An Interactive Visualization System for Streaming Data Online Exploration

Fengzhou Liang[1], Fang Liu[2(✉)], Tongqing Zhou[3], Yunhai Wang[4], and Li Chen[5]

[1] Sun Yat-sen University, Guangzhou 510000, China
liangfzh@mail2.sysu.edu.cn
[2] Hunan University, Changsha 410000, China
fangl@hnu.edu.cn
[3] National University of Defense Technology, Changsha 410000, China
zhoutongqing@nudt.edu.cn
[4] Shandong University, 250000 Jinan, China
[5] University of Louisiana at Lafayette, Lafayette, LA 70503, USA
li.chen@louisiana.edu

Abstract. The practices of understanding real-world data, in particular the high dynamic streaming data (e.g., social events, COVID tracking), generally relies on both human and machine intelligence. The use of mobile computing and edge computing brings a lot of data. However, we identify that existing data structures of visualization systems (a.k.a., data cubes) are designed for quasi-static scenarios, thus will experience huge efficiency degradation when dealing with the ever-growing streaming data. In this work, we propose the design and implementation of an enhanced interactive visualization system (i.e., Linkube) based on novel structure and algorithms support, for efficiently and intelligibly data exploration. Basically, Linkube is designed as a multi-dimensional and multi-level tree with spatiotemporal correlated knowledge units linked into a chain. Interested knowledge aggregations are thus attained via efficient and flexible sequential access, instead of dummy depth-first searching. Meanwhile, Linkube also involves a smart caching mechanism that adaptively reserves some beneficial aggregations. We implement Linkube as a web service and evaluate its performance with four real-world datasets. The results demonstrate the superiority of Linkube on response time (∼25% ↓) and structure updating time (∼45% ↓), compared with state-of-the-art designs.

Keywords: Man-computer interactions · Streaming data · Interactive visualization · Data analysis · Data structure

1 Introduction

Discovering phenomena and essence from real-world data relies on both machine intelligence for tremendous parsing tasks and human intelligence for subjective decision-making. To seamlessly integrate both intelligence, we have practically seen a lot efforts devote to interactive visualization for semi-automatic

S. Longfei and P. Bodhi (Eds.): MobiQuitous 2022, LNICST 492, pp. 515–534, 2023.
https://doi.org/10.1007/978-3-031-34776-4_27

and explainable data exploration in intelligent systems [12,24]. For example, Tableau [25], a famous visualization software, has won unanimous praise from customers by elaborating data statistics visually with interaction interfaces for various applications [15,27] (e.g., pollution monitoring, COVID tracking, city planning).

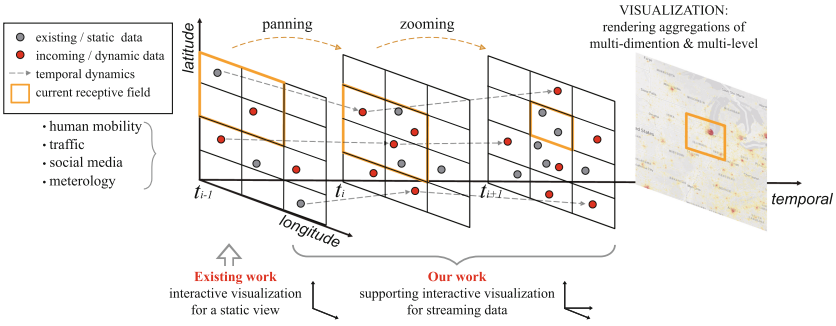

Fig. 1. A toy example of interactive visualization for streaming data. Along with continuous incoming data, statistics (e.g., density of a focused region) that an expert is interested in need to be dynamically re-calculated. Existing work handles such dynamics in an offline manner, while we investigate online visualization to reactively update statistics and respond to interactions with incoming knowledge.

The brain behind these applications is dedicated data structures that support interactive (e.g., spatial and temporal zooming, panning) knowledge searching and querying (e.g., density, numbers), known as data cubes. As a pioneering work, Nanocubes [13] adopts a tree-like structure to organize data and provides comprehensive operation strategies for finding and calculating knowledge aggregations or statistics (these two terms are used interchangeably in this paper). Given that storing all the aggregations in prior would consume significant memory resources, researchers bring forth the Hashedcubes and Smartcube structures in the follow-up work, which manage to identify and only reserves partial aggregations in advance for memory efficiency. However, we emphasize that the existing proposals are all **designed to be constructed offline on quasi-static scenarios**. As shown in the example of Fig. 1, these data cubes are built on data collected in specific spatiotemporal intervals, so the visualized views are in fact static, although the interactions are performed in real-time. That is, the inherent structure intelligence of them only supports experts to explore stale data.

This can be problematic when coming to the cases of streaming data, particularly for dynamic urban data [22]. For example, massive Twitter data are dynamically gathered to examine the evacuation response of residents during Hurricane Matthew in 2017 [18], which can be illustratively depicted in Fig. 1. In this example, for the experts to accurately understand the evolving situations, it is essential for the underlying data cubes to absorb the continuous incoming

event data and update the corresponding aggregations of different regions and levels. Unfortunately, traditional data cubes (e.g., [9,13,14]) cannot properly respond to such frequent updates, as they depend heavily on static knowledge aggregation in advance, which is, however, not possible because streaming data are collected online. As a result, experts would have to experience unexpected latency (far beyond the empirical threshold 500 ms [16]) to get responses for their interactive queries, seriously hampering the exploration.

In this work, we propose to enhance interactive visualization with novel data structure and algorithm support for efficiently and intelligibly exploring streaming data. The systematic design, Linkube, attempts to attain efficient response on interactions by adapting the data cube structure to temporal dynamics. For this, it basically maintains certain structure information to simplify the dummy traversal process (i.e., depth-first searching) for finding aggregation in the built tree. In particular, it constructs a linked list [2] for the spatial or temporal correlated root nodes (i.e., knowledge units) in the tree. Having these structural basics, we further maintain an index for each aggregation (i.e., nodes in the middle layers) to the corresponding first children root node, which facilitates Linkube with the advantages of sequential access for arbitrary knowledge. In this way, the response time of Linkube will be significantly reduced when performing queries on the incoming new data, because it can **update the tree structure and search for and calculate the queried aggregations online at the same time**.

Furthermore, to avoid proactively calculating large numbers of unnecessary aggregations, we design **a smart caching mechanism that reserves beneficial aggregation for dynamic reuse**. It measures the utility of aggregations based on the frequency they are queried, as well as the estimated traversal time for attaining them. We then propose to adaptively cache the pre-accessed values with the highest utility. Finally, the main contributions of this work can be summarized as follows:

- We identify the structural drawbacks of existing data cubes on dealing with streaming data, and propose the design of a novel data structure and its corresponding searching strategies for flexible data exploration. It can adaptively maintain the structure online and provide fast query/search responses simultaneously.
- Technically, Linkube introduces linked lists and vertical indices to retain multi-dimensional structure information of streaming data for sequential data access and efficient knowledge aggregation computation; A smart caching mechanism is also designed to quantify the utility of reserving certain aggregations and further reduce the response time.
- We implement Linkube as a web-based prototype and test its performance on four real-world datasets. The results demonstrate Linkube's superiority on three state-of-the-art data cubes, in terms of both response time and structure updating time.

2 Related Work

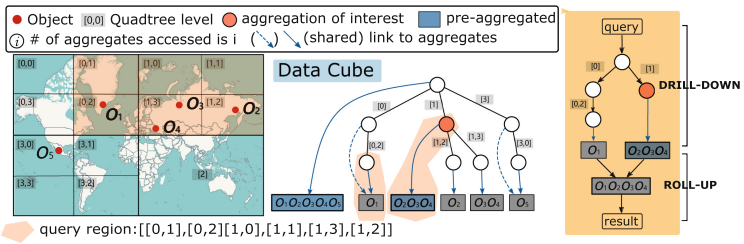

Fig. 2. The above map is divided into four sub-regions and indexed. The data cube builds a tree structure to support efficient searching of objects. An example on conducting a query: finding all objects in query region .

Streaming Visualization. Streaming data is a continuous flow of data, according to the definition by [1]. An obvious feature is that the system cannot control the time of the arriving data also the number of updates. The system generally does not know the scale of data, needing to constantly update to accommodate new data. Streaming visualization is more concerned with showing significant changes in the context of the past data [8]. For example, how to add or remove elements from the visualization based on layout choice. Streaming data visualization is also widely used, such as dynamic network visualization [23], spatiotemporal data analysis [5] and event detection techniques [26].

As a method to deal with the aggregation of stream, sliding window is widely used to filter data in valid spatiotemporal domain. StreamSqueeze [17] use flexible sliding windows, the problem of the data accumulation in stream data is avoided. ScatterBlogs2 [4] takes advantage of the sampling, the topic filtering of documents is carried out in real time to extract the information that users are interested in. These methods can support streaming visualization within a limited range of data, but are difficult to manage massive amounts of data and respond to interaction in real-time. In addition, they pay attention to handling online incoming data, lacking management of entire data in the time domain.

Data Cube. Data cube, as a data management architecture, has gained attractions among researchers in recent years. It is well suited to the handling of multi-dimensional datasets.

There are two basic operations in data cube: **drill-down** and **roll-up**. Drill-down refers to the process of viewing data at a level of increased detail, while roll-up refers to the process of viewing data with decreasing detail. The traversal path in Fig. 2 shows how the data cube responds to queries with the drill-down and roll-up. The aggregates of O_1, O_2, O_3, and O_4 are found after the drill-down can be applied roll-up to produce the result of query. As you can see from figure, data cube will get the aggregate $O_2O_3O_4$ directly instead of O_2 and O_3O_4 during the drill-down, that is why the data cube have well query performance. Queries have

low latencies due to pre-aggregation  , supporting interactive visualization in real-time.

There is a lot of work that use data cubes today [3,11,20,28]. Among all the relevant efforts, Nanocubes [13] is considered to be the baseline. It first takes advantage of the shared links to reduce memory overhead. The query in the temporal domain often requests a period of time rather than a specific time point. Summed-Area table [7] is widely applied to the structure of stored time series. Time Lattice [19] leverages the partial order relation of time to reduce the memory cost but only applies to the temporal domain and does not support spatial data exploration.

To solve the problem of aggregations using a lot of memory, array-based structure Hashedcubes [9,11] achieves low memory usage by sorting multi-dimensional arrays and recording pivots for each dimension. Smartcube [14] supports the creation and deletion of cuboids to support dynamic partial aggregation, and a dynamic adaptive algorithm is designed to maintain the valuable cuboids in the structure. They both introduces a way to adapt to user interaction habits and focus on dimensions of interest to users. In fact, the user's points of interest are variable during the interaction. When a query hits a non-existent cuboid, the Smartcube takes time to drill-down, causing a burst of high latency. Falcon [21], a Linked Visualization of big data. It utilizes data cube to build complete indexes, which is a low-latency solution for the exploration of cold start dataset.

Data cube-based methods have proven themselves to be good at interactive visualization, able to respond to user queries in milliseconds, even with millions or even billions of pieces of Data. This excellent interaction performance is due to the aggregations that are built at the expense of memory. However, they all focus on a static view. When considering dynamically incoming data, they are constrained by longer updating times.

3 Challenges and Motivation

In this section, we briefly point out the shortcomings of the data cube for visualization of streaming data and how do we get inspired to solve these problems.

3.1 Limitations of Data Cube

The data cube uses two key technologies to support real-time interaction. One is to structure data in way to improve the speed of the query path, and the other is to store pre-aggregations to avoid dynamic calculating. As the dataset scale gets larger in recent years, the overhead of data cubes has become the focus of much work. It's hard to implement data cube with both **high retrieval speed** and **low memory overhead**, that we often have to trade off between them. When the visualization task is picky about the data structure and overhead, such as streaming data, meeting challenges.

Firstly, in the previous work, datasets were static and structures were often built only once. Dynamically appended data introduces additional latencies for

data cubes, which is especially problematic for array-based data cubes. Because arrays need to be sorted, it is expensive to resort all existing data every time data arrives. Besides, when the volume of data is large enough, and there are many aggregations stored in the structure, even the tree-based data cubes also significantly degrade performance. Because all involved aggregations also need to be rebuilt during updating.

Secondly, though reducing the number of aggregations alleviates the above problem, too few aggregations will hurt interaction performance. In particular, without pre-aggregations, the tree-based data cubes will drill down to the deeper nodes of the tree to search for results, and the overhead of recursively traversing the tree can be worrying.

As mentioned, it is difficult to solve these two problems simultaneously in streaming data visualization. The key is **how to use fewer aggregations to speed up structure updating while maintaining high query performance**.

3.2 Motivation

In response to any queries immediately, the ideal approach is to calculate all possible aggregations in advance. It inevitably leads to more memory cost and more time spent to build data cubes. Actually, aggregations can be calculated in real time by merging aggregations of child nodes. For example, a query, "How many tweets have been sent?", can be equivalent to the combinations of the following queries: "How many tweets have been sent with Android?" and "How many tweets have been sent with iPhone?", assuming the device can only be one of Android or iPhone.

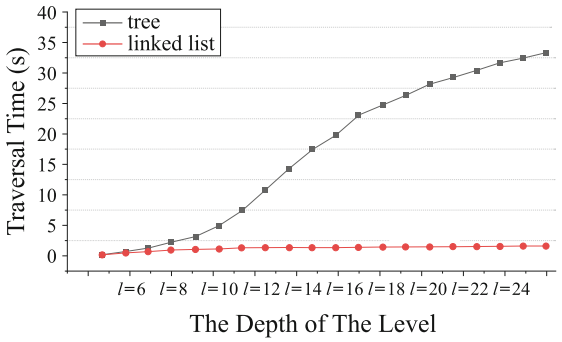

Fig. 3. The comparison of the time traversing all leaves of a tree and a linked list, with increasing depth of the tree. The tree structure consists of 10,000 objects that follow a Gaussian distribution and each node has UP to five branches.

Instead of calculating all possible combinations as aggregations, we can achieve the same effect by merging queries of partial aggregations. It is a universal idea to reduce building time and memory cost in existing studies [9,14].

With a tree-based data cube, multiple queries require the depth-first traversal of the tree. Such recursive operations are expensive. We want to avoid recursive traversal and access aggregations to be merged directly. Intuitively, additional links can be built to increase the retrieval speed, introducing some extra storage overhead but is far less than required for storing many aggregations.

More specifically, a naive solution would be for all nodes to hold links, pointing directly to the aggregations to be merged. Consider a quadtree with N layers, with the number of 4^N leaves for the full tree. The cost for all nodes linking to leaves can be derived as:

$$\sum_{i=1}^{N-2} 4^N = (N-2) \cdot 4^N \tag{1}$$

On the other hand, we propose to link leaves using a linked list like the B+ Tree, so that all nodes with same aggregations can share one linked list. The cost of building such a linked list is $4^N - 1$. Adding the cost of all nodes' links, we get the overall overhead as:

$$\sum_{i=1}^{N-2} \left(4^i\right) + 4^N - 1 = \frac{13}{12} \cdot 4^N - \frac{7}{3} \tag{2}$$

Compared with the naive solution, the traversal speed is the same, while our solution incurs less overhead, with the cost of $\Theta(4^N)$ (Eq. 2) that is smaller than $\Theta(N \cdot 4^N)$ (Eq. 1) for the naive one. As shown in Fig. 3, when the depth of the tree grows large, the cost of accessing linked lists is still small, while accessing the same amount of objects recursively becomes time-consuming.

Based on the observation and motivation, we propose the design and implementation of Linkube that improves retrieval efficiency by linking aggregations with linked lists to reduce tree traversals. Linkube avoids storing many aggregations, which increases update speed and reduces memory overhead. It also speeds up the drill-down, which reduces response time.

4 Design of Linkube

With an observation on the traversal time of existing methods based on trees and linked lists, we are motivated to propose Linkube to accelerate the query process. We expect an approach that query efficiency without building new aggregations.

4.1 Basic Definitions

Attributes in a dataset can be defined with a dimension set $D = \{d_1, d_2, ..., d_n\}$, where $d_i \in D$ refers to a specific attribute. Each object o has a separate label of $d_i(o)$ in dimension d_i.

We index the label value of each dimension. To support the subdivision of dimensions, each dimension can be represented by multiple levels [13]. An object

label value $d_i(o)$ will be denoted as a **chain** $C_i = [l_1, l_2, ..., l_n]$, where the levels satisfy the partial order $l_1 \succcurlyeq l_2 \succcurlyeq ... \succcurlyeq l_n$. We say l_{n-1} is coarser than l_n or that l_n is finer than l_{n-1} if for the same dimension of any two objects $d_i(o), d_i(o')$ have $l_{n-1}(o) = l_{n-1}(o') \Rightarrow l_n(o) = l_n(o')$. The chain can be regarded as the granularity of label value.

Data are stored in structure according to the dimension of each attribute. A dataset can be formulated as **schema** $\{C_1, C_2, C_3, ..., C_n\}$. Following above definitions, **path** of chains $P = \{[l_{11}, l_{12}, l_{13}...], [l_{21}, l_{22}, l_{23}...], ..., [l_{n1}, l_{n2}, l_{n3}...]\}$ refers to the traversal path during the execution of a query, where each chain is defined by the attribute itself.

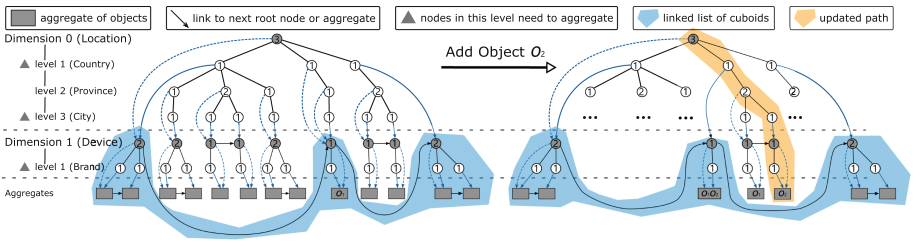

Fig. 4. An illustration of the structure of Linkube and an example of updating a linked list when a new object o_2 is added.

4.2 Linked List-Optimized Structure

As Linkube stores only partial aggregations, queries involving unaggregated nodes need to drill-down. Taking advantage of the linked list, Linkube achieves high query efficiency when accessing unaggregated nodes, by the linking of adjacent nodes that avoids recursively traversing the tree.

Structure Building. Figure 4 illustrates the structure of Linkube, which utilizes shared links to avoid unnecessary node generation and does not calculate all aggregations. The root node can be regarded as the beginning of the aggregation of each dimension. In general, aggregations at the branch node result in the creation of new root nodes in the next dimension. Linkube will chain ↘ the root nodes (the next dimension) as a linked list. When accessing unaggregated nodes, Linkube can perform a drill-down operation by traversing a linked list.

The linked list chains adjacent root nodes of the same dimension. We describe the root node in dimension $i+1$ pointed ↘ (⇘) by the node in dimension i as **content**, which is the first aggregation of the node at the beginning of the arrow. Thus, each node needs to record a number that indicates the number of aggregations to be read in the linked list. As shown in the figure, when querying the aggregation of all objects, such as $\{[all, all, all], ...\}$, Linkube will access the linked list pointed by the node at level 0 in dimension 0 based on the number recorded. The light blue area in the figure represents a linked list of aggregated results, which is the aggregation of three adjacent child nodes linked.

Structure Updating. Since Linkube actually uses lists instead of storing aggregations, it only needs to update the linked list when receiving new data, which is much faster than recursively updating aggregations. Figure 4 shows an example of the structural change when a new object is added to Linkube. The new object generates a new branch at level 2 of dimension 0. Since the nodes of level 2 do not need to maintain aggregation, the number it records and the linked list pointed to by node needs to be updated. The root node created is inserted into the linked list. The node at level 1 of dimension 0 needs to maintain aggregation, pointing to the new aggregation. The previous root node of the subtree with only o_1 as a singleton aggregate has to be removed from the list. As shown in the light blue area , after adding a new object, the structure is rebuilt, and the linked list where the aggregation resides needs to be updated.

Query Engine. Considering queries that involve aggregations which are built with linked-lists, such as a query "How many tweets in the dataset were sent from iPhone?" Since we don't care about the location information of each record, we don't have to drill-down to a specific country or city. The query will be interpreted as the path $\{[all, all, all], [iPhone]\}$. Based on the path $[all, all, all]$, Linkube looks for aggregations of the root node of dimension 0. As shown, find the three aggregations in the linked list according to the number recorded by the root node ③. Since we are interested in iPhone, Linkube will drill-down to search device records of the corresponding brand according to path $[iPhone]$ after three aggregations found.

Linkube significantly speeds up the calculating of aggregations in real-time with almost no aggregation nodes built. For a comparison with the state-of-the-arts on data cube, please refer to Table 1.

Table 1. Comparison of our Linkube with the state-of-the-arts.

	structure	drill-down	aggregation	dataset
Linkube	flexible	ordinal	flexible	spatiotemporal
Smartcube	flexible	recursive	flexible	spatiotemporal
Nanocubes	fixed	recursive	all	spatiotemporal
Hashedcubes	fixed	ordinal	partial	spatiotemporal
Time Lattice	fixed	ordinal	partial	temporal

4.3 Smart Caching Mechanism

In the scenario of streaming data, as the structure is constantly updated, it is inevitable that some queries involve a large amount of data. Even Linkube calculates aggregation efficiently, there are still some queries have high latency. Furthermore, it is typical that only partial aggregations are actually accessed, and only a few ones frequently accessed. Accordingly, We want to build aggregations cache for a few nodes with high response time, increasing the interactive experience without slowing down the updating.

Because of the uncertainty of user behavior, it is difficult to define which aggregations are the most valuable. But we know which aggregation in the past queries were expensive to retrieve. Nodes of the same level have a similar distribution, so nodes with the same level tend to have similar costs. Besides, We notice that aggregations at partial resolution levels are sufficient to support efficient queries, and the aggregated levels can feed back to the parent nodes to reduce the traversal length of the linked list. As such, we introduce an adaptive caching mechanism during the query process to help find which levels of aggregations are worth being built and reserved.

Initialization. An initial threshold k_d for each dimension is given, which is the number of aggregated levels where aggregations of all nodes will be calculated and maintained. l_d indicates the index of levels subdivided in dimension d. During the building process, aggregations are generated at every level by checking:

$$(|l_d| - l_d)\, Mod \left\lceil \frac{|l_d|}{k_d} \right\rceil = 0 \tag{3}$$

where $|l_d|$ is the number of levels that dimension d contains. For nodes at other levels, linked lists are applied to link aggregations.

Updating the Utility. A set $U = \{u_1, u_2, ..., u_d\}$ records each level's utility in each dimension, where $u_d = \{r_{d1}, r_{d2}, ..., r_{dl}\}$. As the utility record of level l, r_{dl} indicates the benefit of maintaining aggregations for nodes at level l. Linkube finds levels with the greatest utility in the same dimension and stores their aggregations. When the query arrives, r_{dl} is updated as follows:

$$r_{dl} = r_{dl} + l_d \times n \tag{4}$$

where n is the number of nodes accessed and may change next time after updating. That is, when new aggregations are calculated and stored for level l', all the parent nodes in levels $l_d (l_d < l')$ will adjust the linked list for their root nodes to partially use the aggregations already stored.

In this way, the latencies for these upper level nodes are shortened and their utilities would also be decreased according to Eq. 4. We also clean up the global record U to avoid possible overflows. Whenever the maximum value of r_{dl} reaches the threshold that we set (half of the MAX INTEGER value), if the minimum value of u_d is 0 then $r_{dl} = \frac{r_{dl}}{2}$ otherwise $r_{dl} = r_{dl} - Min(u_d)$. This operation still ensures the validity of the utility estimate because the existing aggregation state is maintained.

Updating the Aggregation. After u_d gets updated, if r_{dl} is the top k_d terms of set u_d and nodes at level l_d have no aggregations, aggregations for all nodes at level l_d will be calculated. Simultaneously, aggregations of nodes beyond the top k_d are removed from memory and links are rebuilt by pointing to linked lists.

The updating process can be regarded as (1) recording the query overhead of the specific level and (2) keeping aggregations of the more expensive ones stored in memory. Updating process can be done in parallel and it's done very quickly, thanks to existing linked lists that speed up drill-down.

Algorithm 1. REMAKECHAIN(*stack*, *content*)

```
1: child ← POP(stack), node ← POP(stack), pop_size ← 1
2: while node does not need to aggregate and stack is not empty do
3:     if child is LEFTMOSTCHILD(node) then
4:         child ← node, node ← POP(stack), pop_size ← pop_size + 1
5:     else
6:         left_child ← LEFTSIBLING(child)
7:         if ISCHILDSHARED(node,left_child) then
8:             origin_node ← node, copy ← SHALLOWCOPOY(left_child)
9:             REPLACECHILD(node,copy), PUSH(stack,node), PUSH(stack,copy)
10:            node ← copy
11:            for k = 1 to pop_size do
12:                copy ← SHALLOWCOPY(RIGHTMOSTCHILD(node))
13:                REPLACECHILD(node,copy), PUSH(stack,copy), node ← copy
14:            end for
15:            content_copy ← SHALLOWCOPY(CONTENT(node))
16:            INSERTNEXTNODE(content_copy,content)
17:            SETPROPERCONTENT(node,content_copy)
18:            REMAKECHAIN(stack,content)
19:            POP(stack)
20:            while true do
21:                last_node ← POP(stack)
22:                SETSHAREDCONTENT(last_node,CONTENT(LEFTMOSTCHILD(last_node)))
23:                if last_node is origin_node then
24:                    break
25:                end if
26:            end while
27:        else
28:            node ← left_child
29:            for k = 1 to pop_size do
30:                node ← RIGHTMOSTCHILD(node)
31:            end for
32:            INSERTNEXTNODE(CONTENT(node),content)
33:        end if
34:    end if
35: end while
```

5 Algorithm

The key technologies addressed in Linkube are how to build the linked list and how to alter the aggregation state of nodes. We introduce the design details with the pseudo-code in this section.

5.1 Building the Linked List

The building process of Linkube is the insertion of data, allowing data to be added at run-time. When a new data is added, there are two cases: (1) inserted into an existing aggregate; (2) inserted into a new branch created. There is no need to modify any structure in the first case, and the existing structure is still correct. In the second case, a new branch means a new aggregation of the node is created. Linkube needs to update the linked list to ensure the correctness of the structure.

As illustrated in Algorithm 1, the function ***RemakeChain*** is designed to insert root nodes into linked lists, which is called whenever the structure is changed. *content* and a *stack* of path to access node are given as input. The main idea of the algorithm is for all nodes on the updated path to find the

nearest branch on the path while the left sibling node *left_child* exists (line 3–4). Then, it looks for the deepest, rightmost child of *left_child* (line 28–31). The content is inserted after the content of the rightmost child of *left_child* (line 34). Note that some of the nodes may be *shared_node* which will be replaced by cloned nodes during the traversal (line 7–26).

Algorithm 2. UpdateAggregation(*stack*)

```
1: node ← Pop(stack)
2: if node needs to aggregate then
3:     SetCount(root,1)
4:     if node has single child with count is 1 then
5:         SetSharedContent(node,Content(Child(node)))
6:     else if [c_1, c_2, ..., c_n] is not empty then
7:         SetProperContent(node,AggregateContent([c_1, c_2, ..., c_n]))
8:     end if
9:     RemakeChain(Push(Copy(stack),node),Content(node))
10: else
11:     UpdateCount(node)
12:     SetSharedContent(node,Content(LeftmostChild(node)))
13:     RemakeChain(Push(Copy(stack),node),Content(node))
14: end if
15: while stack is not empty do
16:     node ← Pop(stack)
17:     if node need to aggregate then
18:         break
19:     end if
20:     UpdateCount(node)
21:     SetSharedContent(node,Content(LeftmostChild(node)))
22: end while
```

5.2 Aggregation State Updating

Updating the aggregation state is a process of depth traversal of the tree. Specifically, for ***UpdateAggregation*** in Algorithm 2, a stack that stores nodes of a path is used as input, and the node at the top of the stack is the one that needs to update the aggregation state. For nodes that need to be aggregated, the new content is aggregated and inserted into the linked list (line 3–9). Otherwise, Linkube will update the recorded number and set aggregation as the content of the leftmost child node (line 11–13).

6 Evaluation

In this section, we evaluate our Linkube, in comparison with existing visualization methods on four public-available datasets.

6.1 Implementation

We implement Linkube as a web service, using a simple client-server architecture. It exposes the updating API via Socket for data producer and the querying API via HTTP for user. For intuitively displaying Linkube, we implement the

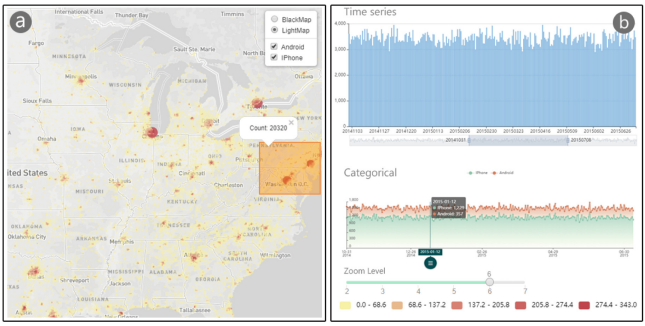

Fig. 5. The prototype of the Linkube. (a) A heatmap case with a dataset of 5 million tweets. (b) A large scale bar chart shows time series, filtering for the time range. The categorical chart of the twitter dataset provides intuitive comparison of device type.

prototype of Linkube using a simple web page as shown in Fig. 5. Users can visually access massive data of different dimensions in real time with Linkube, supporting general interactions (Zooming, Panning, Brushing, and Linking). We make a video for to illustrates our system, which provides more details (link: https://www.youtube.com/watch?v=V8IEywu9qHc).

6.2 Experimental Setup

The four datasets used in our experiments are **BrightKite**, **Flight**, **Twitter**, and **Taxi**. The contained data covers spatial, temporal, and specific categorical dimensions. The amount of data varies from millions to hundred of millions.

- **BrightKite**. BrightKite includes over **4.5 million** items from April 2008 to October 2010. BrightKite is a former location-based social network that recorded location and time when users checked in. [6].
- **Flight**. Airline On-Time Performance covers over **121 million** flights data in a 20-year period. The records include the airport of a flight, departure and arrival time, carrier, delay and other related information. [10]
- **Twitter**. Twitter contains over **5.5 million** tweets between year 2014 and 2015. Using Twitter API to collect the information about tweets, we collected data on the location, time, and device type of tweets sent over time.
- **Taxi**. Taxi includes over **17.6 million** records, which is a sample of T-Drive taxi trajectory dataset which was generated by over 10,000 taxis in a period of one week in Beijing [29,30].

The experiments are performed on an Intel Core i7-9750 CPU with 32 GB RAM. We choose Nanocubes, Smartcube and Hashedcubes as our baselines for different evaluations. We implement them in Java for a fair comparison. Smartcube is set to build aggregations when $\frac{S_{C_{bsc}}}{S_C}$ is greater than the threshold value of 1.5. To evaluate the scalability of Linkube with different utility

strategies, we test the performance of different k_d values. Linkube is considered equivalent to Nanocubes when k_d is set to $|l_d|$. For experiments without k_d legend, we set k_d as 2 for all dimensions.

6.3 Effectiveness of Linkube

We first evaluated the performance of Linkube, focusing on the two main latency causing processes in visualization, building and interaction. Especially in the case of streaming data, the major computation is updates of data cube and queries.

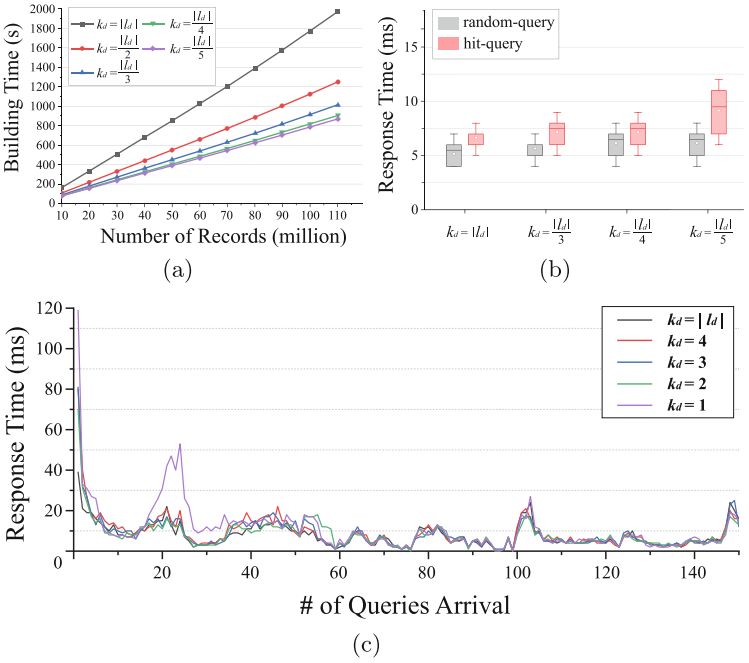

Fig. 6. Performance on the building time and response time. (a) Linkube's building time on the Flight dataset. (b) The response time of 10,000 queries on the Twitter dataset. (c) The response time of Linkube with the arrival of queries.

Building Time. To demonstrate the experimental results of building Linkube, we measured the building time of Linkube with the largest dataset, Flight. Linkube has different construction costs for different values of k_d, indicating that the structure of Linkube is scalable.

As shown in Fig. 6(a), the building time of Linkube is much lower than that of Nanocubes, effectively reducing building time by more than 45%. The building time increases steadily because data objects are inserted during the building process, which can be viewed as a linear function. The greater k_d is set, the more

time it takes to build the structure. When k_d is less than $\frac{|l_d|}{3}$, the difference in drop rate becomes less obvious and tends to be stable.

Interaction. We measure the performance of random-query and hit-query in the experiment, elaborated as follows. Random-query indicates that all query inputs are random, requesting aggregated values for any data depth and distribution. In this form, the query engine returns immediately when it finds that the required aggregation does not exist. For example, suppose we query for the tweets sent from area A with the Android device during March. When the query engine traverses the tree and finds that the node with label A does not exist, it does not need to continue with the deeper dimension (device type and period of time). Hit-query refers to valid queries with the aggregation certain to be found in Linkube. Random-query is more realistic, while hit-query is better to characterize the cost of traversing the tree.

Figure 6(b) shows the response time on the Twitter dataset, corresponding to 10,000 aggregates of both random-queries and hit-queries. Linkube optimizes the drill-down process with sequential access. Linkube's response time increases as k_d decreases, but the increased overhead is acceptable, thus maintaining high query efficiency with fewer aggregation nodes. As shown, the hit-query latencies fluctuate wildly, but random-query performance is similar to Nanocubes which is closer to the real query case, and its. Since all aggregations have been calculated, Nanocubes shows the best and most stable performance of response. We can consider Linkube to have near-optimal query performance.

Caching Mechanism. To evaluate the caching mechanism of Linkube, we compare the response time when a set of queries arrived. Figure 6(c) illustrate the performance with caching mechanism in different settings and domains. In this experiment, 1ms is the lowest possible resolution of the used timing framework.

The result shows the response time with the same queries arrival. We note that partial levels of aggregates can respond to arbitrary queries quickly. The larger the value of k_d, the more levels are aggregated. Upon query arrival, the aggregation state of each level is updated to achieve the best utility. When k_d is set greater than 3, Linkube has almost the same performance as Nanocubes. Even in the early stages, Linkube is able to respond quickly, due to the linked lists built. Thanks to the utility model, Linkube's performance is stable when the state of aggregations converges, and has similar performance to Nanocubes.

6.4 Performance on Streaming Data

The study of streaming data visualization lacks qualitative or quantitative formal evaluation methods. Due to differences in the dataset and the visualization task, the evaluation varies for different scenarios. In order to understand the changes of the data cube in different streaming data, we first test the performance of each method against data with different attributes and granularity. Similar to Fig. 3, data conforming to Gaussian distribution is generated to simulate datasets of different dimensions and levels. We assume that there are 50 times of incoming

data, each with a volume ranging from 500 to 1000, and 1000 queries to respond to after each update.

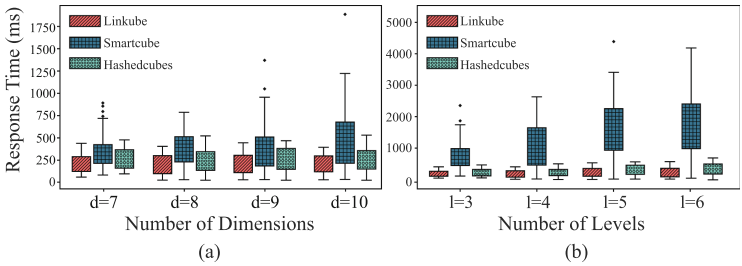

Fig. 7. The influence of different dimensions and levels on the data cube. (a) The average response time of each method in different dimension settings. (b) The average response time of each method in different levels settings.

Figure 7(a) shows the impact of different dimension settings. As datasets tend to be higher dimensional, it is inevitable that the overhead of calculating aggregations and traversing during queries increases. The result demonstrated that Linkube is stable and performs better. Hashedcubes is also stable, but takes much time to update as it needs to rebuild the entire structure with each update. Smartcube is more sensitive, and its performance degrades more significantly as the number of dimension increase. Through analysis, it is found that with the update of streaming data, new aggregations are constantly constructed, and the more aggregations there are, the higher the update cost will be when new data arrives. In addition, Smartcube's updating strategy requires traversing a large number of nodes, which is another reason for the high response time. Figure 7(b) shows the impact of different dimensions set. When l is larger, the query granularity supported by this dimension is larger. Basically the result is the same as the dimension evaluation. The Smartcube performance degrades more at finer granularity (higher l). Nanocubes builds too many aggregations that runs out of memory to complete the test. Nanocubes performs worse after building a large number of aggregations, so that it takes several times than the other methods.

As shown in Fig. 8, we show the update process when $d = 7$. The response time of Nanocubes is much higher than that of other methods. Although Nanocubes has a low query time, the number of aggregations has a significant impact on performance. Similarly, Smartcube is also affected by the number of aggregates and has a larger latency fluctuation, even if Smartcube has a heuristic update strategy. In contrast, Linkube and Hashedcubes are more stable.

To test the performance of our work in a real-world scenario, we adopted a stream dataset of Beijing traffic and Tweets sending. The results are shown in Fig. 9. Intuitively, Linkube performs better than the others. Especially in the Taxi dataset (figure a), Linkube's novel structure supports it to load the

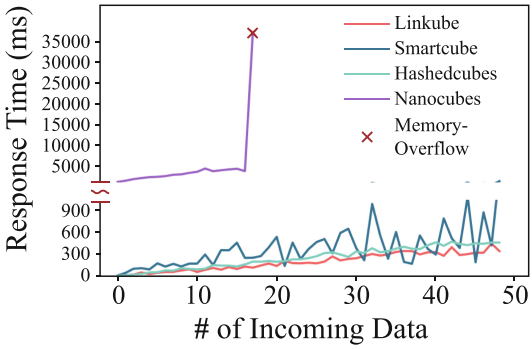

Fig. 8. The response time during stream data update process when dimension is 7.

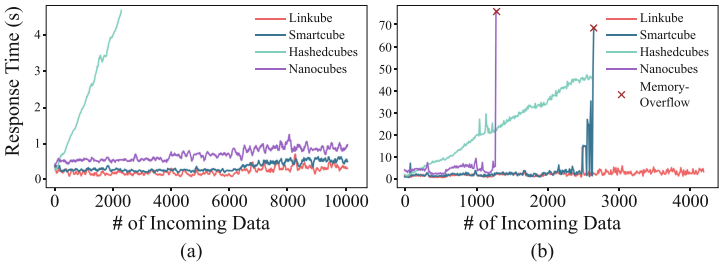

Fig. 9. Performance comparison of different methods under real data. As data arrives, the response time of updating of the structure and queries. (a) The response time of the Taxi dataset. (b) The response time of the Twitter dataset.

entire stream completely, and is the only method to complete the test. Due to hardware limitations, other comparison methods cannot complete the test. Both Smartcube and Nanocubes run out of memory at some point due to building too many aggregations, which is unacceptable for scenarios with large amounts of data. Prior to this, SmartCube performance was close to Linkube, but the response time increased more as the incoming data updated. In contrast, Hashedcubes generally does not have overflow problems because it uses arrays to build finite aggregations. However, it brings up worse problems. When the data volume is large enough, the delay caused by each update is too long. The Hashedcubes performs poorly in the actual streaming data scenario compared to the previous experiment. Due to the higher dimensional of the real spatiotemporal dataset, the data distribution is more dispersed, greatly increasing the rebuilding cost of Hashedcubes. Hence, in both experiments, we did not measure the complete time spent by Hashedcubes.

7 Conclusion

To address the problem of streaming data processing brought by massive devices, we propose Linkube, an efficient and intelligible system with corresponding maintaining algorithms for visualization of streaming data. Briefly, Linkube maintains few knowledge units and significantly reduces the time to update the structure. In order to ensure the efficiency of query response, a novel linked list is applied to replace the dummy depth-first searching strategy for accelerated traversing. For better interactive experience, a smart caching mechanism is designed according to a utility model to determine whether to keep aggregations in memory for flexible resource usage. We implement it as a web-based interactive visualization tool to answer queries from real-world datasets.

For future work, data parallel processing methods in distributed system paradigm can be introduced into Linkube to support the visualization of larger and more complex datasets. The memory cost can be further reduced by utilizing simplifying coding. Machine learning techniques can be integrated into Linkube for user queries prediction and helping decision making, thereby providing more intelligent and efficient data visualization experience.

Acknowledgment. This work is supported by the National Natural Science Foundation of China (62172155, 62072465, 62102425), the Science and Technology Innovation Program of Hunan Province (2021RC2071).

References

1. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2002, pp. 1–16. Association for Computing Machinery, New York (2002). https://doi.org/10.1145/543613.543615
2. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Informatica **1**, 173–189 (1972). https://doi.org/10.1007/BF00288683
3. Beyer, K.S., Ramakrishnan, R.: Bottom-up computation of sparse and iceberg cubes. In: Delis, A., Faloutsos, C., Ghandeharizadeh, S. (eds.) SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, 1–3 June 1999, Philadelphia, Pennsylvania, USA, pp. 359–370. ACM Press (1999). https://doi.org/10.1145/304182.304214
4. Bosch, H., et al.: Scatterblogs2: real-time monitoring of microblog messages through user-guided filtering. IEEE Trans. Vis. Comput. Graph. **19**(12), 2022–2031 (2013). https://doi.org/10.1109/TVCG.2013.186
5. Cao, N., Lin, Y.R., Sun, X., Lazer, D., Liu, S., Qu, H.: Whisper: tracing the spatiotemporal process of information diffusion in real time. IEEE Trans. Visual Comput. Graphics **18**(12), 2649–2658 (2012). https://doi.org/10.1109/TVCG.2012.291
6. Cho, E., Myers, S.A., Leskovec, J.: Friendship and mobility: user movement in location-based social networks. In: Apté, C., Ghosh, J., Smyth, P. (eds.) Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, 21–24 August 2011, pp. 1082–1090. ACM (2011). https://doi.org/10.1145/2020408.2020579

7. Crow, F.C.: Summed-area tables for texture mapping. In: Christiansen, H. (ed.) Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, 23–27 July 1984, pp. 207–212. ACM (1984). https://doi.org/10.1145/800031.808600
8. Dasgupta, A., Arendt, D.L., Franklin, L.R., Wong, P.C., Cook, K.A.: Human factors in streaming data analysis: challenges and opportunities for information visualization. Comput. Graph. Forum **37**(1), 254–272 (2018). https://doi.org/10.1111/cgf.13264. https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13264
9. de Lara Pahins, C.A., Stephens, S.A., Scheidegger, C., Comba, J.L.D.: Hashedcubes: simple, low memory, real-time visual exploration of big data. IEEE Trans. Vis. Comput. Graph. **23**(1), 671–680 (2017). https://doi.org/10.1109/TVCG.2016.2598624
10. Levine, R.A., Sampson, E., Lee, T.C.M.: Journal of computational and graphical statistics. WIREs Comput. Stat. **6**(4), 233–239 (2014). https://doi.org/10.1002/wics.1307
11. Li, M., Choudhury, F.M., Bao, Z., Samet, H., Sellis, T.: Concavecubes: supporting cluster-based geographical visualization in large data scale. Comput. Graph. Forum **37**(3), 217–228 (2018). https://doi.org/10.1111/cgf.13414
12. Li, Q., Wei, X., Lin, H., Liu, Y., Chen, T., Ma, X.: Inspecting the running process of horizontal federated learning via visual analytics. IEEE Trans. Visual. Comput. Graphics **28**(12), 4085–4100 (2021)
13. Lins, L.D., Klosowski, J.T., Scheidegger, C.E.: Nanocubes for real-time exploration of spatiotemporal datasets. IEEE Trans. Vis. Comput. Graph. **19**(12), 2456–2465 (2013). https://doi.org/10.1109/TVCG.2013.179
14. Liu, C., Wu, C., Shao, H., Yuan, X.: Smartcube: an adaptive data management architecture for the real-time visualization of spatiotemporal datasets. IEEE Trans. Vis. Comput. Graph. **26**(1), 790–799 (2020). https://doi.org/10.1109/TVCG.2019.2934434
15. Liu, G., Zhang, Q., Cao, Y., Tian, G., Ji, Z.: Online human action recognition with spatial and temporal skeleton features using a distributed camera network. Int. J. Intell. Syst. **36**(12), 7389–7411 (2021). https://doi.org/10.1002/int.22591. https://onlinelibrary.wiley.com/doi/abs/10.1002/int.22591
16. Liu, Z., Heer, J.: The effects of interactive latency on exploratory visual analysis. IEEE Trans. Visual Comput. Graphics **20**(12), 2122–2131 (2014)
17. Mansmann, F., Krstajic, M., Fischer, F., Bertini, E.: StreamSqueeze: a dynamic stream visualization for monitoring of event data. In: Wong, P.C., et al. (eds.) Visualization and Data Analysis 2012, vol. 8294, pp. 13–24. International Society for Optics and Photonics, SPIE (2012). https://doi.org/10.1117/12.912372
18. Martín, Y., Li, Z., Cutter, S.L.: Leveraging twitter to gauge evacuation compliance: spatiotemporal analysis of Hurricane Matthew. PLoS ONE **12**(7), 1–22 (2017). https://doi.org/10.1371/journal.pone.0181701
19. Miranda, F., et al.: Time lattice: a data structure for the interactive visual analysis of large time series. Comput. Graph. Forum **37**(3), 23–35 (2018). https://doi.org/10.1111/cgf.13398
20. Miranda, F., Lins, L.D., Klosowski, J.T., Silva, C.T.: Topkube: a rank-aware data cube for real-time exploration of spatiotemporal data. IEEE Trans. Vis. Comput. Graph. **24**(3), 1394–1407 (2018). https://doi.org/10.1109/TVCG.2017.2671341
21. Moritz, D., Howe, B., Heer, J.: Falcon: balancing interactive latency and resolution sensitivity for scalable linked visualizations. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, 04–09 May 2019, p. 694 (2019). https://doi.org/10.1145/3290605.3300924

22. Moshtaghi, M., Bezdek, J.C., Erfani, S.M., Leckie, C., Bailey, J.: Online cluster validity indices for performance monitoring of streaming data clustering. Int. J. Intell. Syst. **34**(4), 541–563 (2019). https://doi.org/10.1002/int.22064. https://onlinelibrary.wiley.com/doi/abs/10.1002/int.22064
23. Ponciano, J.R., Linhares, C.D.G., Rocha, L.E.C., Faria, E.R., Travençolo, B.A.N.: A streaming edge sampling method for network visualization. Knowl. Inf. Syst. **63**(7), 1717–1743 (2021). https://doi.org/10.1007/s10115-021-01571-7
24. Sacha, D., et al.: What you see is what you can change: human-centered machine learning by interactive visualization. Neurocomputing **268**, 164–175 (2017)
25. Tableau Software: Tableau-interactive-visualization-examples (2003). https://www.tableau.com/learn/articles/interactive-map-and-data-visualization-examples
26. Steed, C.A., et al.: Web-based visual analytics for extreme scale climate science. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 383–392 (2014). https://doi.org/10.1109/BigData.2014.7004255
27. Tang, J., Liu, J., Zhang, M., Mei, Q.: Visualizing large-scale and high-dimensional data. In: Proceedings of the 25th International Conference on World Wide Web, WWW 2016, pp. 287–297. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2016). https://doi.org/10.1145/2872427.2883041
28. Wang, Z., Ferreira, N., Wei, Y., Bhaskar, A.S., Scheidegger, C.: Gaussian cubes: real-time modeling for visual exploration of large multidimensional datasets. IEEE Trans. Vis. Comput. Graph. **23**(1), 681–690 (2017). https://doi.org/10.1109/TVCG.2016.2598694
29. Zheng, Y., Xie, X., Ma, W.Y.: Understanding mobility based on GPS data. In: Proceedings of the 10th ACM Conference on Ubiquitous Computing (Ubicomp 2008) (2008). https://www.microsoft.com/en-us/research/publication/understanding-mobility-based-on-gps-data/
30. Zheng, Y., Xie, X., Ma, W.Y.: Mining interesting locations and travel sequences from GPS trajectories. In: Proceedings of International conference on World Wide Web 2009 (2009). https://www.microsoft.com/en-us/research/publication/mining-interesting-locations-and-travel-sequences-from-gps-trajectories/