of our system. Section 6 covers related work, and Sect. 7 concludes the paper.

## 2 Infrastructure overview

Our proposed infrastructure is built on a set of nodes virtualized by OpenStack, as shown in Fig. 1. Each node can be viewed as a computer which is the basic unit in the system providing both storage and computation capabilities. A node can either join or depart from a *system task* according to the system's elasticity mechanism. In particular, our system's tasks are of two categories: *storage* and *computation*.

*The storage task* includes maintaining and indexing observed spatiotemporal information, obtained from a variety of sources. In general, the observed information contains spatial information (represented by coordinates), temporal information (represented by timestamps), as well as object identifiers which string separated records into trajectories.

**Definition 1** (*Observed trajectory*) The observed trajectory $\mathcal{T}_i$ for a moving object $O_i$ is a sequence of observed location records with timestamps $\langle c_i(t_1), c_i(t_2), ...\rangle$, where record $c_i(t_j)$ is a spatial location $c_i$ timestamped with $t_j$, also called the snapshot of $O_i$ at $t_j$. Two consecutive records are connected by a line segment to interpolate the locations in between.

In the assumed use scenario, that the system is subjected to continuously updating spatiotemporal streams of timestamped location records. The storage manager employs a distributed index in order to accommodate a very high rate of incoming records and in order to adapt to the dynamic resource requirements. In addition, data replicas are maintained to enhance the system's fault tolerance.

*The computation task* is handled by the query engine. In addition to retaining basic functionalities of processing accurate data, the query engine can also offer high-quality answers to queries on inaccurate data by capturing the uncertainties associated with the data. To do that, the probability simulator is invoked to render probabilistic information from observed trajectories and accuracy parameters. Queries are then evaluated on the materialized uncertain trajectories, and query results are annotated with their confidences. As shown in the literature [6,20], processing uncertain information often involves considerable computational efforts. To satisfy the performance requirements, the infrastructure uses a workload equalizer to decide how and how much the system's parallel capabilities are to be utilized.

The central data object considered in the paper is that of an *uncertain trajectory*. Considering a snapshot of a moving object, existing proposals [6,21] consider the trajectory uncertainty as an *uncertainty region*, where the exact location is a random variable inside the region. The probability density function (pdf) that captures the distribution of an exact location can be determined or approximated by means of object's velocities, parameters of positioning devices [21], or analysis of historical records. The pdf can be described by either a closed form equation [6,7], or a set of discrete instances [22,23]. In the storage phase, we adopt the continuous form of the uncertainty representation, because it is hard to directly obtain sampling points in real applications, and because the closed form equations are compact, which reduces the storage load requirement. In the query phase, we generate the sampling values based on the pdfs, since the discrete form is good for numerical and parallel computation.

For example, object $O_i$ is observed at time $t$ as a point $c_i$. We model $O_i(t)$ by an uncertainty region $\odot(c_i, \gamma_i)$, which is a circle centered at $c_i(t)$ with radius $\gamma_i$, plus a two-dimensional normal distribution $\mathcal{N}\left(c_i, \frac{\gamma_i}{3}\right)$ restricted to the region. In the query phase, $O_i(t)$'s uncertainty region and pdf are retrieved and sampled into $m$ points, which are represented by $O_i(t) = \{O_i(t)^{(1)}, O_i(t)^{(2)}, ...O_i(t)^{(m)}\}$. Let $Pr(v)$ be the existence probability of a random value $v$. The summation of instances' existence probability satisfying $\sum_{j=1}^{m} Pr(O_i(t)^{(j)}) = 1$, meaning that $O_i$ must exist in its uncertainty region at time $t$.

**Definition 2** (*Uncertain trajectory*) An uncertain trajectory $UT_i$ is a sequence of spatiotemporal instances of object $O_i$ at different times: $\langle O_i(t_0), O_i(t_2), \ldots, O_i(t_n)\rangle$. For a time $t \in [t_j, t_{j+1}]$, $O_i(t)$'s uncertainty is the interpolation between $O_i(t_j)$ and $O_i(t_{j+1})$.

It is challenging to manage and query the large and fast cumulative trajectory data. Next, we introduce the distributed indexing infrastructure that is the core part of the storage

**Table 1** Notation

| Notation | Meaning |
|---|---|
| $\mathfrak{S}, \mathfrak{T}, \mathfrak{S} \times \mathfrak{T}$ | Spatial, temporal, spatiotemporal domain |
| $\{O_1, O_2, ..., O_n\}$ | A set of uncertain objects |
| $O_1(t) = \{O_1(t)^{(1)}, ...O_1(t)^{(m)}\}$ | $O_1$ represented by $m$ samples |
| $\{\mathbb{T}_1, \mathbb{T}_2, ...\mathbb{T}_N\}$ | A set of tori |
| $\mathbb{T}_i = \{T_1, T_2, ...\}$ | $\mathbb{T}_i$ contains a set of torus nodes |
| $\{\mathcal{T}_1, \mathcal{T}_2, ...\mathcal{T}_n\}$ | A set of observed trajectories |
| $\mathcal{T}_i = \langle c_i(t_1), c_i(t_2), ...\rangle$ | An observed trajectory |
| $\{UT_1, UT_2, ..., UT_n\}$ | A set of uncertain trajectories |
| $UT_i = \langle O_i(t_1), O_i(t_2), ...\rangle$ | A recovered uncertain trajectory |
| $\Delta t, [t_s, t_e]$ | A time interval from $t_s$ to $t_e$ |
| $R \oplus r$ | Expand $R$'s spatial region by $r$ |
| $\odot(c, r)$ | Circle with center $c$ and radius $r$ |

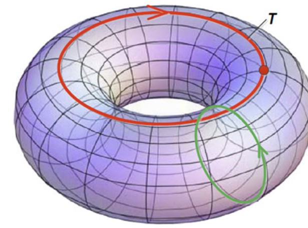


**Fig. 2** Torus

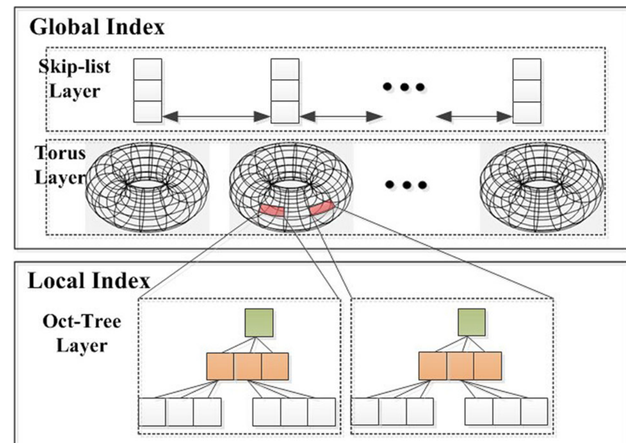


**Fig. 3** Global view of the distributed index

manager, and we further discuss how it offers parallelism for the query engine. Table 1 summarizes the notations used in this paper.

## 3 Indexing infrastructure

In this section, we introduce the proposed storage and indexing infrastructure. Section 3.1 serves as a roadmap. Logically, the indexing infrastructure consists of three layers, a skip-list layer (Sect. 3.2), a torus layer (Sect. 3.3), and an oct-tree layer (Sect. 3.4). In Sect. 3.5, we cover the dynamic operations that are important for storage elasticity. Finally, other system features, such as storage load balancing and fault tolerance, are covered in Sect. 3.6.

### 3.1 Infrastructure overview

Physically, our system consists of a set of shared-nothing nodes. Each node is an independent computational unit, i.e., a computer, whose data range correspond to a fragment of the spatiotemporal domain of the data. Nodes are clustered into a structure called the torus according to the proximity of their spatiotemporal ranges in order to achieve data locality. A torus abstracts the topological features of the peer-to-peer protocol CAN [17]. We show an example of a 2D torus in Fig. 2. In spatiotemporal applications, the spatial domain $\mathfrak{S}$ is often fixed, while temporal domain $\mathfrak{T}$ expands over time. Thus, it is reasonable to define a torus $\mathbb{T}$'s domain as $\mathfrak{S} \times \Delta t$, where $\Delta t \subseteq \mathfrak{T}$ is $\mathbb{T}$'s time domain.

The three logical layers of the system are shown in Fig. 3. The oct-tree layer refers to the local indexes, typically oct-trees, that each node uses for indexing its local data. The two upper layers, the skip-list layer and the torus layer, form the global part of the system, the system's global index. The global index serves as the communication channel between the different nodes. The communication occurs at two levels, *inter-* and *intra-*torus communication, which are handled by the skip-list layer and the torus layer, respectively. The three layers are distributed across all nodes in the system. In the infrastructure, there is no privileged nodes and each node is an equipotent peer. Data access requests through the global index to reach relevant nodes. Then, corresponding local indexes are traversed. In case relevant data are found at multiple nodes, the data are combined and refined. Next, we consider each component in detail (Fig. 3).

### 3.2 Skip-list layer

The skip-list layer connects different tori in the temporal domain. The temporal domain $\mathfrak{T}$ is partitioned into a set of time intervals, $\mathfrak{T} = \{\Delta t_0, \Delta t_1, \ldots\}$ meaning that $\Delta t_i \cap \Delta t_j = \emptyset$ if $i \neq j$ and $\cup_i \Delta t_i = \mathfrak{T}$, and each torus corresponds to a time interval. Tori are chained. An 2D example of a torus chain is shown in Fig. 4. Further, to support fast search on the temporal dimension, we use a double linked skip list [24]. Next, each torus cluster corresponds to a node in the skip list. For a skip-list node, the key is the temporal interval of the torus cluster, and the pointer to it is a preassigned consecutive ip address segment. Within a torus, the skip-list information is shared among all nodes.

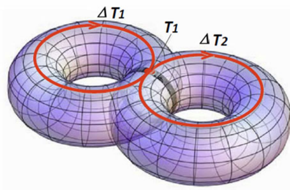


**Fig. 4** Torus chain

For example, suppose tori $\mathbb{T}_1$ and $\mathbb{T}_2$ are directly linked in the skip-list layer. It means that torus node $A \in \mathbb{T}_1$ stores the ip address segment of $\mathbb{T}_2$. For routing targeted at $\mathbb{T}_2$, A randomly picks an ip address from the $\mathbb{T}_2$'s ip address segment, which points to a torus node $B \in \mathbb{T}_2$. After that, intra-torus routing starts in $\mathbb{T}_2$. Since B is randomly chosen, the workload on the skip-list layer is evenly distributed among the nodes in a torus cluster.

*Analysis* If $k$ tori are indexed by the skip list, it costs $O(logk)$ hops to reach a torus. Each torus takes $O(logk)$ to store the skip-list node.

### 3.3 Torus layer

In the system, a torus corresponds to a cluster of nodes. Each node is an independent computational unit (a physical computer or virtual machine). The torus structure is the core part of the index. We use a torus because: (1) it supports multidimensional data; (2) it preserves the elasticity of peer-to-peer techniques (CAN [17]), where a node can easily depart from or enter the system. Correspondingly, the intra-torus communication between different nodes follows the CAN [17] routing scheme. To achieve that, each node stores a routing table about its neighboring nodes. The routing table has two major columns: the ip address of a neighbor and the data range of the neighbor. Initially, the number of neighbors is set to 6, since there are three dimensions, and each dimension has two directions. Thus, the cardinality of the routing table is 6. Notice that the topology of a 3D torus is different from that of a cube structure. On a torus, nodes on a dimension are strung together end-to-end in a ring structure, whereas on a cube, they are connected by a line, meaning that the head and tail nodes do not connect.

The routing within a torus is done as follows. Let nodes A and B be in the same torus. $A$ gets a request $q$ to access data in $B$. $A$ handles the routing by seeking in its routing table to find the neighbor whose data range is closest to $q$. If multiple neighbors are equally close to $q$, A randomly selects a neighbor, since the expected number of hops to reach B is the same for each candidate. Then, on $A$'s neighbor node, the above process is repeated until $B$ is reached. The detailed routing process is described by Algorithm 5 in the "Appendix". If

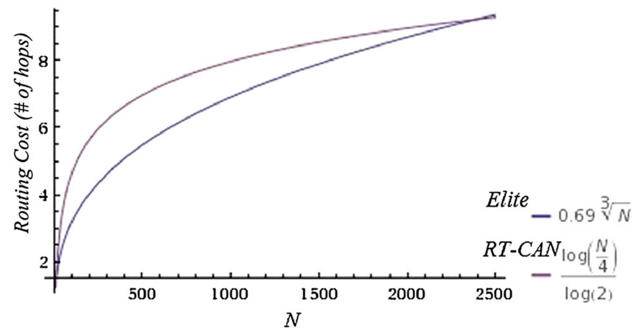


**Fig. 5** RT-CAN versus Elite (routing cost)

$A$ and $B$ are located in different tori, routing on the skip-list layer occurs first. Based on the routing scheme, we can easily design the range query that is the basic operator for data access in a torus. Given a spatiotemporal query region $q$, i.e., a 3D rectangle, the range query finds all relevant nodes and forwards the search to local indexes for further processing, as detailed in Algorithm 4 in the "Appendix".

*Analysis* The expected routing cost of a torus containing $N$ nodes is $0.69\sqrt[3]{N}$ hops (Lemma 2), and the maximum routing cost is $0.91\sqrt[3]{N}$ hops (Lemma 1, also in the "Appendix"). If extra links are built, as RT-CAN [15] does, the average routing cost can be reduced to logarithmic forms, e.g., $\log\frac{N}{4}$ for intra-torus routing and $\log\frac{kN}{4}$ for routing over the entire infrastructure. We choose to follow the simple but powerful CAN protocol [17]. We plot the routing costs of RT-CAN and our proposal (Elite) in Fig. 5, ignoring their common part $O(logk)$. RT-CAN starts to perform better only when the size of a single torus cluster is extremely large ($\geq$2500 nodes). We expect such cases to be rare. Also, the simple structure has smaller update and storage costs (see Table 2) and thus retains the ability for light weight self-adjustment in dynamic scenarios.

The routing cost can be used to estimate the system throughput for storage tasks. Suppose the storage workload is uniformly distributed among different peers, the throughput of a peer-to-peer scheme can be estimated as the total bandwidth of the system divided by the number of messages that a task takes. If the throughput of a fully packed computer is 1, the throughput of RT-CAN and Elite can be upper bounded by the quotient of $kN$ divided by their routing costs, respectively (see Table 2).

### 3.4 Oct-tree layer

The local index contains an oct-tree and a hash table. An oct-tree is a three-dimensional version of a quad-tree and is used to store the observed locations of a trajectory. Each stored location has a pointer to the its successor location. Then, operations on a trajectory are similar to those on a linked

**Table 2** Complexity comparison

| | Centralized index | RT-CAN [15] | Elite (skip-list linked CAN [17]) |
|---|---|---|---|
| Routing | O(logk + logN) | log(kN/4) | $O(\log k) + 0.69\sqrt[3]{N}$ |
| Throughput | 1 | $kN/\log(kN/4)$ | $\frac{kN}{O(\log) + 0.69\sqrt[3]{N}} \approx 1.45kN^{2/3}$ |
| Updating cost | O(logk + logN) | O(logk + log N) | O(1) (intra-torus update) or O(logk) (inter-torus update) |
| Extra storage cost | 0 | O(logk + logN) | O(logk) |

list. The data format of each location record is a five-element tuple (TrajID, Obs_Pt, Next_Pt, Next_ptr, UPara). Here, TrajID identifies a trajectory uniquely. Then, Obs_Pt stores the trajectory's observed location, and Next_Pt stores the successor's location. The two points form a line segment that work as the basic unit in comparison with query ranges. All interpolated points (on the line segment) have to be considered during query processing to avoid false negatives. Next_ptr identifies the location record of the next location of the trajectory. Then, the observed trajectory can be traversed by starting at the trajectory's head then following the successor pointers. The UPara field stores information for describing the uncertainty associated with the trajectory, including an uncertainty region size and a probability distribution for each observed position.

The hash table is used to map a trajectory's ID to its observed locations. In particular, it stores pointers to trajectories' heads and tails. The tail is the latest observation of a trajectory in the torus, and the head is the first observation. Since we assume that the trajectory data arrive in chronological order, the tail pointer is used for efficient insertion. The root node of oct-tree represents the entire domain of the torus node. Each internal node has eight children, each of which corresponds to 1/8 of its parent's range, as depicted in Fig. 6a. The trajectory data are inserted into an oct-tree leaf node, if the node and the data overlap. In case the leaf node is full, i.e., the capacity exceeds 4Kb, it is split into eight new nodes, from octant 1 to 8 in Fig. 6a.

Given the structure above, a local range search $Q$, the query first finds the leaf nodes overlapping with its range; then it descends the index to find the line segments overlapping with its range. Finally, the trajectory IDs of overlapping segments are retrieved.

*Discussion* We use the oct-tree [25] instead of the frequently used R-tree because of three reasons. First, the oct-tree is more consistent with the space decomposition approach of the torus layer. Second, it is straightforward and simple to apply to oct-tree to index trajectory data. Existing R-tree variants for trajectory data, e.g., the 3D-Rtree, the TB-tree, and the MV3-RTree, target disk-based indexing of disk-resident data and thus are either not well suited for in-memory settings or would benefit from improvements [26,27]. Third, the performance of local indexes is a secondary issue for the system. The performance bottleneck is found in the networking part, and the local indexes do not limit the performance.

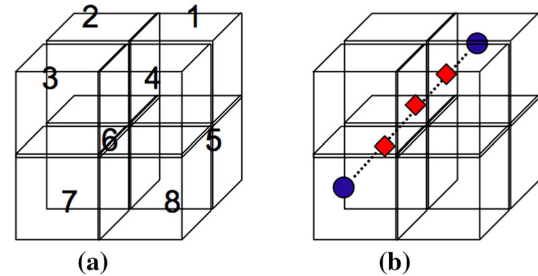


**Fig. 6** Oct-tree node splitting

### 3.5 Dynamic operations

The infrastructure achieves storage elasticity through dynamically allocating and recycling storage loads. We cover dynamic operations on the skip-list layer in Sect. 3.5.1 and operations on the torus layer in Sect. 3.5.2.

#### *3.5.1 Dynamic skip-list operations*

If a new torus is to be appended to the system, this is broadcast to all torus clusters by reporting the new torus's address segment and temporal range. Upon receiving the report, each skip-list node establishes the double links with some probability [24]. We set the probability to be 0.5. Then, the torus cluster with a positive decision sends its ip address segment and data range back to the new torus. This results in a total of $k + \log k$ hops between torus clusters. On each torus cluster, the time complexity is $O(logk)$.

#### *3.5.2 Dynamic torus operations*

The system has two types of dynamic operations at the torus level: splitting and condensing.

*Torus node splitting* We set a maximum capacity for each torus node. If the storage on torus node A exceeds the capacity, the system allocates a new torus node B to share the storage load. Logically, the torus node A is split into two

nodes, A and B, along a specified splitting dimension $d$ for better storage load balancing. We choose the splitting dimension $d$ which results in the least difference between the storage load of the new A and B.

Suppose A's region $R$ is split into two equally volumed regions $R_d^+$ and $R_d^-$ along dimension $d$. After the splitting, A's value region is set to $R_d^+$. Meanwhile, a new torus node B is appended whose value region is set to $R_d^-$.

There are four types of information to be migrated from A to B: the trajectory data overlapping with $R_d^-$, the index structure for $R_d^-$, the auxiliary hash table for $R_d^-$, and the updated routing table containing the new neighbor information. The index structure is easy and efficient to handle, as the oct-tree itself is constructed through splitting of the domain space. Since a torus node is split into two equal volume nodes, a torus node is split along the splitting surface of the oct-tree root node. In other words, the process does not incur any oct-tree node splitting.

To get the trajectory and the hash table, we need to traverse every trajectory from head to tail. During the traversal, we (1) determine which line segment belongs to $R_d^-$; (2) if a line segment spans over A and B, the line segment breaks at the intersection with the splitting dimension; (3) construct the hash table for B. The routing tables of A and B are updated accordingly. Then, the retrieved information is packed and sent to B.

*Torus node condensing* In case a torus node A is under-utilized, say below a minimum capacity,[1] the system can migrate the information of the node to another under-utilized torus node B. Logically, the two under-utilized nodes A and B are merged. To implement that, the system: (1) informs A's neighbor to change the routing table with the address of B; (2) migrates the index, data, and routing table; (3) recycles node A. Notice that now B has two data ranges and routing tables. The routing package is analyzed to determine whether it is sent from B's neighbors or from neighbors of the old A. The service can be triggered either manually or automatically.

### *3.5.3 Oct-tree node splitting*

When a leaf node reaches its capacity, it is split into eight octants by splitting along three dimensions. The trajectories associated with the leaf node are then placed in the appropriate new nodes. During the splitting, we need to interpolate some points on the splitting dimensions and include them in the trajectory. Otherwise, false negatives occur in the query processing. For example, the range query $Q$ in Fig. 7 will miss trajectory $a \rightarrow b$, if points $x_1$ and $x_2$ are not interpolated and stored in octant 2.

[1] The minimum capacity is below half of the maximum capacity. Otherwise, the condensed node may exceed the maximum capacity.

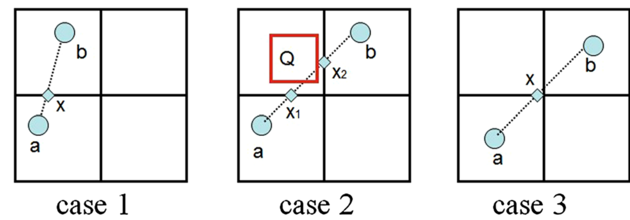


**Fig. 7** Three cases for oct-tree node splitting

There are four cases to consider according to the relative position of $a$ and $b$. If $a$ and $b$ are on the diagonal corners, it is case 0, shown in Fig. 6b. Otherwise, they are on the same side of a splitting dimension, as shown in cases 1–3 in Fig. 7.

If a and b are separated only by one splitting dimension, we can calculate the intersection $x$ of the line segment $\overline{ab}$ and the splitting dimension (case 1 in Fig. 7). If a and b are separated by two splitting dimensions (both vertically and horizontally as in case 2 in Fig. 7), we can calculate two intersections $x_1$ and $x_2$. A special case occurs when $x_1$ and $x_2$ degenerate into one point $x$, as shown in case 3 in Fig. 7). In our work, we use a threshold $\epsilon$, such that if the distance between $x_1$ and $x_2$ is smaller than $\epsilon$, we use the center point to approximate the intersection (case 3 in Fig. 7). The last case occurs when $a$ and $b$ are in the diagonal corners, meaning that they are separated by all three splitting dimensions, as depicted in Fig. 6b. This case can be transformed into the cases in Fig. 7 by arbitrarily selecting a splitting dimension to decompose the line segment into two parts. Each part is then covered by one of the cases 1–3.

### 3.6 Discussion on the torus structure

*Storage load balancing* Storage load balancing is achieved by integrating with the OpenStack cloud platform that adopts a scheduler to determine which physical node a virtual node should be launched on. By default, the scheduler will randomly select a physical node for a virtual node allocation request in order to evenly distribute the virtual nodes among all physical nodes. The random strategy alleviates the problem of data skew. In real applications, it is likely that only a fraction of the data are being accessed frequently. If such "hot" data are divided among multiple physical nodes, the corresponding operations over the data will also be more evenly distributed. Another level of storage load balancing is achieved by condensing virtual nodes. The system is able to periodically merge virtual nodes that underflow in order to better utilize computational resources.

*Fault tolerance* We implement a synchronization strategy for a torus node to replicate the data it owns (called *a primary copy*) onto two neighboring nodes (called *repli-*

*cas*). The replicas help increasing the data's resilience and potentially enhance query performance. In our system, the synchronization can occur in two ways: (1) transactional replication, which mirrors the updates to the replicas simultaneously with updates to the primary copies; (2) periodical replication, which propagates batched updates of primary copies periodically. Then, the recovery process consists of three steps. Suppose a torus node, $T_1$, fails. First, another node, $T_{new}$, is initialized and appended to the torus cluster. Second, a replica of the data in $T_1$, which is kept in a neighboring node of $T_1$, is sent to $T_{new}$. Third, $T_{new}$ loads the data and reconstructs the local index.

*Locking mechanism* The system handles updates by implementing read–write locks on the index. If write–write or read–write conflict occurs, the index (or, equivalently, the root node) is locked until the task that holds the lock finishes. Opportunities exist for optimizing the locking granularity, i.e., from locking the root node to locking child nodes. However, studies of locking granularities are orthogonal to this work.

*Potential routing optimization* There exist rooms to further optimize the routing and indexing schemes considered in the work.

*Inter-torus routing optimization* In a torus chain, there exist some torus nodes which have adjacent domains but are allocated to different tori. For example, as shown in Fig. 4, tori $\Delta\mathbb{T}_1$ and $\Delta\mathbb{T}_2$ overlap at a slice $T_1$ highlighted in black. The torus nodes that touches the slice can be interconnected so that the routing across the slice can be done without the skip list.

*Intra-torus routing optimization* It is possible to reduce the intra-torus routing cost by accommodating trajectories in a $R^+$-tree manner. For example, if a trajectory spans over multiple torus nodes, we maintain the copy of the whole trajectory for those nodes. Then, if a trajectory is to be accessed, a local node instead of multiple nodes is accessed and thus reduces the intra-torus communication. Actually, it has been partially achieved in the replica mechanism presented in the work.

In this work, we do not go deeper in these directions because the routing cost is not the dominant part of the query evaluation.

## 4 Query engine

In this section, we describe the query engine. We first define two representative spatiotemporal queries, STRQ and STNNQ (Sect. 4.1). Then, we describe how the parallelism is used for processing these queries efficiently (Sect. 4.2). Finally, we describe how the parallelism can be used for processing queries elastically (Sect. 4.3).

### 4.1 Query semantics

The query engine supports data veracity by applying uncertainty models to the observed trajectory data. A query then returns possible answers together with their qualification probabilities, indicating the confidence of the result.

#### 4.1.1 Spatiotemporal range queries

**Definition 3** (*Spatiotemporal range query* (STRQ)) given a three-dimensional query region $Q$, and an uncertain trajectory database $D$, STRQ($Q$, $D$) retrieves all uncertain trajectories $\{UT_i\}$ that belong to $Q$ with nonzero probability. Formally,

$$STRQ(Q, D) = \{(UT_i, qp^R(Q, UT_i)) \mid qp^R(Q, UT_i) > 0 \wedge UT_i \in D\}$$

**Definition 4** The *qualification probability* ($qp^R$) of an uncertain trajectory $UT_i$ satisfying STRQ($Q$, $D$) is:

$$qp^R(Q, UT_i) = 1 - \prod_{t=1}^{|UT_i \cdot \Delta t \cap Q \cdot \Delta t|} F(Q, UT_i, t), \qquad (1)$$

where

$$F(Q, UT_i, t) = \sum_{\{j \mid O_i^{(j)}(t) \notin Q\}} Pr(O_i^{(j)}(t))$$

Here, $F(Q, UT_i, t)$ is the probability that $UT_i$ does not satisfy $Q$ at time point $t$. Hence, the second term of Eq. 1 calculates the probability that $UT_i$ cannot appear in $Q$ at any time. To obtain $F(Q, UT_i, t)$, we sum up the existence probabilities of $O_i$'s samples that located outside $Q$.

#### 4.1.2 Spatiotemporal nearest neighbor queries

**Definition 5** (*Spatiotemporal nearest neighbor query* (*STNNQ*)) given a query $Q = (q, \Delta t)$ consisting of a spatial point location $q$ and a time interval $\Delta t$, and an uncertain trajectory database $D$, STNNQ($Q$, $D$) retrieves all uncertain trajectories $\{UT_i\}$ in $D$ that can be $q$'s nearest neighbors during $\Delta t$ with nonzero probability. Formally,

$$STNNQ(Q, D) = \left\{(UT_i, qp^N(Q, UT_i)) \mid qp^N(Q, UT_i) > 0 \wedge UT_i \in D\right\}$$

**Definition 6** The *qualification probability* ($qp^N$) of an uncertain trajectory $UT_i$ satisfying STNNQ($Q$, $D$) is:

$$qp^N(Q, UT_i) = 1 - \prod_{t=1}^{|UT_i \cdot \Delta t \cap Q \cdot \Delta t|} F(q, UT_i, t), \quad (2)$$

where

$$F(q, UT_i, t) = 1 - \sum_{\{j|O_i^{(j)}(t)\}} Pr(O_i^{(j)}(t)) \prod_{g \neq i \wedge O_g \in D} Pr(|q, O_i^{(j)}(t)| < |q, O_g(t)|)$$

Equation 2 is similar to Eq. 1, where $F(q, UT_i, t)$ is the probability that $UT_i$ is not the nearest neighbor of $q$ at time point $t$. The value of $F(q, UT_i, t)$ is calculated by one minus the probability that $UT_i$ is the nearest neighbor of $q$. Note that at time point $t$, $UT_i$ will be the nearest neighbor of $q$ if $O_i$ appears at $O_i^{(j)}(t)$ and the distances between all other objects and $q$ exceed $|q, O_i^{(j)}(t)|$.

### 4.2 Evaluating spatiotemporal queries

We proceed to describe how to efficiently compute the STRQ and STNNQ queries. Query evaluation proceeds in three phases. The first phase, *filtering*, utilizes the distributed index (Sect. 3) to locate the torus nodes that overlap with the query region in order to retrieve candidate trajectories by accessing local indexes. The second phase, *node allocation*, allocates idle torus nodes for the refinement phase. In the third phase, *refinement*, we use sampling methods to generate a set of possible instances, simulate the imprecision of trajectories, and calculate the corresponding qualification probabilities. The results are then merged and returned.

#### *4.2.1 Spatiotemporal range query*

The evaluation of STRQ is formalized in Algorithm 1. In the filtering step, the query is forwarded to torus nodes with regions that overlap with the query range $Q$. For each node $T_i$, the intersection with $Q$ is $Q_i$. The query $Q$ is thus decomposed into a set of sub-queries that can be run on different nodes in parallel. Let us consider $T_i$ with $Q_i$. The local index is traversed to retrieve trajectories overlapping with $Q_i \oplus U_{max}$, where $U_{max}$ is the size of the uncertainty region, and "$\oplus$" is a binary operator that extends the spatial region of the left operand with a length indicated by the right operand. Formally,

$$\oplus : (\mathfrak{S} \times \mathfrak{T}) \times N \rightarrow (\mathfrak{S} \times \mathfrak{T})$$

For example, given a region $R = ([x_{low}, x_{high}], [y_{low}, y_{high}], [t_{low}, t_{high}])$ and a length $r$, $R \oplus r$ is the region $R' = ([x_{low} - r, x_{high} + r], [y_{low} - r, y_{high} + r], [t_{low}, t_{high}])$. In the torus node $T_i$, if an uncertain trajectory does not intersect $Q_i \oplus U_{max}$, it cannot to be in the region $Q$. Based on the number of candidates retrieved, we can estimate the refinement cost by equation $F^R$, to be detailed in Sect. 4.3.2.

**Algorithm 1** Spatiotemporal Range Query

```
1: function STRQ(query Q)
2:    Find all torus nodes {T_i} overlapping with Q   ▷ Step 1. Filtering
3:    Q_i is the intersection between Q and the region of T_i
4:    In parallel for each T_i do
5:       C_i ← LocalOctTreeRangeSearch(UT, Q_i ⊕ U_max)
6:                ▷ C_i ={ UT } is a set of candidate trajectories
7:                ▷ U_max is the largest uncertainty region size
8:       n# ← F^N(|C_i|)      ▷ n# is the number of torus nodes requested
9:       {T_j} ← nalloc(n#)          ▷ Step 2. Node Allocation
10:       Decompose Q_i into n# sub-queries {Q_i^(j)}
11:       for each sub-query Q_i^(j) do
12:          Find its corresponding candidates C_i^(j)   ▷ C_i^(j) ⊆ C_i
13:          Deliver (Q_i^(j), C_i^(j)) to corresponding node T_j
14:       In parallel for each T ∈ {T_j} do         ▷ Step 3. Refinement
15:          for each trajectory UT ∈ C_i^(j) do
16:             Calculate qp^R(Q_i^(j), UT)
17:    Merge the results of {Q_i^(j)} and return to the query issuer.
```

The second phase is node allocation (Nalloc). Upon receiving the Nalloc task, a representative torus node, called **node*,[2] allocates a number of idle nodes to $T_i$ for the refinement phase. Query $Q_i$ together with the candidates obtained beforehand is distributed evenly among the assigned nodes. Refinement is then done in parallel in order to meet response-time requirements. Details about Nalloc and the cost estimation function $F^R$ are provided in Sects. 4.3.1 and 4.3.2, respectively. In the end, the query results are collected and returned to the user.

#### *4.2.2 Spatiotemporal nearest neighbor query*

The evaluation of STNNQ is detailed in Algorithm 2. In the filtering step, the query retrieves in parallel all candidates that can be the nearest neighbor of $q \in Q$. For each torus node $T_i$ intersecting with $Q$, assume $Q_i$ is the intersection part. $Q$ is decomposed into a set of $\{Q_i\}$, each of which is run independently in the corresponding torus node $T_i$. $T_i$ issues a range query parameterized by $(Q_i \oplus d_{max}) \oplus U_{max}$ to filter objects outside the range, as they have no chance to be STNNQ answers. Here, $d_{max}$ is equal to $max_{t \in Q_i \cdot \Delta t}\, min_{\forall UT \in T_i}(|O_i(t), q|)$. The process of obtaining $d_{max}$ is discussed in "Appendix 4". Notice that the filtering

2 Details on the *node are covered in Sect. 4.3.1.

**Algorithm 2** Spatiotemporal Nearest Neighbor Query

1: **function** STNNQ(query $Q < q(x, y), \Delta t(t_s, t_e) >$)
2: Find all torus nodes $\{T_i\}$ overlapping with $Q$ ▷ **Step 1.** Filtering
3: $Q_i$ is the intersection between $Q$ and $T_i$'s region
4: **In parallel** For each $T_i$ **do**
5: Obtain $d_{max}$ ▷ "Appendix 4"
6: $R_i \leftarrow Q_i \oplus d_{max} \oplus U_{max}$
7: $C_i \leftarrow RangeQuery(R_i, T_i)$ ▷ $C_i = \{UT\}$ is a set of candidate trajectories
8: $n^\# \leftarrow F^N(C_i)$ ▷ $n^\#$ is the number of nodes requested
9: $\{T_j\} \leftarrow nalloc(n^\#)$ ▷ **Step 2.** Node Allocation
10: Decompose $Q_i$ into $n^\#$ sub-queries $\{Q_i^{(j)}\}$
11: **for** each sub-query $Q_i^{(j)}$ **do**
12: Find its corresponding candidates $C_i^{(j)}$ ▷ $C_i^{(j)} \subseteq C_i$
13: Deliver $(Q_i^{(j)}, C_j)$ to corresponding node $T_j$
14: **In parallel** for each $T \in \{T_j\}$ **do** ▷ **Step 3.** Refinement
15: **for** each trajectory $UT \in C_i^{(j)}$ **do**
16: Calculate $qp^N(Q_i^{(j)}, UT)$
17: Merge the results of $\{Q_i^{(j)}\}$ and return to the query issuer.

range might span multiple torus nodes. The qualified trajectories are shipped back to $T_i$. We then estimate the refinement cost by function $F^N$, to be detailed in Sect. 4.3.2. The allocation and refinement phases are similar to those of STRQ.

*Discussion* Note that the distributed refinement workloads are run independently over different torus nodes. Suppose $UT_i \cdot \Delta t$ is partitioned into several time intervals, i.e., $UT_i \cdot \Delta t = \{\Delta t_1, ..., \Delta t_s\}$, each of which corresponds to an allocated torus node. The qualification probability equation of STRQ (Eq. 1) can be rewritten as follows.

$$qp^R(Q, UT_i) = 1 - \prod_{t=1}^{|\Delta t_1 \cap Q \cdot \Delta t|} F(Q, UT_i, t) \times \cdots \times \prod_{t=1}^{|\Delta t_s \cap Q \cdot \Delta t|} F(Q, UT_i, t) \quad (3)$$

Thus, each $\prod_{t=1}^{|\Delta t_l \cap Q \cdot \Delta t|} F(Q, UT_i, t)$ $(l = 1, \cdots, s)$ can be computed independently by a torus node allocated in the Nalloc phase, and the final result can be obtained easily by combining the local results according to Eq. 3. The distributed calculation of the qualification probabilities for STNNQ queries can be handled in a similar way.

### 4.3 Elastic query evalaluation

Our system achieves computational elasticity by allocating dynamic computational resources on demand. To explain how it works, we first cover the mechanism for idle node allocation (Nalloc), which fetches computational resources upon requests, in Sect. 4.3.1. We describe the workload estimator in Sect. 4.3.2.

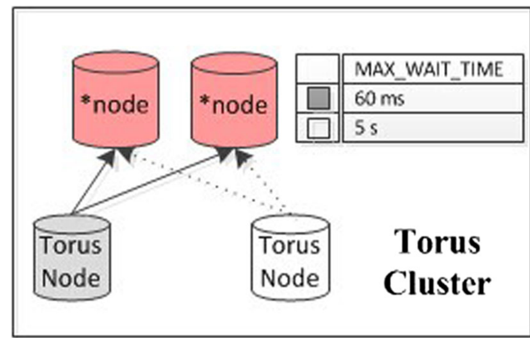


**Fig. 8** Heart beats

#### 4.3.1 Implementation of Nalloc

The idea of the mechanism that provides elasticity is to collect statistics from nodes periodically. Then, computational resources are allocated according to the currently available nodes and workload estimators.

*Heart Beats* The system monitors the query workload in real time by having torus nodes report their current workload periodically to a representative torus node, called *node (see Fig. 8). The *node is selected among torus nodes with light workloads. Initially, a random node is selected as *node as they are equally idle. In our implementation, the heart beat frequency is set to be 1 HZ. The system has a threshold to indicate whether a node is idle or busy. The *node maintains a table of currently idle nodes in the cluster. Our system allows multiple *nodes. Then, the idle nodes in the table are also shuffled at each heart beat to alleviate the problem of allocating the same idle node to multiple tasks. The table is checked when allocating nodes. A by-product of the heart beats is to monitor whether to replace abnormal torus nodes.

*Idle nodes retrieval* Based on the workload estimation, torus node $T_i$ gets the number of nodes required for the local refinement task. Before sending the request, the torus node checks itself to see if it is idle. If no, the refinement task is forwarded to the *node for allocation of idle nodes. Otherwise, the requested number is decreased by one and delivered to the *node. In case of multiple *nodes in the cluster, the request is forwarded to a random one. Based on the statistics collected at each heart beat, the *node tries to match the request with the currently idle nodes. Other torus clusters are visited if too few idle nodes are found. The request is forwarded to *nodes of other such clusters through the skip-list layer. Due to its special role and to ensure the smooth operation of monitoring and scheduling, the *node is not allocated to refinement tasks. The process is illustrated in Fig. 9.

#### 4.3.2 Workload estimator

Now, we consider cost estimation for both STRQ and STNNQ. According to Algorithm 1, for STRQ, the refine-

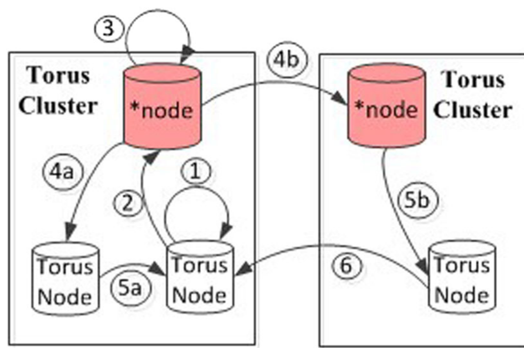


**Fig. 9** Find the idle node

ment phase contains two parts: (1) generate sampling points; (2) check whether the sampling points overlap $Q$. Let $\alpha$ and $\beta$ be the cost per sampling point for the two parts, respectively. The refinement cost can be estimated by $F^R$ as follows, where $n$ is the number of total sampling points.

$$F^R(Q) = \alpha n + \beta n \tag{4}$$

Similarly, based on Algorithm 2, for STNNQ, the refinement phase contains three steps: (1) generate sampling points; (2) pruning non-qualifying points; (3) calculate the qualification probabilities. Let $\alpha$, $\beta$, and $\gamma$ [3] be the cost per sampling point for these three steps. Then, the refinement cost can be estimated by $F^N$ as shown below, where $h$ is the number of sampling points after pruning ($h \leq n$).

$$F^N(Q) = f^N(Q_{snapshot}) \cdot Q_N \cdot \Delta t,$$
$$\text{where } f^N(Q_{snapshot}) = \alpha n + \beta n + \gamma(h^2/2)$$

Here, $f^N$ is the cost for each sampled snapshot on the time dimension. Note that the cost of calculating probabilities is proportional to $h^2/2$ using the method in our previous work [28]. The value of $n$ can be collected after the filtering step. The value of $h$ for STNNQ can be estimated as detailed in Appendix 1. Then, the number of nodes required for the computation can be calculated by Eq. 5. The expected response time is $F^R(Q)$ for STRQ and $F^N(Q)$ for STNNQ.

$$\#\text{ of Nodes Required} = \left\lceil \frac{\text{Expected Response time}}{\text{MAX_ResponseTime}} \right\rceil \tag{5}$$

Having estimated the number of nodes required, the request is sent to *node. Upon receiving the request, *node looks up the heart beat statistics to find currently available idle nodes and sends the feedback. Then, the task is decomposed into equal sized subtasks, each of which is assigned to an idle node.

[3] From experiments, we obtained $\alpha = 5e^{-4}$, $\beta = 1e^{-6}$, and $\gamma = 1e^{-6}$.

## 5 Experimental analysis

Section 5.1 details the experimental setup. Sections 5.2 and 5.3 concern the performance of the storage manager and query engine, respectively. Section 5.4 discusses results over real datasets.

### 5.1 Setup

We construct the torus clusters with virtual machines created by OpenStack. We study performance while considering torus clusters containing 4, 8, 12, 18, and **27** nodes, with each node having 2G RAM, a two-core 2.4G CPU, and using CentOS 6.4. The default size is 27 nodes.

We examine our methods with both synthetic and real datasets. We first use Brinkhoff's generator[4] to incrementally produce a large dataset based on the road network of San Francisco. A dataset containing 103.68 million data points and 7.02 million trajectories is used for the torus cluster with 27 nodes. So, in our default settings, each torus node stores 3.84 (=103.68/27) million data points and 0.26 (=7.02/27) million trajectories on average. For differently sized torus clusters, we randomly extract corresponding fractions of data to make sure the average storage load on each node is approximately the same as described above. For example, the torus cluster with 18 nodes stores 69.12 (=103.68 · 18/27) million data points. By doing this, we can test the scalability over torus cluster size while eliminating other factors.

We also use the Geolife dataset[5] from MSRA that contains 17,621 trajectories, and more than 19 million data points. We "inject" uncertainties for each observed point according to the GPS error,[6] which follows a Gaussian distribution with a sigma of 10 meters. By default, we use 200 sampling points to represent such a distribution.

We compare our system with several mainstream spatial big data platforms: SpatialHadoop [12], SpatialSpark [29], MongoDB,[7] and GeoCouch.[8] SpatialHadoop and Spatial Spark do not support temporal data and trajectory data in their latest versions. None of the systems support uncertain data, let alone uncertain trajectories. In comparison with these systems, we use spatial points. We apply our default settings for each competitor, meaning that each maintains 27 nodes storing 103.68 million points.

All programs are written in C/C++ and complied by gcc/g++ 4.4.7.

[4] http://iapg.jade-hs.de/personen/brinkhoff/generator/.

[5] http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/.

[6] http://en.wikipedia.org/wiki/DifferentialGPS.

[7] http://www.mongodb.org.

[8] https://github.com/couchbase/geocouch.

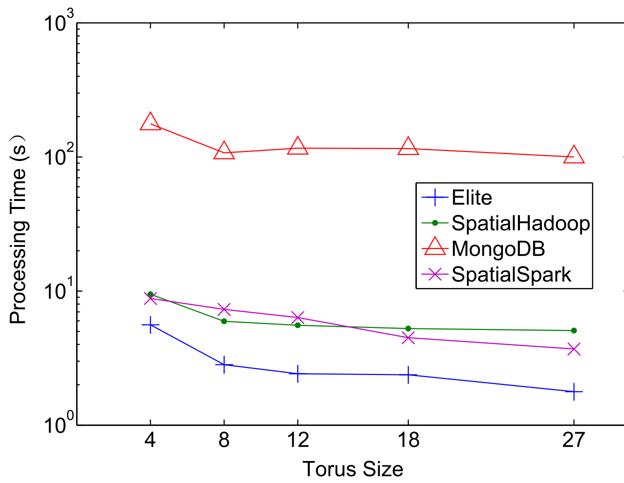


**Fig. 10** Storing and indexing comparison

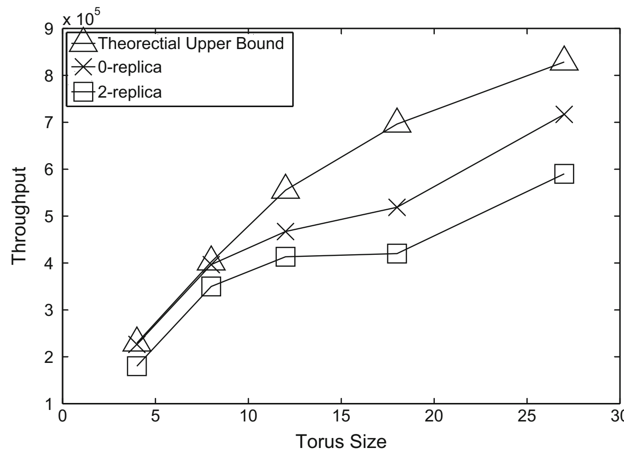


**Fig. 11** Throughput versus torus size

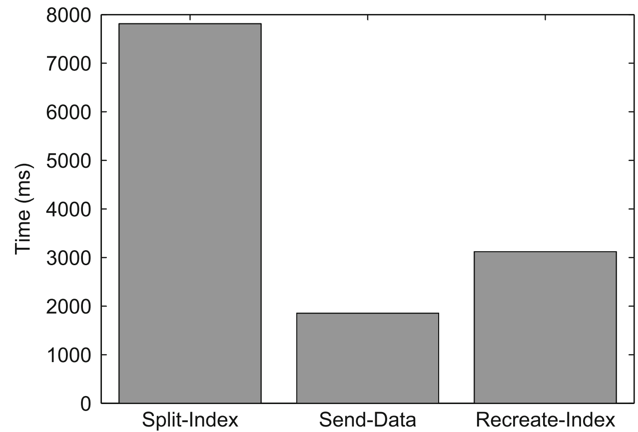


**Fig. 12** Breakdown for node splitting

## 5.2 Storage manager study

*Storage and indexing comparison* Elite and MongoDB are capable of storing data and maintaining the index synchronically. SpatialHadoop and SpatialSpark have to store the data first and index it afterward. We compare their performance at storing and indexing 1 million data points. Figure 10 shows the task processing time when varying the torus size from 4 to 27. The results show that Elite dominates the competitors in all settings. SpatialHadoop and SpatialSpark have similar performances and are about half an order of magnitude slower than Elite. We also tested the performance of GeoCouch, whose speed is two orders of magnitudes below that of MongoDB during data insertion. Next, we study the performance over uncertain trajectories.

*Throughput versus torus size* We show a magnified view of our method in Fig. 11. As can be seen, the throughput increases when more nodes are used. A larger torus has a larger network bandwidth, and the throughput increases. This is consistent with the potentially increased parallelism with respect to the torus cluster size. Also, the throughput increases sub-linearly with respect to the cluster size, indicating that our system has good scalability, thus supporting elastic storage management. The gap between the theoretical upper bound is due to extra system spending, e.g., heart beating. We also show the performance of Elite if no replicas are stored, marked as *0-replica* in Fig. 11. As shown, storing two replicas (our default setting) reduces the throughput by about 10–20 %. This performance reduction is the cost of fault tolerance.

In the sequel, we report results for two operations, node splitting and node appending, with which storage elasticity is achieved.

*Breakdown for node splitting* We report the results of splitting a torus node in Fig. 12. The threshold of node splitting is set to 3 million. This is below the average data volume of 3.8 million so that the results of node splitting can be observed. In practice, the setting of the capacity is adjusted according to the amount of memory available at virtual nodes, the number of replica copies maintained per virtual node, and the reserved heap and stack for programs. Node splitting involves three phases: splitting the oct-tree in the original node, sending data to the new torus node, and recreating the oct-tree in the new node. As shown in Fig. 12, the time for splitting the oct-tree is about 8 s, and the whole process takes about 13 s.

*Appending a new torus node* There are three steps for appending a new torus node, i.e., sending the executable binary files to the new torus node, starting the torus service in the new node, and broadcasting this change throughout the torus. Our experiments show that, the first step is the most expensive, requiring about 0.8 s. The second step needs about 0.7 s, and the third consumes only a few milliseconds. The whole cost of appending a new torus node is about 1.5 s.

*Recovery* The recovery process involves three steps: (1) appending a new node; (2) sending the replica to the new node; (3) rebuilding the index from the data. The cost of the

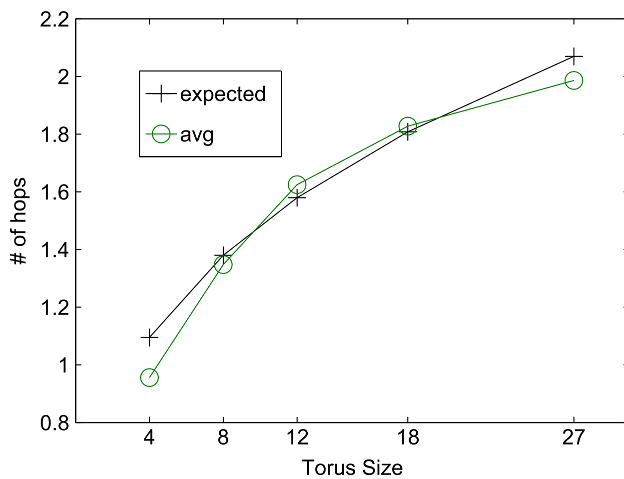


**Fig. 13** Routing cost analysis

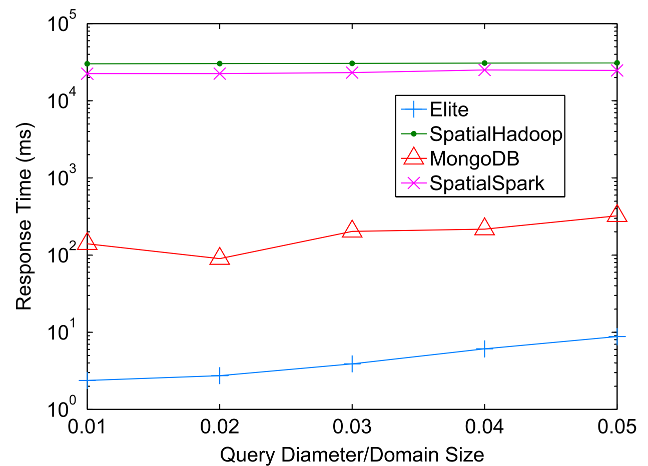


**Fig. 14** Query comparison

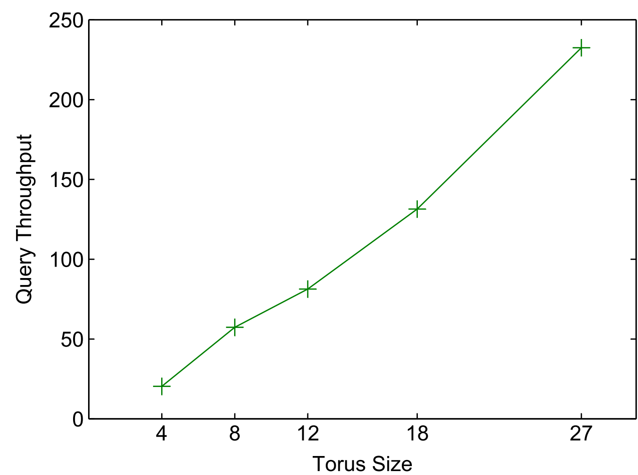


**Fig. 15** Query throughput versus torus size

first step is as mentioned above. The costs of the second and third steps depend on the size of the dataset. For our default setting with 3.8 million data points per node, we need about 4.05 s for sending the data, and 30.13 s for rebuilding the index. Thus, the time cost for recovering the index is higher than the time cost of node splitting (Fig. 12). This is because that during node splitting the local index is serialized and deserialized, which takes much less time. In other words, if we maintain the serialized index, we can reduce the index reconstruction time from 30.13 s to less than 4 s. But maintaining both the serialized index and the replica takes extra efforts and affects the insertion throughput. So, in our implementation, we opt to maintain "light-weighted" replicas.

*Routing cost analysis* In Sect. 3.3, we analyze the torus routing cost. Figure 13 shows the theoretical values with the experimental results. According to the results, the average routing costs are very close to the expected ones.

## 5.3 Query engine study

### 5.3.1 Query analysis

*Query comparison* We compare the response times of Elite, SpatialHadoop, SpatialSpark, and MongoDB in Fig. 14. GeoCouch does not yet support distributed indexing and is not included in this comparison. Again, for a fair comparison, we only consider selective queries over spatial points. A selective query over spatial points is a simplified version of STRQ, where no refinement is needed. As illustrated in Fig. 14, Elite performs several orders of magnitude faster than the three other systems. Now, we examine Elite alone for the queries studied in the paper, which involves both index traversal and probability computation.

*Query throughput* We evaluate the query throughput, i.e., the number of queries processed per second, for tori with different sizes. In all cases, each node contains about 3.84 million data points. The query set contains 10,000 queries. Half of the queries are STRQs, and the other half are STNNQs. For STRQ, the query diameters vary from 1 to 5 % of a node's domain size. For STNNQ, the query time span varies from 50 to 250. In this test, queries are sent to the tori continuously. As shown in Fig. 15, the throughput increases almost linearly with respect to the torus size. Hence, our system also has good scalability in terms of query processing.

*Accuracy* We test the accuracy of derived qualification probabilities with respect to the number of sampling points. We randomly generate 200 queries for both STRQ and STNNQ. For each query, we calculate the qualification probability by varying the number of sampling points, i.e., from 100 to 500, for each object at each timestamp. We find that the values of the derived qualification probabilities converge fast if the number of sampling points exceeds 200. Based on experiments with many different queries, we find that the variance of qualification probabilities is smaller than $10^{-7}$

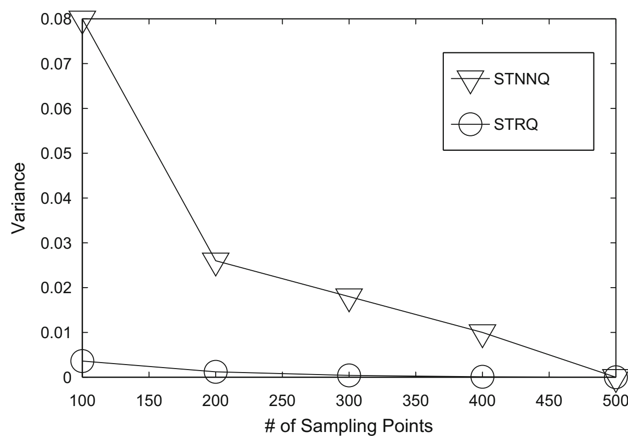


**Fig. 16** Variance versus # of samples

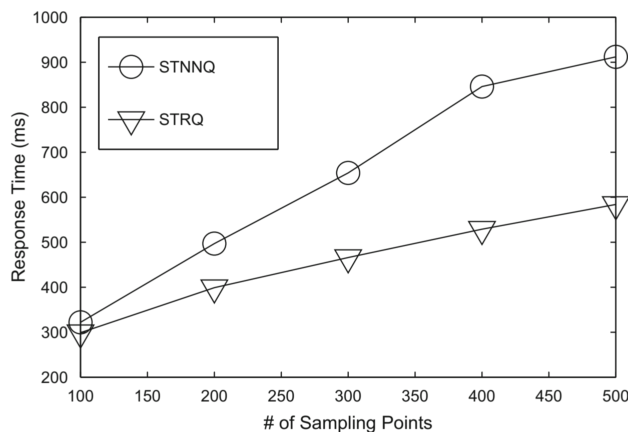


**Fig. 17** Response time versus # of samples

when the number of samples is 500, and we find that increasing the sample size decreases the variance only little. Hence, we regard the results obtained by using 500 samples as 'correct'. Moreover, according to the experience in previous work [23,28], 200 samples are sufficient to obtain high-quality results for probabilistic queries. Then, we count the difference from the result with 500 sampling points and calculate the variance over 200 queries. From Fig. 16, $x = 200$ is the inflection point. So, the setting of 200 sampling points is used as the default.

We also report the response time w.r.t. the number of sampling points in Fig. 17. The response time increases moderately. This is because the elastic method uses more computational units for higher workloads, i.e., those with more samples.

*Locking mechanism* We study the locking mechanism by generating a workload with mixed reading (STRQs with default settings) and writing (data insertion) tasks. We control the amount of the writing tasks to examine how much the simultaneous reading tasks are affected. The results are shown in Fig. 18, where the x-axis is the throughput of data

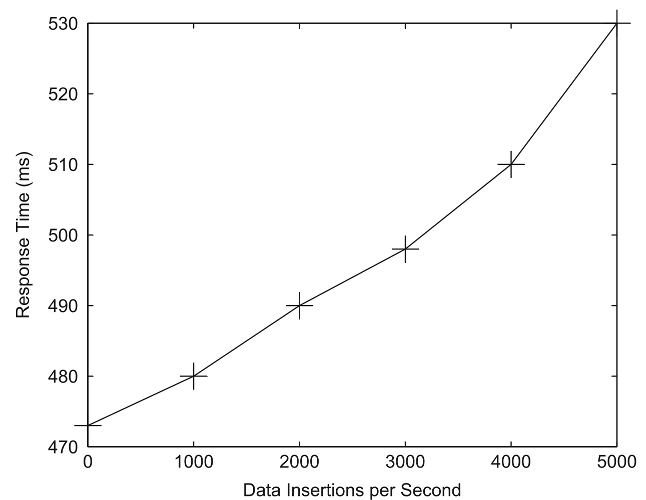


**Fig. 18** Effect of locking mechanism

insertions, and the y-axis is the response time of STRQ. We have two observations. First, the query response time increases proportionally with the data insertion rate. This occurs because a larger insertion rate increases the chance of "blocking" reading tasks. Second, the response increase is modest. For example, when the insertion rate increases three times (from 1000 to 3000 per second), the response time increases less than 3 %. This is consistent with the fact that the dominant part of a query is the refinement instead of the filtering that accesses the index. Also, the performance can be further improved by using finer locking granularities. We thus conclude that the locking mechanism performs stably across diverse workloads and that conflicts constitute only a minor issue in relation to the query performance.

*Effect of torus node condensing* Intuitively, a torus node's routing cost is proportional to its storage workload. If a torus node underflows, its bandwidth is not fully utilized because there is less chance that the node is accessed. Torus condensing aims to reduce the under-provisioning of storage and communication resources. If we assume a torus node's bandwidth is not exhausted as long as it does not overflow then a condensed node does not use all its bandwidth. A condensed torus node has a higher communication cost, which can potentially affect the routing efficiency. Even so, the routing overhead affects the filtering phase on slightly. As shown in Figs. 22 and 25, the cost in the filtering phase is dominated by the refinement cost. In summary, the effect of condensing on the query performance is small or non-existing.

*Scalability* We study the scalability when varying the number of trajectories stored in each node. To expel other factors, we use the default query setting, which fixes the query diameter to be 3 % of the domain size for STRQ and the query time span to be 150 for STNNQ. As shown in Fig. 19, the response time increases moderately. For example, when the number of trajectories is doubled (e.g., from 1M to 2M), the response time increases only 30 %. This is because our

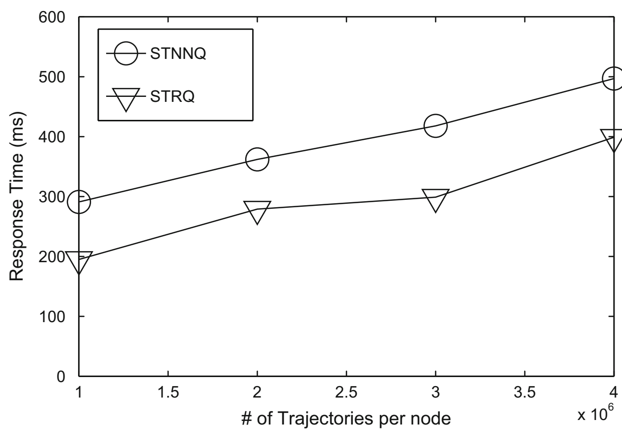


**Fig. 19** Response time versus # of trajectories

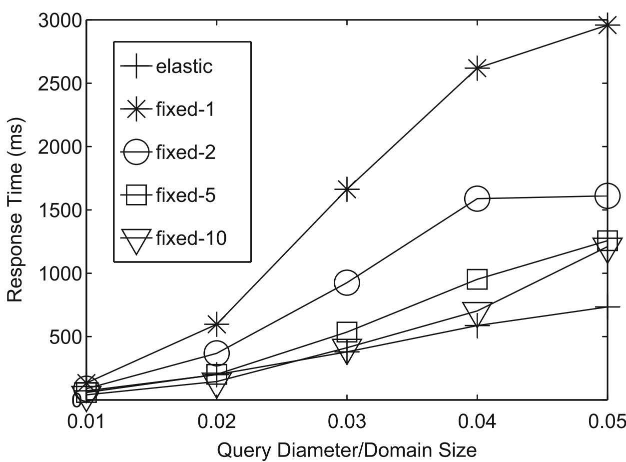


**Fig. 20** Response time of STRQ

elastic approach can utilize more computational resources for queries with higher costs. We proceed to investigate the computation elasticity for STRQ and STNNQ.

*Query time versus query range* The refinement phase of probabilistic queries usually dominates the query evaluation time because the computation of probabilities is expensive. In our framework, a torus node can share the refinement task with other nodes, if a query is too expensive to be handled by the local node. Our *elastic* strategy estimates the refinement cost and allocates an appropriate number of nodes. With this fine-grained parallelism, the query response time can be controlled to perform stably, e.g., below a tolerable limit. In contrast, an *inelastic* strategy ignores the query cost and always requires fixed number of nodes.

Figure 20 shows the response time of our approach and four *inelastic* approaches with fixed numbers of nodes, specially, 1, 2, 5, and 10 nodes, for performing refinement tasks. We denote them as fixed-1, fixed-2, fixed-5, and fixed-10. The query diameter varies from 1 to 5 % of the domain size.

As shown in the figure, our elastic approach performs the best. When the query diameter equals 5 % of the domain size, the response time of the elastic approach is about 600 milliseconds, while the inelastic approaches need 1–3 s. This is because the elastic approach can automatically tune more nodes to accomplish the refinement task. The *fixed-10* approach always requires ten nodes, which is more than enough for small queries, thus generally wasting system resources. For example, when the query diameter is 0.01, the performance *fixed-10* is very close to that of *elastic*. However, to achieve this performance, *fixed-10* mobilized five times as many nodes as *elastic*.

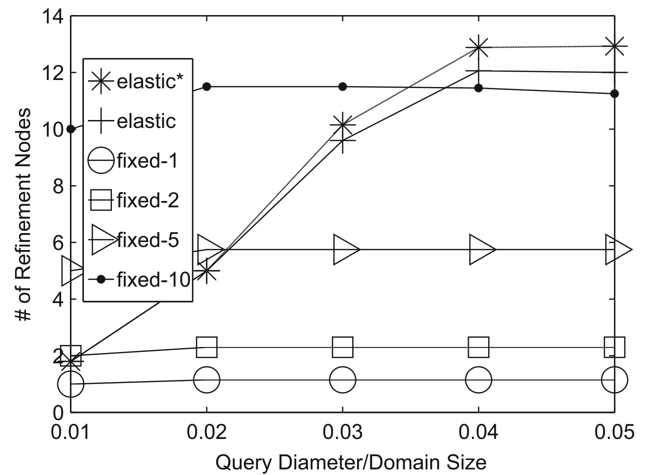


**Fig. 21** # of Refinement computers of STRQ

In Fig. 21, we illustrate the average number of nodes utilized for each refinement task. Here, *elastic** is the number of nodes requested by the *elastic* approach. The number of nodes actually obtained is smaller than the requested number when the query diameter is large because not enough idle nodes exist when the workload is high.

As illustrated by the figure, the *elastic* approach will apply for computing resources according to the workload. Hence, it can make good use of the computing resources for query processing. The inelastic approaches, however, utilize excessive computing resources when the query diameter is small and ask for insufficient resources for large queries. Note that a query may overlap with multiple torus nodes and may then be split into several sub-queries. Each sub-query will require refinement nodes independently. For example, that is why the *fixed-10* approach utilizes more than ten nodes.

In summary, *inelastic* methods are not favorable in a cluster environment, in the sense that they either waste computational resources or exhibit deteriorating throughput, because they request too many or too few resources.

*Breakdown for query time* Figure 22 shows the time cost in filtering and refinement phrases, when the query diameter is 5 % of the domain size. The *elastic* approach requires more time for performing filtering than the inelastic ones do. During the filtering phase, a query may be split into several sub-queries, and each sub-query is sent to an idle node to per-

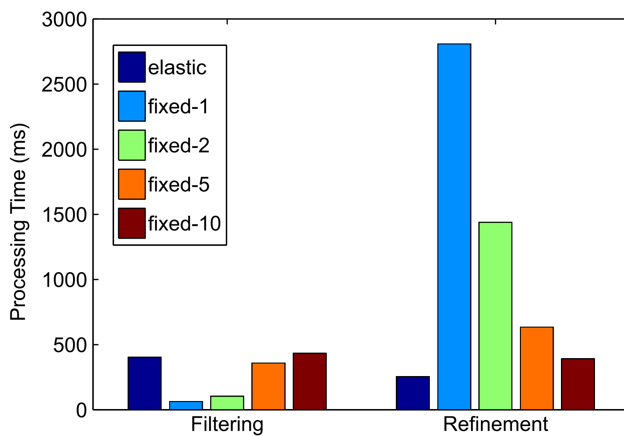


**Fig. 22** Breakdown analysis of STRQ

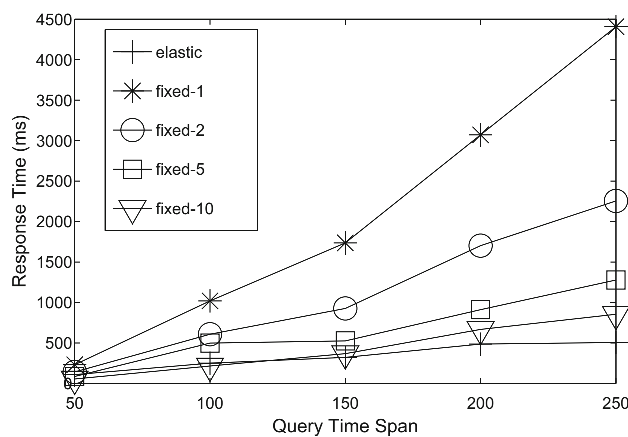


**Fig. 23** Response time of STNNQ

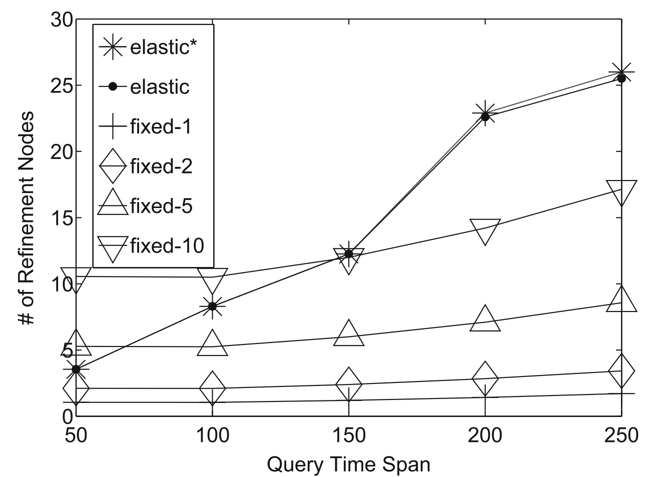


**Fig. 24** # of Refinement computers of STNNQ

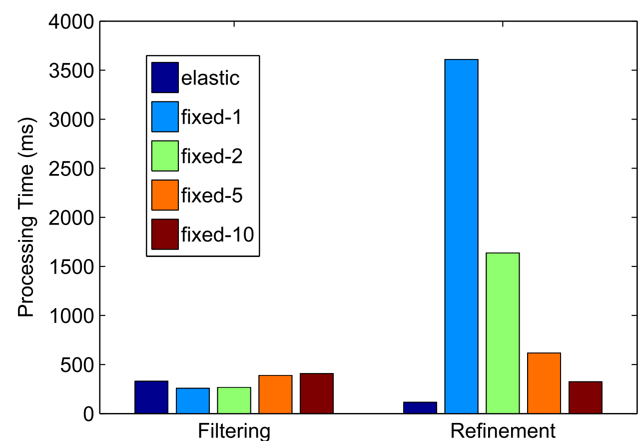


**Fig. 25** Breakdown analysis of STNNQ

form refinement. The more sub-queries, the larger the time needed to prune unrelated candidate trajectories. The effort is rewarding in the refinement phase. As shown in Fig. 21, the *elastic* approach requires a suitable number of refinement nodes for these sets of queries. Similarly, *fixed-10* also takes more time than the other three inelastic approaches in filtering, but achieves better performance in refinement and overall query evaluation.

#### *5.3.2 STNNQ*

*Query time versus query length* Figures 23 and 24 show the response time and number of refinement nodes for STNNQ. The x-axis of these two figures shows the time interval of STNNQs.

Similar to STRQ, the *elastic* approach again surpasses the inelastic approaches. Note that the refinement phase of STNNQ is usually much more costly than that of STRQ. Hence, more nodes are needed for performing refinement tasks.

*Query time breakdown* Figure 25 shows the breakdown analysis of STNNQ. Similar to STRQ, the *elastic* approach needs more time in the filtering phase to generate more sub-queries, and this results in less time extended in the refinement phase.

### 5.4 Results on real datasets

Figures 26 and 27 show the results over the real dataset, Geolife. Considering the relatively small data volume, we choose to deploy the whole dataset on an eight-nodes torus. We compare our *elastic* approach with *fixed-2* since there are only eight nodes. Different from the synthetic dataset, the data points in Geolife are quite sparse. The query results become quite small, and the average response time of *elastic* fluctuates around 20 milliseconds. However, our approach still outperforms the inelastic one. The *fixed-2* approach always partitions the queries into two parts, which is not necessary because the queries can be evaluated very fast on the local node. The unnecessary partitioning leads to extra filtering

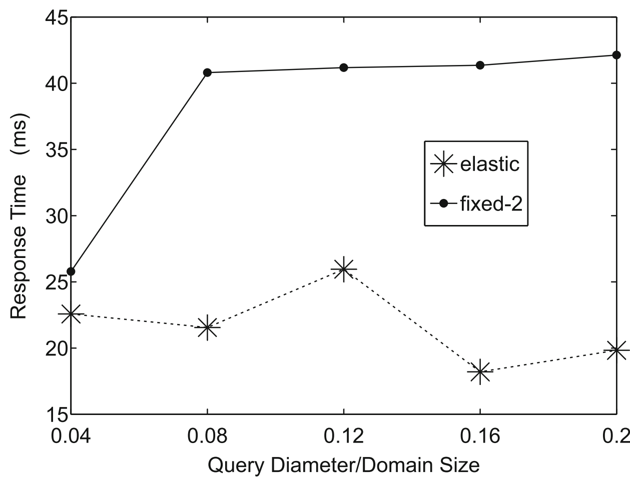


**Fig. 26** Response time of STNNQ

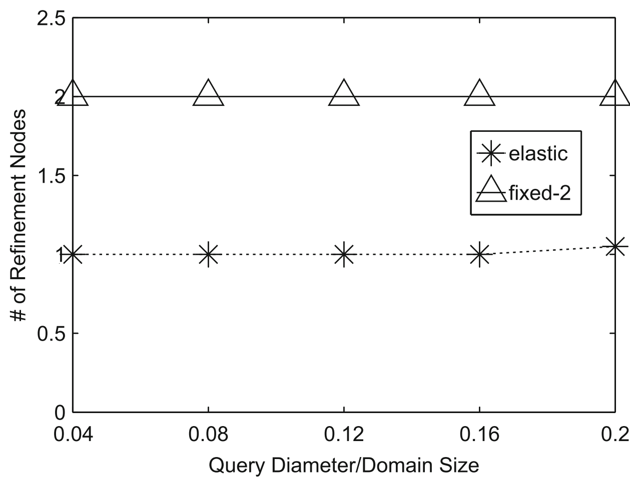


**Fig. 27** # of Refinement computers

and network transmission costs, thus increasing the response time.

# 6 Related work

## 6.1 Querying uncertain trajectories

Figure 28 shows representative models for uncertain trajectories. If the motion of a moving object can be bounded at each time instance [8,30], the position can be bounded using the cylinder model in Fig. 28a. If an object updates its precise location periodically, the position between two updates is bounded by a cone, as shown in Fig. 28b. Given the maximum speed, the location between two successive updates can be bounded by an ellipse [21] as shown in Fig. 28c.

We adopt the cylinder model because of its simplicity in capturing trajectory uncertainty. Our framework and algorithms can be adapted to using other representations as well,

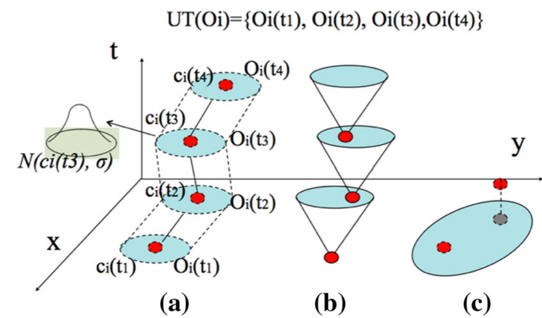


**Fig. 28** Example of uncertain trajectory models: **a** Cylinder model [8, 30], **b** Cone model [7], **c** Ellipse model [21]

by using cylinders to minimally bound cones or ellipses. Then, the uncertainty can be recovered during querying by sampling points.

Recently, a number of interesting results have appeared on how to evaluate queries over trajectory data while considering uncertainty. Zheng et al. [31] study range queries over trajectory data. Trajcevski et al. [8] investigate the problem of efficiently executing continuous *NN* queries for uncertain moving objects trajectories. Zheng et al. [32] study two variants of the *k-NN* query for fuzzy objects. They return the qualifying objects that satisfy a probabilistic distance threshold or a range of probability thresholds, respectively. Xie et al. [33] study how to evaluate trajectory nearest neighbor queries over imprecise location data.

However, all the above studies perform query evaluation on a centralized server. This approach falls short when the workloads considered cannot be accommodated by a single machine.

## 6.2 Indexing spatiotemporal data

Indexing trajectory data have been studied extensively for more than a decade, resulting in proposals such as the MV3-RTree [34], TB-tree [35], and SETI [36]. It is not clear how those works can be extended to distributed settings. Recently, substantial efforts have also been devoted to the organization of data in a distributed setting. Two types of approaches may be distinguished depending on how the indexes are deployed.

*Centralized index distributed storage* The infrastructure consists of a *master node* and many *data nodes* [11–14]. The entire index or its upper layers are deployed at the master node as a global index. Such a structure falls short in three respects: (1) inefficiency in handling large current data requests due to the throughput bottleneck; (2) limited scalability to the index size which is proportional to the data volume; (3) vulnerability to infrastructure maintenance.

*Distributed index distributed storage* The infrastructure is based on a peer-to-peer network. The query goes through several hops to retrieve the data required [15,16,37,38]. Such

a strategy alleviates the single-node problem of a centralized index. However, existing proposal falls short in addressing the specific requirements of large spatiotemporal data. The hashing techniques that constitute the basis of Appache Cassandra [38] do not guarantee the correctness of the proximity search considered in this work. RT-CAN [15] and MIDAS [37] construct indexes on a predefined domain space, whereas the domain space of spatiotemporal data evolves over time. Notably, the temporal dimension expands over time. These existing proposals focus mainly on how to best distribute the storage load across nodes. They put less emphasis on how to distribute the computational workload among nodes, which is critical in spatiotemporal applications. Further, the existing proposals are designed primarily with point data in mind rather than complex uncertain spatiotemporal trajectory data.

## 7 Conclusion and future work

In this paper, we continue and study an elastic infrastructure for managing big spatiotemporal trajectories. The infrastructure achieves both storage and computational elasticity by employing a fully distributed peer-to-peer storage scheme and a parallel computing mechanism. We investigate how the infrastructure can support the two most fundamental queries, namely range and nearest neighbor queries. Extensive experiments offer insight into the elasticity and efficiency of the proposed infrastructure.

While the paper provides the core system for trajectory data management, future research should study how the system can be used for supporting other queries than range and nearest neighbor queries. One such query is the *spatiotemporal kNN query (STkNNQ)* that extends the STNNQ to retrieve $k$ nearest neighbors. This query can be defined by using expected distances[9] for the ranking of trajectories. The expected distance between an object $O_i$ at time point $t$ and a query point $q$, denoted by $E(|q, O_i(t)|)$, is given by the following equation.

$$E(|q, O_i(t)| = \sum_j |q, O_i(t)^{(j)}| \cdot Pr\left(O_i(t)^{(j)}\right)$$

Then, a query $Q = (q, \Delta t, k)$ consisting of a spatial point location $q$, a time interval $\Delta t$, and an integer $k$ retrieves a set of uncertain trajectories $\{UT_i\}$ such that $E(|q, O_i(t)|)$ is among the $k$ smallest expected distances at any time point in $Q \cdot \Delta t$. To compute the query, we can reuse the STNNQ algorithm with two modifications:

– To obtain $d_{max}$ it is necessary to find at least $k$ objects at each time point in $Q \cdot \Delta t$. This can be achieved by setting $m$ to $k$ in Eq. 8;
– The cost involves: (1) generating sampling points; (2) calculating expected distances; (3) retrieving top-$k$ results; (4) merging the result. The cost function is thus a linear function w.r.t. the number of candidates.

As another example of a relevant query type, the *spatiotemporal NN query over trajectories (STNNTQ)* extends the STNNQ by replacing the spatial point and a time interval $(q, \Delta t)$ parameters with an uncertain trajectory $U_Q$. Given an uncertain trajectory $U_Q$, an STNNTQ retrieves a set of uncertain trajectories $\{UT_i\}$ that can be $U_Q$'s nearest neighbors during $U_Q \cdot \Delta t$ with nonzero probability.

– We obtain a new qualification probability function by replacing $q$ with $U_Q$ in Eq. 2. So, $F(q, UT_i, t)$ is replaced with $F(U_Q, UT_i, t)$, where:

$$F(U_Q, UT_i, t) = 1 - \sum_{\{j|O_i^{(j)}(t)\}} Pr(O_i^{(j)}(t)) \prod_{g \neq i \wedge O_g \in D} Pr\{|O_Q(t), O_i^{(j)}(t)| < |O_Q(t), O_g(t)|\}$$

– The filtering bound is set to the minimum cylinder containing $U_Q \oplus d_{max} \oplus U_{max}$. The derivation of $d_{max}$ follows that of the STNNQ. A series of consecutive cylinders containing $U_Q$ can be used to achieve additional filtering.
– We can also set a threshold for pruning candidate trajectories with low qualification probabilities.

In addition to the above example queries, it is relevant to study how the infrastructure can support analytical queries, e.g., spatial joins, with the proposed fundamental queries as building blocks.

**Acknowledgments** This work was supported by the 973 program with No 2012CB316205, a grant from the Obel Family Foundation, and National Science Foundation of China under Grant No. 61432006. The work was done in part when some of the authors visited SA Center for Big Data Research at Renmin University of China. The center is partially funded by the Chinese National "111" Project "Attracting International Talents in Data Engineering and Knowledge Engineering Research".

[9] There exist different semantics for top-k queries over uncertain data, such as U-Top*K*, U-*k*Ranks, Expected scores, and Expected ranks. Among them, the Expected score and Expected rank might be best ones in terms of properties such as Containment and Unique-Rank [39].

## Appendix 1: Algorithms

**Algorithm 3** Intersect

```
1: function INTERSECT(region a, region b)
2:   for each dimension i = 1 to d do
3:     if a.low[i] > b.high[i] or a.high[i] < b.low[i] then
4:       return false
   return true
```

**Algorithm 4** Range Query (torus level)

```
1: function RANGE QUERY(range query q, torus node T)
2:   if Intersect(T.R, q.R) is true then
3:     for each dimension i = 1 to d do
4:       ForwardRequest(q,T,i)
5:     Performs local search on T
6:   else
7:     for each dimension i = 1 to d do
8:       if q.R.low[i] > T.R.high[i] or q.R.high[i] < T.R.low[i] then
9:           ▹ the condition tests if q and T overlap on dimension i
10:        ▹ the condition cannot be false d times; otherwise, q and T intersect and were handled by steps 1–4
11:          ForwardRequest(q,T,i)
12:          break
```

**Algorithm 5** ForwardRequest

```
1: function FORWARDREQUEST(query range q, torus node T, dimension i)
2:   n_low[d] ← the lower neighbor node of T on dimension d
3:   n_high[d] ← the upper neighbor node of T on dimension d
4:   if q.R.low[i] < t.R.low[i] then
5:     SendQueryRequest(q,n_low[i],i)
6:   if q.R.high[i] > t.R.high[i] then
7:     SendQueryRequest(q,n_high[i],i)
```

## Appendix 2: Routing cost estimation

**Lemma 1** *The maximum routing cost of a three-dimensional torus of $N$ nodes is approximately equal to* $0.91N^{\frac{1}{3}}$.

*Proof* For any torus node $n$, it takes one hop to reach its six neighbors (first-order neighbors) and two hops to reach its 18 second order neighbors. The number of $i$-th order neighbors $a^i$ can be represented by [40]:

$$a_i = 2 + 4i^2$$

Suppose the furthest node on the torus takes $m$ hops from node $n$. The total number of nodes $N$ visited equals the summation of the number of 1-th to $m$-th order neighbors. We have:

$$\sum_{i=1}^{m} a_i = N \Rightarrow \sum_{i=1}^{m} 2 + 4i^2 = N \Rightarrow m \approx \left(\frac{3}{4}N\right)^{\frac{1}{3}}$$

Thus, the maximum routing cost equals to the distance to $n$'s $m$-th order neighbor, $0.91N^{\frac{1}{3}}$. □

**Lemma 2** *The average routing cost of a three-dimensional torus of $N$ nodes is approximately equal to* $0.69N^{\frac{1}{3}}$ *hops.*

*Proof* Based on Lemma 1, the furthest node from $n$ requires $m$ hops. Then, the average number of hops is:

$$\begin{aligned} avg_number_of_hops &= \frac{1}{N}\sum_{i=1}^{m} a_i \cdot i = \frac{1}{N}\sum_{i=1}^{m} 2i + 4i^3 \\ &= \frac{1}{N}\left(m^4 + \frac{8}{3}m^3 + 2m^2 + \frac{1}{3}m\right) \\ &\approx 0.69N^{\frac{1}{3}} \end{aligned} \tag{6}$$

□

## Appendix 3: Estimation of $h$

Let us consider the cost estimation on torus node $T_i$. After the range search $Q_i \oplus d_{max} \oplus U_{max}$ (`Step 6` in Algorithm 2), we get a set $C_i$ of candidate trajectories with the average length $\overline{\mathcal{T}_c \cdot \Delta t} = \frac{1}{|C_i|}\sum_{\mathcal{T}\in C_i} \mathcal{T} \cdot \Delta t$. According to Definition 5, the cost of STNNQ depends on the number of trajectories at each snapshot. To estimate that, we first assume the trajectories are uniformly distributed in the spatiotemporal region $Q_i \oplus d_{max} \oplus U_{max}$.

$$\#\ of\ objects\ per\ snapshot = \frac{\overline{\mathcal{T}_c \cdot \Delta t} \cdot |C_i|}{|Q_i \cdot \Delta t|} \tag{7}$$

We define the density $\rho$, as the number of objects per snapshot divided by the area of the filtering bound $\pi(d_{max} + U_{max})^2$.

**Lemma 3** *Assume a two-dimensional region $S$ in the spatial domain $\mathfrak{S}$, where the points are uniformly distributed, and let $N(S) = m$ represent the fact that there are $m$ points inside region $S$. The probability of $N(S) = m$ is given by:*

$$P(N(S) = m) = \frac{\rho|S|e^{-\rho|S|m}}{m!} \tag{8}$$

*Proof* The probability that $m$ points out of $n$ objects are in $S$ is:

$$P(N(S) = m) = \binom{n}{m}\left(\frac{|S|}{|\mathfrak{S}|}\right)^m \left(1 - \frac{|S|}{|\mathfrak{S}|}\right)^{n-m}$$

The extreme form of the binomial distribution is a Poisson distribution. Let $\rho = \frac{n}{|\mathfrak{S}|}$. The above equation becomes:

$$P(N(S) = m) = \frac{(\rho|S|)^m e^{-\rho|S|}}{m!}$$

Then, the probability that there is at least one point in $S$ is:

$$P(N \geq 1) = \sum_{i=1}^{\infty} P(N = i) = \sum_{i=1}^{\infty} \frac{(\rho|S|)^i e^{-\rho|S|}}{i!} = 1 - e^{-\rho|S|}$$

□

Then, we can infer that there is a nearest neighbor within the circular region $S$ to the query point with a probability higher than $P^*$. In our implementation, we set $P^*$ to 0.9, which is reasonably large for $S$ to contain the nearest neighbor.

$$|S| = -\frac{ln\,(1 - P^*)}{\rho} \tag{9}$$

The number of candidate objects per snapshot is estimated as:

$$h = \rho(|S \oplus U_{max}|). \tag{10}$$

## Appendix 4: Obtaining $d_{\max}$

In our system, we try a series of range queries $Q_i \oplus d \oplus U_{max}$ to incrementally obtain $d_{max}$, where $d = 5, 10, 20\,\%, \cdots$ of torus node $T_i$'s spatial domain size. Upon collecting the candidate trajectory set $C_i$ by the range search parameterized with $d$, we test whether the union of these trajectories' time spans can cover $Q_i$'s $\Delta t$, i.e., to decide whether $\cup_{UT \in C} UT \cdot \Delta t \supseteq Q_i \cdot \Delta t$ is true. If true, it means that there always exists at least an object for each timestamp in $Q_i \cdot \Delta t$. Therefore, current $d$ is taken as $d_{max}$, which is sufficiently large for not missing any possible candidate trajectories. Otherwise, we need to increase $d$ incrementally and repeat the aforementioned process.

## References


1. Ceikute, V., Jensen, C.S.: Vehicle routing with user-generated trajectory data. In: MDM (2015)
2. Yang, B., Guo, C., Ma, Y., Jensen, C.S.: Toward personalized, context-aware routing. VLDB J. **24**(2), 297–318 (2015)
3. Dai, J., Yang, B., Guo, C., Jensen, C.S.: Efficient and accurate path cost estimation using trajectory data. In: CoRR abs/1510.02886 (2015)
4. Stougiannis, A., Pavlovic, M., Tauheed, F., et al.: Data-driven neuroscience: enabling breakthroughs via innovative data management. In: SIGMOD (2013)
5. Manyika, J., Chui, M.: Big data: the next frontier for innovation, competition, and productivity. In: McKinsey Global Institute (2011)
6. Cheng, R., Kalashnikov, D.V., Prabhakar, S.: Evaluating probabilistic queries over imprecise data. In: SIGMOD (2003)
7. Cheng, R., Kalashnikov, D.V., Prabhakar, S.: Querying imprecise data in moving object environments. TKDE **16**(9), 1112–1127 (2004)
8. Trajcevski, G., Tamassia, R., Ding, H., et al.: Continuous probabilistic nearest-neighbor queries for uncertain trajectories. In: EDBT (2009)
9. Civilis, A., Jensen, C.S., Pakalnis, S.: Techniques for efficient road-network-based tracking of moving objects. TKDE **17**(5), 698–712 (2005)
10. Jensen, C.S., Pakalnis, S.: Trax - real-world tracking of moving objects. In: VLDB (2007)
11. Eldawy, A., Li, Y., Mokbel, M.F., Janardan, R.: CG_Hadoop: computational geometry in mapreduce. In: GIS (2013)
12. Eldawy, A., Mokbel, M.F.: A demonstration of SpatialHadoop: an efficient MapReduce framework for spatial data. In: VLDB (2013)
13. Aji, A., Wang, F., Vo, H., et al.: Hadoop-GIS: a high performance spatial data warehousing system over mapreduce. In: VLDB (2013)
14. Nishimura, S., Das, S., Agrawal, D., El Abbadi, A.: MD-HBase: a scalable multi-dimensional data infrastructure for location aware services. In: MDM (2011)
15. Wang, J., Wu, S., Gao, H., et al.: Indexing multi-dimensional data in a cloud system. In: SIGMOD (2010)
16. Tsatsanifos, G., Sacharidis, D., Sellis, T.: Index-based query processing on distributed multidimensional data. GeoInformatica **17**(3), 489–519 (2013)
17. Ratnasamy, S., Francis, P., Handley, M., et al.: A scalable content-addressable network. In: SIGCOMM (2001)
18. Wei, L.Y., Zheng, Y., Peng, W.C.: Constructing popular routes from uncertain trajectories. In: KDD (2012)
19. Pei, T., Zhou, C., Zhu, A.-X, et al.: Windowed nearest neighbour method for mining spatio-temporal clusters in the presence of noise. In: International Journal of Geographical Information Science (2010)
20. Dalvi, N.N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: VLDB (2004)
21. Pfoser, D., Jensen, C.S.: Capturing the uncertainty of moving-objects representations. In: SSDBM (1999)
22. Lian, X, Chen, L.: Monochromatic and bichromatic reverse skyline search over uncertain databases. In: SIGMOD (2008)
23. Kriegel, H.P., Kunath, P., Renz, M.: Probabilistic nearest-neighbor query on uncertain objects. In: DASFAA (2007)
24. Pugh, W.: Concurrent maintenance of lists. Technical report, Dept. of Computer Science, University of Maryland (1990)
25. Gargantini, I.: An effective way to represent octrees. Commun. ACM **25**(12), 905–910 (1982)
26. Sidlauskas, D., Saltenis, S., Christiansen, C.W., et al.: Trees or grids?: indexing moving objects in main memory. In: GIS (2009)
27. Sidlauskas, D., Saltenis, S., Jensen, C.S.: Processing of extreme moving-object update and query workloads in main memory. VLDB J. **23**(5), 817–841 (2014)
28. Cheng, R., Chen, J., Mokbel, M., Chow, C.Y.: Probabilistic verifiers: evaluating constrained nearest-neighbor queries over uncertain data. In: ICDE (2008)
29. You, S., Zhang, J., Gruenwald, L.: Large-scale spatial join query processing in cloud. In: ICDE Workshops (2015)
30. Trajcevski, G., Wolfson, O., Zhang, F., Chamberlain, S.: The geometry of uncertainty in moving object databases. In: EDBT (2002)
31. Zheng, K., Trajcevski, G., Zhou, X., Scheuermann, P.: Probabilistic range queries for uncertain trajectories on road networks. In: EDBT (2011)
32. Zheng, K., Fung, G.P.C., Zhou, X.: K-nearest neighbor search for fuzzy objects. In: SIGMOD (2010)

33. Xie, X., Yiu, M.L., Cheng, R., Lu, H.: Scalable evaluation of trajectory queries over imprecise location data. TKDE **26**(8), 2029–2044 (2014)
34. Tao, Y., Papadias, D.: MV3R-Tree: a spatio-temporal access method for timestamp and interval queries. In: VLDB (2001)
35. Pfoster, D., Jensen, C.S., Theodoridis, Y.: Novel approaches to the indexing of moving object trajectories. In: VLDB (2000)
36. Chakka, V.P., Everspaugh, A.C., Patel, J.M., et al.: Indexing large trajectory data sets with SETI. In: The first biennial conference on innovative data systems research (CIDR) (2003). http://www.cidrdb.org/cidr2003/program/p15.pdf
37. Tsatsanifos, G., Sacharidis, D., Sellis, T.: RIPPLE: a scalable framework for distributed processing of rank queries. In: EDBT (2014)
38. The apache cassandra project. http://cassandra.apache.org/
39. Cormode, G., Li, F., Yi, K.: Semantics of ranking queries for probabilistic data and expected ranks. In: ICDE (2009)
40. Born, M.: On the stability of crystal lattices. IX. Covariant theory of lattice deformations and the stability of some hexagonal lattices. In: Proceedings of the Cambridge Philosophical Society 38 (1942)