SmartCube: An Adaptive Data Management Architecture for the Real-Time Visualization of Spatiotemporal Datasets

Can Liu, Cong Wu, Hanning Shao, and Xiaoru Yuan, *Senior Member, IEEE*

Abstract—Interactive visualization and exploration of large spatiotemporal data sets is difficult without carefully-designed data preprocessing and management tools. We propose a novel architecture for spatiotemporal data management. The architecture can dynamically update itself based on user queries. Datasets is stored in a tree-like structure to support memory sharing among cuboids in a logical structure of data cubes. An update mechanism is designed to create or remove cuboids on it, according to the analysis of the user queries, with the consideration of memory size limitation. Data structure is dynamically optimized according to different user queries. During a query process, user queries are recorded to predict the performance increment of the new cuboid. The creation or deletion of a cuboid is determined by performance increment. Experiment results show that our prototype system deliveries good performance towards user queries on different spatiotemporal datasets, which costing small memory size with comparable performance compared with other state-of-the-art algorithms.

Index Terms—data management, spatial-temporal data

1 Introduction

Real-time interactive visualization of large spatiotemporal data is difficult without an efficient way to aggregate the original data sets. Often data is pre-aggregated over certain constraints in interactive scenarios. Data Cube [10] pre-computes and stores all aggregates of datasets on every possible combination of dimensions. An aggregation query can immediately get the results from the pre-computed data cubes. Many visualization systems organize data structures based on data cubes for real-time response.

In a data cube, the aggregation of certain dimensions combination is called a **cuboid**, which stores all the possible aggregates on a combination of several dimensions. For example, the data cube of the dataset with dimensions $D = \{country, language, device\}$ has 2^3 cuboids. One of the cuboids is $\{country, language, all\}$, which stores the aggregates of dimension $\{country, language\}$, i.e., all the possible values in the $country \times language$. The memory usage of the data cube grows exponentially as the number of dimension increases. A series of works have been done to reduce memory usage in data management with data cubes. One of those solutions is to store only a subset of the cuboids, while omitting other rarely-visited cuboids.

For example, TimeLattice [21] introduces a method to reduce the number of pre-computed cuboids by only storing cuboids with partial order relations along temporal dimensions, such as cuboids with dimension combination $\{year, month, day, hour\}$, $\{year, month, day\}$, $\{year, month\}$, and $\{year\}$. Another method to reduce space is to share memory among cuboids as much as possible. Nanocubes [17], one of such algorithms, builds a tree-like structure to store a data cube, in which cuboids share nodes when possible.

However, both approaches only improve on a limited range of data and tasks. On the one hand, in data management, nanocube is difficult to organize the dense time series data with many constraints in the temporal dimension, for memory usage grows rapidly when the number of dimensions increases. The experiment in TimeLattice [21] indicated that the memory usage of nanocube was over 100 times of TimeLattice in the dense time series. On the other hand, TimeLattice cannot handle queries other than time series.

These algorithms cannot adapt to various datasets mainly because of the diversity in datasets and tasks. Different spatiotemporal datasets have different dimension properties. For example, some time dimensions have partial order relations while some other dimensions are independent. These properties may lead to different performance in memory and time usage. Given a new dataset, the visualization system designers need to carefully choose the suitable algorithm for the query of the visualization tasks.

Apart from the low latency requirement, there are some resource limitations on the hardware (e.g., memory). For large data management, the most significant resource limitation is the memory size, which will affect the performance when the data I/O in the disk is needed. Considering the diversity in datasets and tasks, the limits in memory, and the interactive requirement, ideally, we need better data structures or management algorithms to handle different datasets and tasks scenarios with lower memory usage and high interactive performance.

To meet these requirements, this paper proposes an adaptive spatiotemporal data management framework, SmartCube, which can update itself according to user queries. A user query can be translated to data aggregates on a data structure, which can be fit into the specific cuboids. For example, when a user wants to explore the aggregated value of each hour in 24 hours of the different days of a week, the cuboid with the $day_{week} \times hour$ dimension can be used for this query. The diversity in the query pattern leads to different importance in the cuboids, which guides the SmartCube to manage the cuboids by keeping the valuable cuboid and omitting the trivial ones to save the memory. SmartCube changes its structure by operating the creation and removing in the cuboid to reach a low latency on queries. To make an adaptive data structure framework, the machine learning model can be combined to enhance the performance. We show the possibility of deep learning models to determine cuboids that need to be initialized at the initialization process. We build an LSTM-model that takes the statistic property of the data set as input and outputs the information of the initialized cuboids at the beginning.

We conduct several experiments on different data sets under different query tasks. Compared with some state-of-the-art algorithms, SmartCube requires much smaller memory and faster performance in most scenarios.

The contributions of this work are as follows.

1. We introduced SmartCube, a data structure adaptive to different type of spatial-temporal datasets and query tasks by operating the cuboids during query process;

• *Can Liu, Cong Wu, Hanning Shao, and Xiaoru Yuan are with Key Laboratory of Machine Perception (Ministry of Education), School of EECS, Peking University. E-mail: {can.liu, wucs, hanning.shao, xiaoru.yuan}@pku.edu.cn. Xiaoru Yuan is also with National Engineering Laboratory for Big Data Analysis and Application, Peking University. Xiaoru Yuan is the corresponding author.*

Manuscript received 31 Mar. 2019; accepted 1 Aug. 2019.
Date of publication 16 Aug. 2019; date of current version 20 Oct. 2019.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TVCG.2019.2934434

2. We built an update algorithm that can create and delete cuboids in SmartCube dynamically

3. We proposed the algorithm for real-time query in the dynamic data structure, which can change its query route according to the structure.

2 Related Work

High efficient data management is an important topic for visualization [11, 29, 30, 36]. The large size of datasets and the low-latency visualization require the data structure to be efficient. The real-time interaction with the system can affect the exploration. For example, as Liu and Heer observed [18], a delay of 500 ms harms users' explorations.

2.1 Efficient Data Management for Visualization

Several methods were proposed to improve the response time to support real-time visualizations. Some increase the resource for data management. For example, VisReduce [13] by Im et al. computes visualization results in a distributed fashion using a MapReduce [6] algorithm and data compression. GPU acceleration is also used [19, 20] for higher performance. However, increasing the calculation resource can not totally solve the problem.

If some precision loss is allowed, approximate techniques [3, 9] could be taken. BlinkDB [1] allows interactive queries over extremely large datasets with approximate results by sampling. Stolper et al. presented a framework for visual analysis progressively [28]. Online aggregation [12] is a method to provide estimated results for the user with low latency and then increase the accuracy at the process of waiting for the query results. Wang et al. proposed NNCubes [32]to train a deep neural network to synthesize the query answer to support the visualizations.

Tree structure has a wide range of applications in data management. The partitioning in spatial data often uses the KD-tree method to discretize the data space using binary encoding [4, 35]. STIG [8] combines CPU and GPU for parallel processing based on traditional KD-tree. The Quad-tree uses a fixed spatial partitioning which divides the block into four sub-blocks in the next level [25]. There are other structures, such as R-tree [31], STR-tree [7] and KDB-Tree [24]. The methods above mainly focus on spatial dimension management.

Besides, to predict the users' actions is also a choice. Battle et al. proposed Forecache [2] to predict user actions and loads related datasets before the query to improve system performance.

2.2 Data Cubes-Based Method

In spatiotemporal data, the time has multiple periodic dimensions, and its analysis tasks need to be queried under different combinations of dimensions; while spatial distribution faces huge resolution challenges, and spatial dimensions may need to be divided into more detailed dimensions for indexing. In the analysis of multi-dimensional spatiotemporal data, the content is often needed to be aggregated over several dimensions. The data cube [10, 27, 34] solves this need by calculating data values for any combination of dimensions to support rapid exploration analysis, and it can lead to extremely low latency. Data cubes have good query performances, but computing all data cubes requires huge memory cost. As the number of dimensions increases, the memory required explodes exponentially.

There are two kinds of methods to reduce the usage of memory. One is to share memory between cuboids. Lins et al. proposed a tree-shaped data cube, Nanocubes [17], in which all cuboids are shared in a single tree-like structure. They save the memory by sharing nodes and create nodes only when necessary. When the original data is sparsely distributed, there will be more shared nodes. However, this method does not essentially solve the problem of the memory of the data cube explosion as the number of dimensions increases. The main difference between Nanocubes and previously published sparse coalesced data cubes such as Dwarf cubes [26] is in the design of aggregations across spatiotemporal dimensions.

The other way is to store only a part of the cuboids. Data cubes have different optimizations in spatiotemporal data sets with different characteristics. For a dataset biased toward the time dimension, Miranda et al. proposed TimeLattice [21] by setting several dimensions, such as year, month, and day, in the time grid without pre-calculating the combination of all dimensions.

The memory footprint of its data structure is very small. For queries that are not in its stored cuboids, the query can be implemented by DRILL DOWN and ROLL UP operations. Compared with Nanocubes, Hashedcubes [23] reduced memory usage to a greater extent. It uses a linear array to store the data and calculates the bounds of each dimension based on the value of the node. However, when the query does not satisfy the order of the indices, it needs to calculate the result on the fly. In the management of biased spatial data, unlike the storage of geographic grid quadtrees, Li et al. proposed a cluster-based data cube based on semantics. The structure can represent the concept of "country-province-county" [16]. Liu et al. decomposed the data cube into a set of 3D and 4D subcubes and data slices to reduce the number of combination of dimensions, and the data slice allows it to take advantage of parallel processing on the GPU [19]. However, it does not change the time complexity.

In addition to pure data cubes, there are extensions for specific tasks in visualization analysis. Gaussian Cubes proposed by Wang et al. [33] are extensions to data cubes. In addition to the functions supported by traditional data cubes, it adds some modeling variables and pre-calculates enough statistical data, such as Gaussian distribution, ordinary least squares, and PCA, to accommodate some linear models and provide more data exploration. TopKube [22] is also an extension to Nanocubes by ranking objects and answering top-k queries to find potential trends. Li et al. proposed SemanticsSpace-Time Cube [15] by considering the text's semantic to help analyze the text-related spatiotemporal scenarios.

Compared with these works, especially Nanocubes, our method does not pre-compute all cuboids in the beginning. Instead, we choose some valuable cuboids based on some rules such as statistics of the data structures generated by data sets and querying frequency. Then, we reduce memory usage, and improve performance with more cuboids that change adaptively.

3 Method

We propose SmartCube, an adaptive data structure for spatiotemporal data sets, to reach low-latency with less memory cost for different spatiotemporal datasets.

3.1 Method Overview

Rather than querying on the dataset to get detailed items, real-time visualization often requests aggregated values, such as *count*, *sum*, *minimum*, *maximum*, and *average*. Data Cube [10] is a structure that stores all the possible aggregates of the dataset.

As the number of dimensions increases, the memory usage of the data cube grows exponentially. The number of cuboids is 2^d if the number of dimensions is d. The cuboids in a data cube are not equally important, since some of them are used more frequently than others. Those being hit by query frequently are the valuable cuboids. Those less used consume a considerable amount of memory while devoting nearly nothing to improve the query efficiency. SmartCube is designed to be able to manage the cuboids it stores during the query process, i.e., it may create or delete cuboids dynamically when queries hit or miss for certain cuboids. As a result, the flexible data structure maintains efficiency by only storing the most valuable cuboids and removing those less valuable cuboids.

To reduce the memory usage, SmartCube stores all cuboids in a uniform hierarchical structure to support sharing of the nodes of cuboids. We develop an updating algorithm on the tree by creating and deleting cuboids dynamically. To query the structure based on the existed pre-computed cuboids, we develop a query engine that can choose the most efficient route in the tree to reach high performance.

Figure 1 shows the workflow of SmartCube. The workflow can be divided into an adaptive data structure part and supported algo-

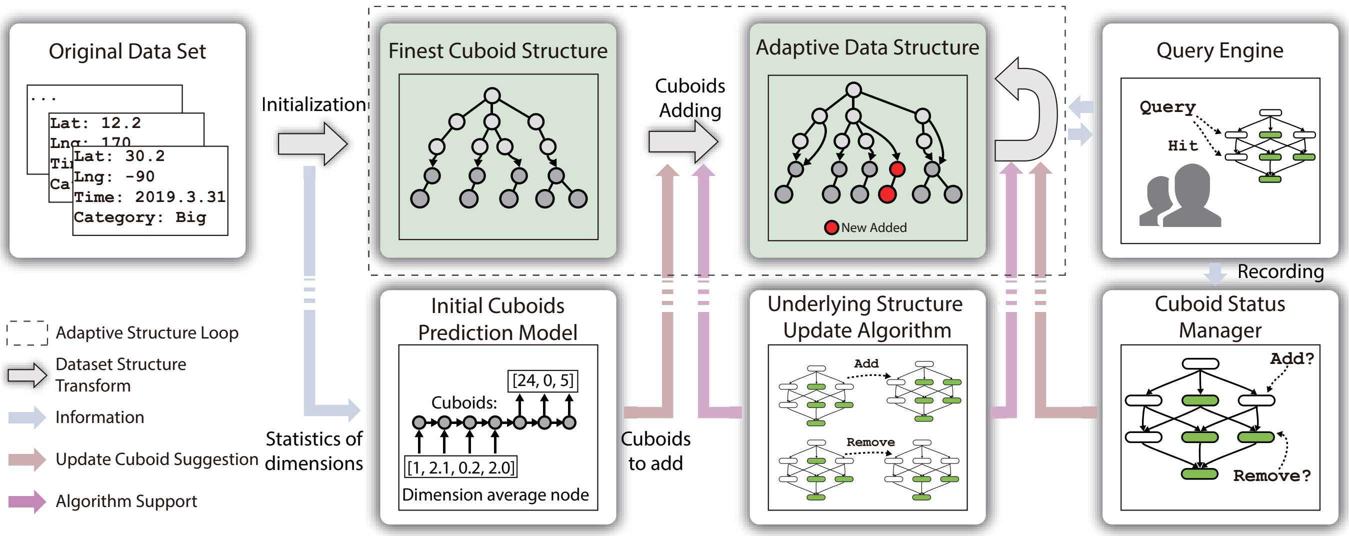

Fig. 1. The workflow of SmartCube. According to the original spatiotemporal data set, the algorithm builds the initial structure with the finest cuboid, which contains the finest aggregates. A machine learning based model can be added as an additional part to decide the cuboids that need to be created based on the statistic of the dimensions. The cuboids adding algorithm adds the cuboids to the data structure accordingly. SmartCube records user queries, and an updating decision making part decides whether a cuboid should be created or deleted according to the query records.

rithms part. For the data structure aspect, the original dataset is loaded to build the data structure with the finest cuboids. With a carefully-designed structure and the cuboid structure updating algorithm, the data structure can be updated dynamically. During the query process, the updating algorithm decides what cuboids to keep in the memory and when to create new cuboids. As an additional part, a prediction model can be added to decide which cuboids to be created at the beginning to address the cold-start problem.

3.2 Structure Definition

We first discuss the generic concept of data cubes. We then discuss SmartCube, which is an adaptive tree-like structure version of data cubes.

3.2.1 Data Cubes

Data cube [10] is a structure to handle data aggregate (e.g., count, sum, max, min) queries efficiently. The data cube of a dataset with dimension set $D=\{d_1,d_2,...,d_n\}$ has 2^n **cuboids** $C=\{c_1,c_2,...,c_n\}$, in which $c_i \in \{d_i, all\}, i=1,2,...,n$. We call C a k-dimensional cuboid when $k=|\{c|c\in C \wedge c \neq all\}|$. Each cuboid corresponds to an element of the power set of D. We use value *all* to represent the aggregate attributes. A cuboid with $c_i = all$ means that this cuboid aggregates on d_i dimension. We use $|c|$ to represent the cardinality of dimension c. Particularly, the cardinality of *all* is 1, i.e., $|all|=1$.

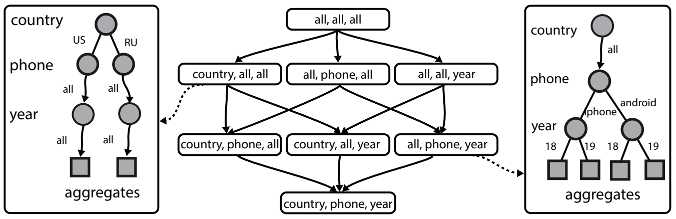

Fig. 2. The data cube of a dataset with level $D=\{d_1,d_2,d_3\}$. Each box here is a cuboid, and the *all* here means the aggregation level. Totally, there are $2^{|D|}$ (2^3) cuboids.

For example, as Figure 2 shows, given a dataset of the phone sales in some countries in several years, i.e., $D=\{country,\ phone,\ year\}$. There are 2^3 cuboids in the data structure. In the right box of Figure 2, the cuboid $\{all,\ phone,\ year\}$ is a 2-dimensional cuboid that aggregates over dimension *country* and stores value of $\{phone,\ year\}$ dimensions, e.g, iPhone sale in 2018.

In a data structure, a cuboid C can be calculated based on another cuboid C' when the dimensions in C is a subset of that in C'. For $C=\{c_1,c_2,...,c_n\}$ and $C'=\{c'_1,c'_2,...,c'_n\}$, We say C' is the **base cuboid** of C, $C \preceq C'$, if $|c_i| \le |c'_i|, i=1,2,...,n$. Among the base cuboids, the one that shares the smallest different dimensions (levels) is the **closest base cuboid (CBC)**. For example, the cuboid $\{all,\ phone,\ year\}$ is a base cuboid of $\{all,\ phone,\ all\}$. The cuboid with all the dimensions (levels) cannot be converted from any other cuboids, which is called the **finest cuboid**.

3.2.2 SmartCube Structure

The index dimensions in spatiotemporal data sets include spatial, temporal, and categorical dimensions. A query of the data set sets constraints on those dimensions to define the data items and returns the aggregate value of those items. To support the aggregates query on spatial, temporal, and categorical dimensions, a SmartCube data structure is built by organizing those dimensions on the original dataset.

SmartCube needs to index on the spatial, temporal, and categorical dimensions to support the queries. We build the index of spatial and categorical dimensions based on nanocubes [17]. We adopt the similar tree-like structure but each cuboid is independent. The categorical dimension is directly indexed by its value, while the range of temporal and spatial dimension cannot directly use the value because the value is tremendous and continuous. Thus, we need further division and discretization on those dimensions. Considering the different properties between spatial and temporal dimensions, we discuss them separately. To better describe the sub-dimensions of dimension (e.g., spatial or temporal dimension), we use the **level** to present the sub-dimension (e.g., *year* and *month* in the temporal dimension), and use the word **path** to refer to the ordered level series inside a dimension. Take the structure of Figure 3 as an example, the *year* and *month* are the levels of the temporal dimension and the path is $\{year,\ month\}$.

Categorical Dimension. The index of the categorical dimension is built directly by the value. There is no extra division in the categorical dimension. To use the uniform concept level and path, we define the categorical level l as the categorical dimension itself, and the path $p=\{l\}$.

Spatial Dimension. Quad-tree is often used in spatial data management [25]. It can be regarded as a cross binary division on latitude and longitude attributes. In SmartCube, quad-tree is used to build the hierarchical layers in the spatial dimension, which is highly efficient. Specifically, a 25-layer hierarchical quad-tree can make a precision loss of less than 1 meter, which is accurate enough to re-

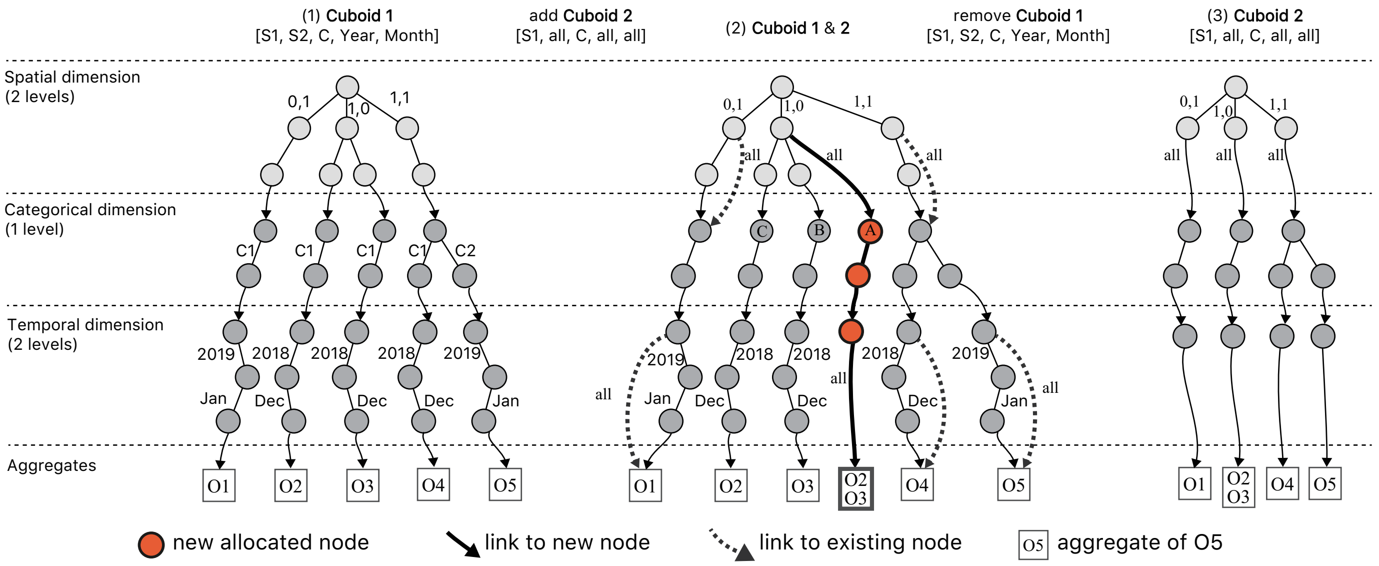

Fig. 3. The illustration of the updating algorithm. The structure on the leftmost only has Cuboid 1. This cuboid has all the possible levels, which are spatial levels S_1, S_2, and categorical level C, and temporal levels *Year*, *month*. The very right one is the tree with only cuboid 2 that aggregated on level S_2. In the middle are the virtual tree created during the process of creating Cuboid 2, and the tree with both Cuboid 1 and Cuboid 2 as well. Two cuboids share the nodes pointed by the dashed arrow line.

fer to any position in the world. The spatial levels can be presented as $\{s_i | i \leq depth_m\}$, in which s_i is the i-th layer on the quad-tree division and $depth_m$ is the max spatial division layers (we often set max depth as 25). The path of the spatial dimension can be presented as $\{s_1, s_2, ..., s_{depth_m}\}$. The granularity goes finer along the path of the spatial dimension. In the division of the spatial dimension, the deeper level depends on the upper level because the deeper level is the sub-division of the upper level. Thus there would not be the cuboid with combination skip in the levels of a dimension. e.g., Cuboid $C = \{s_1, all, s_3\}$ is not meaningful in real-world scenarios. The cuboids we can construct on the spatial dimension is the combination along the path order $\{s_1, s_2, s_3, ..., s_i, \underbrace{all, all, \ldots, all}_{depth_m - i}\}$, $i \leq depth_m$. For a more compact representation, the *all*s at the tail of spatial dimension can be omitted and the spatial dimension can be presented as: $C = \{s_1, s_2, ..., s_p\}$, where $p \leq depth_m$, and the *all* levels are omitted.

Temporal Dimension. Users would be interested in the aggregates time series on several levels or certain resolution. However, the levels of the temporal is complex. On one hand, there are some natural orders in the temporal dimension. We should consider the natural properties. For *month* and *year*, a *month* is defined under a *year* and a *year* always contains 12 *months*, we say the *month* is covered by *year*. On the other hand, some levels in the dimension are independent. For example, the *week* follows it's own order and is independent of *month*, and vice versa.

We define the partial order on the levels of temporal dimension, in which one level is totally covered by the next level. The real-world partial order chains built by connected partial order levels, e.g., $second \prec minute \prec hour \prec day_{month} \prec month \prec year$ and $second \prec minute \prec hour \prec day_{year} \prec year$. in which A_B is the level of n-th A inside a B, e.g., day_{month} represents the n-th *day* level under a *month*. Path (1) presents that there are several *month* inside a *year*, and several day_{month} inside a *month*, etc. Besides, if the path can fully recover the time stamps in certain resolution, we call this path a **partial order level list (POLL)**, e.g., the $hour \prec day_{month} \prec month \prec year$ is a POLL, while $second \prec minute \prec month \prec year$ is not. The division under the POLL can handle queries under certain resolutions and can get a time series as a result by drilling down and rolling up the aggregates over certain dimensions, which can support the times series query under the resolution. However, a POLL path is not enough to support the the query with levels across POLLs, e.g, query on the *month* and day_{week} are not supported by any POLLs. To support the query across POLLs, SmartCube extends the POLLs with extra levels as the base path. More than one path is allowed in SmartCube. The requirements are as follows.

- At least one path covers a POLL, supporting a complete query on time series.
- Unlike the spatial dimension which only has one path inside the dimension, there can be several ways to discretize the temporal dimension.
- The new path can be built according to another path. The new path is a sub-path of the existing path.
- The level order inside a path is not allowed to conflict the order in any POLL, e.g., the path day_{month} is always finer than *month*, while there is no constraint on day_{week} and day_{month}.

Formally, the path set in the time $P = \{p_i | i < M\}$, M is the path number, $p_i = \{t_1^i, t_2^i, ..., t_{depth_i}^i\}$. SmartCube seeks a trade-off between the partial order relation and the demands of the full user query. By default, we set the paths with two paths $P = \{\{year, month, day_{month}, day_{week}, hour\}, \{day_{week}, hour\}\}$.

Cuboid. With the definition, the cuboid can be formulated as: $C = \{\{s_1, s_2, ..., s_p\}, \{d_1\}, \{d_2\}, ...\{d_Q\}, \{t_1^i, t_2^i, ..., t_r^i\}\}$ Where $p \leq depth_m$, Q is the category dimension number, $r < depth_i$, $i < M$. We also use **schema** to present a cuboid, which is an array with the depth (i.e., the number of non-*all* levels) of each dimension. For example, the schema of $C = \{\{s_1, s_2, s_3, all, all\}, \{all\}, \{d_2\}, \{t_1^i, t_2^i\}\}$ is $S(C) = \{3, 0, 1, 2\}^i$, i is the path *id* in the temporal dimension.

The partial order relation in the cuboids of SmartCube is like this: For cuboid C and C', we say $C \preceq C'$ if the depth of each dimension of C equal to or smaller than corresponding depth in C'.

Specially, for a path p_i and cuboid C_f^i, if all the cuboids on the path satisfy $C \preceq C_f^i$, we call C_f^i the **finest cuboid**, which can be easily presented as: $S(C_f^i) = \{depth_m, 1, 1, ...1, depth_i\}^i$, which stores all the information on each levels in all dimensions. For example, in Figure 2, the cuboid $\{d_1, d_2, d_3\}$ is the finest cuboid. Since there may be more than one path in the temporal dimension and there is a finest cuboid for a path, it is possible that there is more than one finest cuboid.

3.2.3 Tree structure

The hierarchical layers follow the dimension order of spatial, categorical, and temporal. The order is similar to the nanocubes [17]. Inside each dimension, the levels follow in path order. We take an example to explain the tree structure of a cuboid. On the left side of Figure 3 is Cuboid 1. There are several levels in each dimension, e.g., $\{year, month\}$ in temporal dimension.

The nodes of cuboids share memory in the tree-like data structure, which is similar to nanocubes. However, each cuboid in the structure is allowed to add or remove. Thus, the sharing status of the node is needed to store in the nodes. The **accumulate shared number (ASN)**, which means how many cuboids the node is shared with, is introduced.

3.3 Structure Update Algorithm

The structure update algorithm operates on the nodes of the tree structure to create or delete a cuboid. We propose the algorithm of creating and deleting cuboids on the structure.

3.3.1 Creating Cuboids

After the original data set is loaded to the data structure, the new cuboids are, if needed, calculated base on the existing cuboids on the tree. We need to find the CBC of the new cuboid to calculate the cuboid. A new cuboid can always be added since at least the finest cuboid is the base cuboid.

Figure 4 describes the Pseudo-code of creating a new cuboid. We traversing the new tree from the root. For each child node, we count the corresponding nodes in the CBC. If there is only a corresponding node, we directly link it to the corresponding node according to the sharing mechanism. For there are more than one, we have to allocate a new node. Finally, in the aggregate level, we calculate the aggregate value of the new nodes based on the corresponding aggregates in the CBC. We show an example describe in Figure 3, we want to add Cuboid 2 on the Structure. Here, Cuboid 1 $\{S_1, S_2, C, year, month\}$ is the CBC for Cuboid 2. Starting from the root node, Cuboid 2 can share nodes with cuboid 1 until the S_2 level. The *all* node that directly links to categorical nodes in S_1 dimension is needed to calculate. For the $(0,1)$ node and the $(1,1)$ node in S_1, there is only one node in the categorical dimension. They can be directly linked to the categorical dimension. However, for the $(1,0)$ node, there are two nodes B and C in the categorical dimension. Thus a new node A is allocated. For the child node of A, there are two categorical values, which are C_1 and C_2. On the one hand, both B and C have the category value C_1, so the count is 2. We need to add a C_1 child node for A. On the other hand, the count for category value C_2 is 0, so there is no C_2 node for A.

3.3.2 Deleting Cuboids

The process of deleting a cuboid from the structure is showed below. First, find all the nodes of the cuboid when traversing the whole structure. Then, the ASN of the nodes in the cuboid minus 1. In the meanwhile, those nodes whose ASN value comes down to zero are freed. Figure 3 shows an example of the deleting process. Cuboid 1 is removed from the structure, and the nodes only kept by Cuboid 1 are freed.

3.4 Query Engine

SmartCube supports queries on spatial, categorical, and temporal dimensions. Different from other algorithms, the query route of the same query may change when the cuboids of the structure changes.

Take query $Q = \{country = America,\ language = all,\ device = iPhone\}$ for example, if the cuboid $\{country,\ all,\ device\}$ exists in the tree, the result can be obtained immediately. If $\{country, all, device\}$ is not in the tree, we need to drill down to $\{country,\ language,\ device\}$ to get the value of *America* and *iPhone* with different *language*s and aggregate them. The cuboid existing in the structure which can support the query is called the **supporting cuboid**s. Among them, the cuboid with needs least calculation for the query is called the **best supporting cuboid (BSC)**. The query process is divided into following parts, which are (1) finding the BSCs of the query, (2) drilling down to the BSCs to get the results, and (3) rolling up to get the aggregates.

Query on Spatial Dimension. The query on a certain spatial region is essential in the visualization of spatial-temporal data. Several common scenarios need the spatial region, e.g., density map on the geo-map of the current screen, the aggregate value of a selection of the region. The first dimension of the SmartCube is the spatial Quad-tree structure, in which a given region can be computed under the quad-tree intersecting algorithm [25]. The query seldom matches the region of the nodes in the quad-tree exactly, in the most situation we need to drill down to deeper layer to get the finer division.

We demonstrate a case to help to understand the data query on the spatial dimension. In Figure 5, we are interested in the rectangular areas on the map (a). We divide the region to fit the nodes of the tree: the $(1,0)$ node is totally inside the region, while the $(0,1)$ and $(1,1)$ node intersects with the region, which needs to go deeper. The child of $(0,1)$ do not fit the query, while the child of $(1,0)$, i.e., $((0,1),(0,0))$, is covered by the region. Now there are two levels of nodes need to calculate; they may refer to different CBCs respectively. In two situation (b) and (c), the cuboids in the tree are different. For (b), there is only a cuboid $\{s_1, s_2\}$ in the tree, the node $(1,0)$ still need to drill down to the second layer to get the aggregate value from cuboid $\{s_1, s_2\}$. For (c), there are two cuboid $\{s_1, all\}$ and $\{s_1, s_2\}$, the node $(1,0)$ follows the cuboid $\{s_1, all\}$ while the node $((0,1),(0,0))$ follows the cuboid $\{s_1, s_2\}$.

To be general, each layer of the spatial dimension need to find a CBC to support the query, which may lead to a situation where a query requires several different CBCs.

Query on Category Dimension. The categorical dimensions follow the flat trees, which have a root layer and the nodes of the root refer to different values in the category. The query engine simply follows the path with the value. It does not engage extra CBC as the Spatial query do.

Query on Temporal Dimension. We divide the temporal dimension into levels with different resolutions, which supports query on time series under different resolutions. SmartCube supports query on time series under a certain range with constraints on certain time levels and the given resolution. The query on the time series will drill down from the root of the time dimension until the given resolution, and the constraints of each layer and time range are considered for pruning the tree. If the current structure does not contain the cuboid of such layer, the drill-down and roll-up to the BSCs are also needed.

Considering the query together with the spatial region, there may be several paths on the spatial query which may have different BSC query. The final time series results are aggregated from the results of each BSC.

3.5 Updated Cuboids Decision Algorithm

During the process of the user query, SmartCube records the query to help decide which cuboid to create or delete.

Query Recording. We first define the cuboids that the query directly hit as **query cuboids**. A query can be presented as an array of cuboids and weight, which means the directly hit cuboid and their percentage. For a new query updated to the recent query record, there is a rate r to attenuate the frequency of the past query cuboid, i.e., the sum frequency in the past is set to r while the new query's frequency is set to $(1-r)$, then add the new query into the recent query records.

Update by Recent Query. We estimate the total latency of recorded query when creating a new cuboid and choose the cuboid that will reduce the latency to the greatest extent. For the cuboid C and its BSC C_{bsc}, the latency can be estimated by the size of these two cuboids, i.e., the larger the difference of the two cuboids' size, the more need to calculate for a results. The query latency mainly comes from the drill-down and roll-up operation, which is linear to $\frac{S_{C_{bsc}}}{S_C}$ on average. A cuboid's size S_C can be present by: $S_C = \Pi |c_i|_a, c_i \in C$, $|c|_a$ presents the average node of each level C. Thus, the query latency can be predict as $\frac{S_{C_{bsc}}}{S_C} = \Pi \frac{|c'_i|_a}{|c_i|_a}$. The estimated latency from a cuboid C to its BSC C_{bsc} comes from the cardinality of those levels that exists

```
function AddCuboid(root, new_schema, basic_schema)
    // root: root of the cube
    // new_schema: schema of the cuboid to be added
    // basic_schema: schema of the cuboid to be used
    nodes ← [root], candidates ← [[root]]
    for i ← 0 to Len(new_schema)-1 do
        [next_nodes, next_candi] ←
            AddCuboidChildren(nodes, candidates, 0, new_schema[i])
        Clear(nodes), Clear(candidates)
        for k ← 0 to Len(next_nodes) do
            node ← next_nodes[k]
            candi ← next_candi[k]
            children ← []
            for j ← 0 to Len(candi) do
                depth ← basic_schema[i] − new_schema[i]
                GetChildrenRec(candi[j], depth, children)
            end for
            if not HasContent(node) then
                if Len(children)==1 then
                    SetContent(node, GetContent(children[0]))
                else
                    SetContent(node, Node( ))
                end if
            end if
            IncEdgeRef(node, GetContent(node))
            Push(nodes, GetContent(node))
            Push(candidates, Map(GetContent, children))
        end for
    end for
    for i ← 0 to Len(nodes) do
        // these nodes store aggregates
        IncRef(nodes[i])
        UpdateContent(nodes[i], candidates[i])
    end for
end function
```

```
function AddCuboidChildren(nodes, candidates, depth, new_layer)
    if depth==new_layer then
        Foreach(IncRef, nodes)
        return [nodes, candidates]
    end if
    nnodes ← [], ncandi ← []
    for i ← 0 to Len(nodes) do
        node ← nodes[i]
        IncRef(node)
        children ← GetChildrenAll(candidates[i])
        nchildren ← Classify(children) by EdgeValue
        for (key, children) in nchildren do
            if not HasChild(node, key) then
                if Len(children)==1 then
                    SetChild(node, key, children[0])
                else
                    SetChild(node, key, Node( ))
                end if
            end if
            IncEdgeRef(node, GetChild(node, key))
            Push(nnodes, child)
            Push(ncandi, children)
        end for
    end for
    return AddCuboidChildren(nnodes, ncandi, depth + 1, new_layer)
end function
```

```
function GetChildrenRec(node, depth, children)
    if depth==0 then
        Push(children, node)
        return
    end if
    for child in GetChildren(node) do
        GetChildrenRec(child, depth − 1, children)
    end for
end function
```

Fig. 4. The pseudo-code of creating a new cuboid on the structure. In the left, the function AddCuboid builds the new cuboid based on the basic cuboid with the schema of these cuboids. In the right, the function AddCuboidChildren shares nodes or creates new nodes.

in C_{bsc} while not in C. Considering add a new cuboid, since the cuboid C is unknown, we use the average node number $|c'|$ of C_{bsc} to estimate the average node number $|c|$ of C. Thus, the latency from C to C_{bsc} is estimated as: $\Pi\{|c'_i|_a \mid c'_i \in C_{bsc},\ |c'_i|_a > 1,\ |c_i|_a = 1$.

With the latency estimated method defined, the update process is (1) to find a new cuboid C_{add} that has the maximum total latency reducing recent query list and (2) to find an existing cuboid C_{remove} that has the smallest total latency increase. SmartCube sets a threshold on the latency reduction when considering a cuboid to add.

When queries meet SmartCube, there are new valuable cuboids to be created or less important existing cuboids to be removed. With the valuable cuboids to be created, the latency of the SmartCube converges into a small value, which is called **convergence**.

3.6 Initial Cuboids Prediction Model

With the update algorithm and the update decision algorithm, the structure can work well for the query tasks. However, the query at the beginning may be time-consuming when the dataset is extremely large. We propose an additional part to show the possibility of a machine learning technique used to reduce the cold-start in the initialization. We introduce a machine learning model for predicting valuable cuboids and add them to the data structure. The input is the information of the datasets and the output is the cuboids that are predicted to add. The average node number of each level shows the basic properties of the data structure, e.g., the spatial and categorical distributions. In this simplest trial, the average node number array is taken to present the datasets. The output is a series of cuboids, we set as three cuboids, each of which is presented as the schema. The input is like this, (1, 2, 2, 1.3, 2.3), each number is the average node of each level; the output is like this, {3, 0, 2}, {2, 0, 1}, {1, 0, 2}, which is several cuboids' schema.

The input and output of the model are series. LSTM is a common method to deal with the series problem. We take the input and output as string and use a natural language translate model (OpenNMT) [14] to fit it. The training data set is generated randomly, we first generate a random average node series. A synthesized spatiotemporal dataset is generated to fit the series. We randomly generate the query with patterns (e.g., Q3) for the datasets. We apply the queries on it to find several valuable cuboids. The final cuboids are used as the output, and the dataset information is used as input. The model is then trained, which make the initialization be faster and suit for real datasets.

4 Implementation

We use C++ based implementation in the data structure. In order to support the structure, we need to store some information such as the path information, sub-tree information, and ASN information. Tagged pointers are used to store them for less memory usage. Libtcmalloc is used for optimizing memory usage as well.

We develop the approach as the back-end service, which supports HTTP API for website queries. The browser-based interface is developed in HTML, CSS, JavaScript, and D3.js library.

5 Performance Evaluation

We conducted several experiments to evaluate the efficiency of the SmartCube. These tasks in the experiments cover the common scenarios in visualization. To evaluate the adaptive ability of SmartCube, we also conducted a comparison with several state-of-the-art works on diverse tasks and data sets. From the results, we can conclude that SmartCube can be adaptive to different query tasks for different spatiotemporal data sets.

5.1 Evaluation Setup

The experiments were performed on two workstations. One of the experiment was performed on a Intel Core i7-8700 CPU clocked at 3.20GHz with 8 GB RAM. We used this workstation to evaluate our performance in small memory setting. The other one was performed

Table 1. The performance compared with the state-of-the-art algorithms under different datasets. The experiments with the '*' is the testing results quoted from the Time Lattice [21].

	Data	S_d(GB)	N(M)	Q1 M_c(GB)	Q1 t_c(ms)	Q2 M_c(GB)	Q2 t_c(ms)	Q3 M_c(GB)	Q3 t_c(ms)	Q4 M_c(GB)	Q4 t_c(ms)	Q5 M_c(GB)	Q5 t_c(ms)
SmartCube	BrightKite	0.37	4.7	0.61	0.12	0.89	1.66	1.53	3.89	1.53	13.82	-	-
NanoCube				2.64	0.25	2.64	3.32	2.64	1.46	2.64	21.99	-	-
SmartCube	Weibo	2.77	55.6	12.56	1.66	14.18	25.92	17.06	13.45	12.23	3.58	20.99	22.05
NanoCube				97.29	1.43	97.29	17.82	97.29	1.83	97.29	11.01	97.29	17.33

	Data	S_d(GB)	N(M)	Q6 M_c(MB)	Q6 t_c(ms)	Q7 M_c(MB)	Q7 t_c(ms)	Q8 M_c(MB)	Q8 t_c(ms)	Q9 M_c (MB)	Q9 t_c(ms)
SmartCube	Time Series	2.59	100	67	27.54	67	1.80	68	3.93	68	9.01
*Time Lattice				397	40.50	397	15.00	397	12.80	397	92.40
*NanoCube				41799	116.00	41799	4.60	41799	2491.80	41799	40083.90

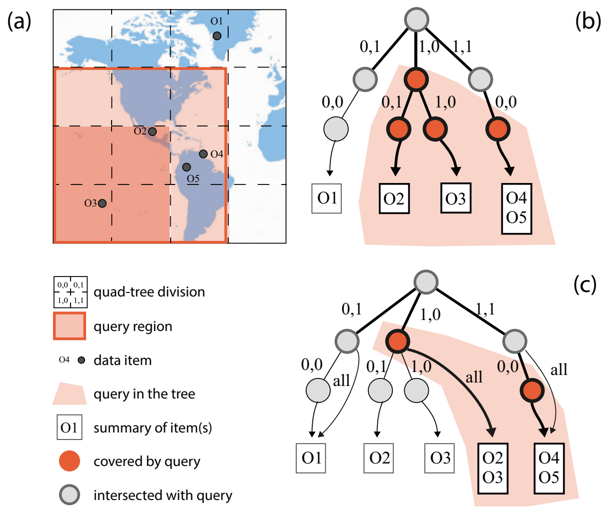

Fig. 5. The quad-tree division and the query on it in spatial region (a). Querying a region on the map will be translated to certain query on the tree. (b) and (c) are the same query on different tree status. The query routes are different. (c) can directly jump to the aggregate of $\{O2, O3\}$ while (b) needs extra drilling down on the tree to get aggregate of $\{O2\}$ and $\{O3\}$, and then combines the two values.

Table 2. The converge process in data Brightkite and Weibo. M_i the initial memory. T_i the initial time. T_c the convergence time. M_c the memory after convergence. t_i the query time before convergence. BK the BrightKite dataset. WB the Weibo dataset.

Data	M_i (GB)	T_i(s)	query	T_c(s)	M_c(GB)	t_i(ms)
BK	0.485	14	Q1	5.23	0.624	27.34
			Q2	3.42	0.911	108.12
			Q3	1.04	1.564	24.18
			Q4	1.12	1.565	64.01
WB	12.18	289	Q4	87	12.23	2768

on an Intel Xeon E5-2650 v4 CPU clocked at 2.20GHz, 128 GB RAM, on a Linux CentOS system to follow the setting of Time Lattice [21] since the open-source code is not accessible online.

We applied three datasets that range in size from 4 million records up to over 100 million records in our experiments to evaluate the adaptability of SmartCube. These data sets include both time series data and spatial-temporal data.

BrightKite. The BrightKite dataset contains 4.5M items with spatial and temporal attributes. It was collected by Cho et. al. [5], which contains the check-ins of a website between Apr. 2018 and Oct. 2010.

Weibo. Weibo is a microblogging social-media website mainly used in China. We collected the weibo post data from Jan. 2014 to Dec. 2014. The data set contains 55.6 Million posts with location of latitude, longitude and timestamps as well.

Time series. For time series evaluation, we focused on the range

Table 3. Queries in the Spatiotemporal Data Sets. Queries from Q1 to Q5 are related to spatial region query, some also related to time series query. Queries from Q6 to Q9 are the time series query.

Q1	query **the sum of values** given range and granularity
Q2	query day_{week} **and** $hour_{day}$ **distribution** given **spatial region** and **granularity**
Q3	query **time series** given **spatial region**, **temporal range** aggregated by **certain resolution**
Q4	query **time series** given **spatial region**, **temporal range** and **temporal constraints** aggregated by **certain resolution**
Q5	query **time series** given **spatial region**, **temporal range**, **temporal constraints** and **categorical constraints** aggregated by **certain resolution**
Q6	**select** time series **between** December 14, 1970 5:20 and February 3, 1972 9:20 **aggregated by** hour
Q7	**select** time series **group by** hour
Q8	**select** time series **where** time **between** 09:30 and 17:30 **group by** day_{week}
Q9	**select** time series **where** time **between** 09:30 and 17:30, month **in** [January, February, March] **group by** hour, minute

and time constraint query under dense time series. We follow the setting of Time Lattice [21] to generate the synthetic time series with 100M items. The data set is at *second* resolution with each second as a time step. Random number in uniform distribution is used as the value for each time step.

The queries used in the experiments are list in Table 3. Those queries cover the usual setting in the visualization scenarios.

Q1 to Q5 are the queries on a certain spatial region. Q1 is a query to get an aggregate value on a given region under a certain spatial granularity, which is useful to have a quick overview of a certain region. Q2 returns a 7×24 array on week day and hour. The visualization that needs to explore time patterns requires to query on such scenarios. Q3 queries a time series on a given time range, specific temporal resolution, and a given spatial region, which is a common query for the spatial region and time consideration. Q4 further increases the query complexity by adding the temporal constraints on the Q3, e.g, obtaining the time series of the work hour, work day in a certain region and time range. The query is useful when studying the labor-related problem. Q5 adds constraints on categorical dimensions based on Q3. One of the scenarios of Q5 is that we want to compare the spatial and temporal dimensions under different categories on the same spatial region and temporal range.

For the pure time series query, we focus on the aggregate query and the time series query. We adopted the temporal queries setting in Time-Lattice [21] and used it in our evaluation. Q6 queries for the time series under a range in certain resolution. Q7 is the query for the aggregation of certain time resolution. Q8 queries for the aggregation on the day of the week with the constraints in hour and minutes. Besides, the query for the time series with certain constraints under certain resolution is

Table 4. Comparison with some state-of-the-art algorithms.

	structure	query	cuboids	datasets
SmartCube	flexible	flexible	flexible	spatiotemporal
TimeLattice	fixed	direct, on the fly	some	temporal
nanocubes	fixed	direct	all	spatiotemporal
HashedCube	fixed	direct, on the fly	some	spatiotemporal

also common in visualization setting, as Q9 does.

5.2 Convergence of SmartCube

In summary, SmartCube can handle all the common queries with low latency. SmartCube was able to quickly converge on the overhead of adding cuboids during the query process to reach state-of-the-art latency. The memory after convergence is several times smaller than other state-of-the-art algorithms. Even before convergence, the query time is acceptable in the visualization exploration scenarios. In our evaluation, we conducted different types of queries on the BrightKite data sets and Q4 of Weibo data sets. As Table 2 shows, SmartCube fits into each query in a small convergence time within seconds. Even for large Weibo data with more than 50 million items, the convergence time is less than several dozens of seconds. Therefore we can ignore the convergence time in the real-world visualization exploration for users. (a) to (d) of Figure 7 shows the time performance as the query over time for each seconds. (e) and (f) in the figure shows the time performance and memory change as the temporal resolution of query changes. The two peaks are at the beginning of the resolution change.

In the beginning, the convergence time may face the cold-start problem. In order to reduce the latency at the beginning when the dataset is extremely large. We adopt a Machine learning model to show the possibility of predicting valuable cuboids when initializing the dataset. We train the model using synthesizing datasets and validate in the Weibo dataset (which is large) in order to show the generalization. The datasets number of the model is set to 300 because it has the best performance in the scenario. As Figure 8 shows, with the assistant of machine learning, the query latency during convergence time is several times lower than the cold start.

5.3 Comparison in Spatial-Temporal Dataset

The exploration on a visualization of spatial temporal dataset requires the understanding on the spatial relationship and time series together. Figure 6 shows a visualization system we built to explore spatial-temporal patterns with aggregate patterns. A visualization system with the density heatmap on a zoomable and panable map can help users to understand the spatial distribution. The related patterns on certain dimensions can help users to understand the distribution in a certain spatial region. For example, Figure 9 shows the week-hour pattern in certain regions under different spatial levels of the weibo data. This visualization can help the user to understand the day-hour pattern in a different region. The different task in the visualization system of spatial-temporal data can be covered by the queries in Table 3 from Q1 to Q5.

To evaluate the efficiency in those spatiotemporal case, we conduct the comparison from Q1 to Q5 with the state-of-the-art algorithms under the datasets of BrightKite and Weibo. Table 4 shows the properties of state-of-arts algorithms. When the structure store more cuboids, it can support more direct query. TimeLattice and HashedCube only stores some of the cuboids, the query on it require calculation on the fly. Among them, SmartCube has a flexible structure, which may need calculation on the fly at the beginning but soon directly get the results. Nanocubes [17] has the lowest query latency because it can get all the results directly. Hashedcube [23] focus on reducing the memory cost but result in lower performance. Most results on the hashedcube need calculation on the fly, which leads to a worse performance than nanocubes. SmartCube can create the cuboids to fit in all the similar queries next time, which may not need to calculate on the fly when getting convergence. We show that SmartCube can reach the corresponding performance with nanocube, which is higher than hashedcube.

Table 1 shows the time performance and memory size comparison with nanocubes under two datasets. The BrightKite data set is run in the machine with smaller RAM. In this setting, SmartCube have the schema: lat(25), lon(25), year(2), month(3), day(5), weekday(3), hour(5), while nanocube have the schema: $lat(25)$, $lon(25)$, $year(2)$, $month(3)$, $day(5)$, $weekday(3)$, $hour(5)$. For the larger data set on weibo, consider the nacube can not run on the smaller machine, we run on the machine with larger memory. SmartCube have the schema: $lat(25)$, $lon(25)$, $year(2)$, $month(3)$, $day(5)$, $weekday(3)$, $hour(5)$, $num_{image}(4)$, while nanocube have the schema: $lat(25)$, $lon(25)$, $time(16)$, $weekday(3)$, $hour(5)$, $num_{image}(4)$. These settings are to make sure nanocube can handle all the queries. The SmartCube can generate the results on the fly or directly get the results from the newly added cuboids. As SmartCube can add those cuboids quickly, most of the performance after convergence is similar to that of nanocube. For Q1 and Q2, the structure of Smartcube after convergence is similar to nanocube. In Q1 and Q2 of dataset BrightKite, the machine is with a CPU memory of 8G, the larger memory cost in nanocube may need more I/O exchange, that is why the SmartCube is quicker than nanocubes. In Q1 and Q2 of weibo dataset, the machine is with a large CPU and the I/O exchange is not needed, so nanocube is quicker than Smartcube. Consider the difference in the temporal dimensions, the performance in Q3 - Q5 may be quite different from nanocubes. As the constraints on the temporal level, the performance of SmartCube becomes better (From Q4 and Q5). That's because the hierarchical structure of the temporal dimension help to jump those nodes who's don't fit the constraints in a high level while nanocubes need to go through an empty array to put into each value.

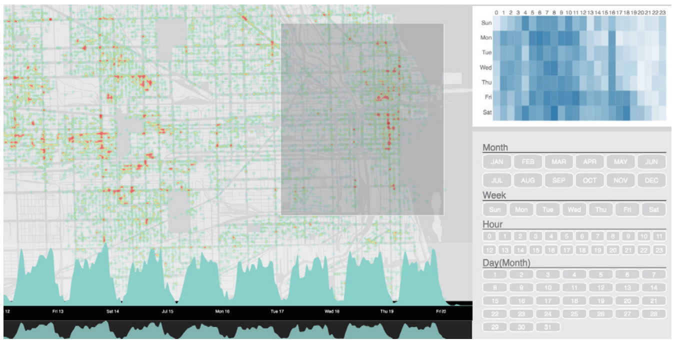

Fig. 6. An example of visualization that requires the aggregations on several combination of the data sets. The constraints on time can be added by the right part. The query for spatial region is for time series, temporal pattern in hour of week days.

5.4 Comparison in Time series dataset

SmartCube supports the query on certain constraints on the levels and return the time series with a certain resolution. The common queries on temporal dimension is supported by the the SmartCube structure. To evaluate the efficiency in temporal dimension. We conduct the comparison the comparison with the Time Lattice, the state-of-art data management method [21]. Time Lattice conduct several comparison on time series data with existing techniques, which outperforms other techniques in memory cost and time costing. However, the code of the Time Lattice is not accessible. To compare with Time Lattice, we follow the hardware and software setting in Time Lattice. In hardware setting, our CPU is Intel Xeon E5-2650 CPU clocked at 2.20GHz and CPU of Time Lattice is Intel Xeon E5-2650 CPU clocked at 2.00 GHz. There is only small difference in clock frequency. The memory costs in different environment should be the same. Results show that SmartCube costs smaller memory then TimeLattice and can support the same query. Both SmartCube and TimeLattice can be regarded as hierarchical trees in the time series. In Q6, SmartCube have similar performance with TimeLattice. While in Q7 - Q8, SmartCube can be faster than TimeLattice mainly because the multi-path in temporal dimension. Our performance in the 100M random datasets shows the high performance than other counterparts.

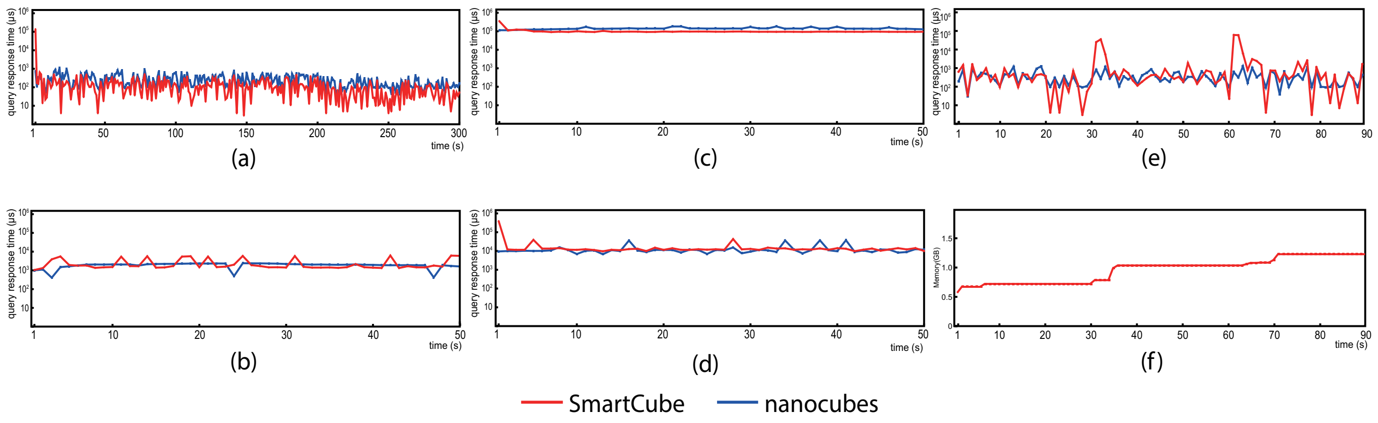

Fig. 7. (a) to (d) is the performance of Query Q1 to Q4 change over time in the BrightKite dataset; (a) for Q1, (b) for Q2, (c) for Q3, and (d) for Q4, respectively. (e) and (f) shows the query performance and memory change as the query pattern of different resolution. The query time convergence soon as the new cuboid are added to the data structure.

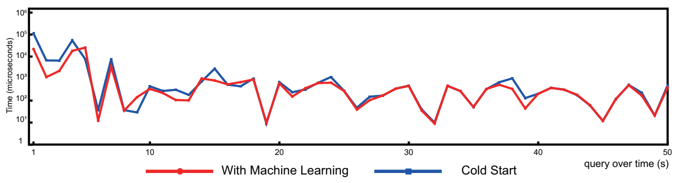

Fig. 8. The performance of ML-initialize towards the cold start. It shows that ML reduces the query time before finishing convergence.

6 Discussion

SmartCube is an adaptive structure for the spatiotemporal datasets. It realized the adaptive ability by operating the cuboid. A cuboid is defined by the dimension combination, which contains aggregations in all the k-dimensional array. Our process only updates the cuboid information when a cuboid is finished adding to the cuboid, i.e., during the adding cuboid process, the query for such cuboid cannot reach a partially finished cuboid and can only query existing cuboids on the structure to get the results. In the current setting, for the extremely large dataset, the query hit a non-existed cuboid would have a high latency until the building of the cuboid is finished. However, during the adding process, some query on this cuboid can be handled by the partially finished cuboid. The query on partially finished cuboids has two possible solutions. One is to record the more detailed information than cuboid; another is to the estimation of the probability to successfully query on these cuboids, together with the fault-tolerance algorithm to support the query.

Under the multi-user environment, the common situation is that many users use the same visualization systems querying on the same structure. The convergence memory cost will still be smaller than the state-of-art algorithms like nanocubes. The reason is that there are limits of the possible queries in a visualization system. Even if all the cuboids that directly fit the queries of a system is load, the cuboids are still a subset of all the cuboids, which guarantee that the memory of SmartCube will be relatively smaller than nanocubes.

To suit for the extremely large datasets, the distributed system may be induced to the data management. The dynamic management of the SmartCube can be combined with a distributed system for a high-performance cuboid update and distributed query.

To better suit for the memory limits, setting a hard constraint on the memory is useful and can easily be extended from current framework, e.g., on a server of visualization of large datasets, set a memory limit to make sure the data structure stay in the memory.

In the future, machine learning techniques will be introduced in more parts of the pipeline for a smarter data structure.

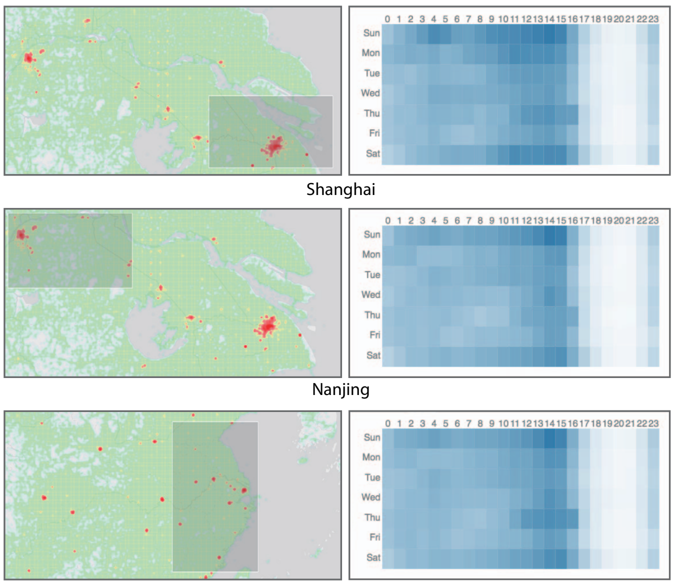

Fig. 9. The heat map shows the weekday × hour values according to the certain region on the map. The darker the square color is, the higher the value it represented is.

7 Conclusions

In this work, we proposed SmartCube, a flexible framework for spatiotemporal data sets. Not only organizing the data structure at the beginning, but the framework also changes adaptively according to query patterns . The framework takes the cuboids as the basic unit and updates the data structure by creating and deleting cuboids. When new query are received, the updating engine will judge the query changes, estimate the performance and memory changes, and finally, make the decision of cuboids changes.

We conducted several comparisons with state-of-the-art algorithms. The results indicate that our algorithm is adaptive to different forms of spatiotemporal datasets. The performance increases as the users interact with the system. Our algorithm reaches a comparable performance of the state-of-the-art algorithm with smaller memory cost.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments. This work is supported by NSFC No. 61672055 and 61872013, and the National Program on Key Basic Research Project (973 Program) No. 2015CB352503. This work is also supported by PKU-Qihoo Joint Data Visual Analytics Research Center.

References

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pp. 29–42. ACM, New York, NY, USA, 2013. doi: 10.1145/2465351.2465355

[2] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of the International Conference on Management of Data*, SIGMOD '16, pp. 1363–1375. ACM, New York, NY, USA, 2016. doi: 10.1145/2882903.2882919

[3] L. Battle, M. Stonebraker, and R. Chang. Dynamic reduction of query result sets for interactive visualizaton. In *Proceedings of the IEEE International Conference on Big Data*, pp. 1–8, Oct 2013. doi: 10.1109/BigData.2013.6691708

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975. doi: 10.1145/361002.361007

[5] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pp. 1082–1090. ACM, New York, NY, USA, 2011. doi: 10.1145/2020408.2020579

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] K. Deng, K. Xie, K. Zheng, and X. Zhou. *Trajectory Indexing and Retrieval*, pp. 35–60. Springer, 09 2011. doi: 10.1007/978-1-4614-1629-6_2

[8] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire. A GPU-based index to support interactive spatio-temporal queries over historical data. In *Proceedings of the IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 1086–1097, May 2016. doi: 10.1109/ICDE.2016.7498315

[9] D. Fisher, I. Popov, S. Drucker, and m. schraefel. Trust me, i'm partially right: Incremental visualization lets analysts explore large datasets faster. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pp. 1673–1682. ACM, New York, NY, USA, 2012. doi: 10.1145/2207676.2208294

[10] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, Mar 1997. doi: 10.1023/A:1009726021843

[11] H. Guo, J. Zhang, R. Liu, L. Liu, X. Yuan, J. Huang, X. Meng, and J. Pan. Advection-based sparse data management for visualizing unsteady flow. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2555–2564, 2014.

[12] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997. doi: 10.1145/253262.253291

[13] J. Im, F. G. Villegas, and M. J. McGuffin. Visreduce: Fast and responsive incremental information visualization of large datasets. In *Proceedings of the IEEE International Conference on Big Data*, pp. 25–32, Oct 2013. doi: 10.1109/BigData.2013.6691710

[14] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. OpenNMT: Open-source toolkit for neural machine translation. *ArXiv e-prints*.

[15] J. Li, S. Chen, W. Chen, G. Andrienko, and N. Andrienko. Semantics-space-time cube. a conceptual framework for systematic analysis of texts in space and time. *IEEE Transactions on Visualization and Computer Graphics*, 2018.

[16] M. Li, F. Choudhury, Z. Bao, H. Samet, and T. Sellis. Concavecubes: Supporting cluster-based geographical visualization in large data scale. *Computer Graphics Forum*, 37(3):217–228, 2018. doi: 10.1111/cgf.13414

[17] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, 2013.

[18] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, Dec 2014. doi: 10.1109/TVCG.2014.2346452

[19] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. *Computer Graphics Forum*, 32(3pt4):421–430, 2013. doi: 10.1111/cgf.12129

[20] B. McDonnel and N. Elmqvist. Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1105–1112, Nov 2009. doi: 10.1109/TVCG.2009.191

[21] F. Miranda, M. Lage, H. Doraiswamy, C. Mydlarz, J. Salamon, Y. Lockerman, J. Freire, and C. T. Silva. Time lattice: A data structure for the interactive visual analysis of large time series. *Computer Graphics Forum*, 37(3):23–35, 2018. doi: 10.1111/cgf.13398

[22] F. Miranda, L. Lins, J. T. Klosowski, and C. T. Silva. Topkube: A rank-aware data cube for real-time exploration of spatiotemporal data. *IEEE Transactions on Visualization and Computer Graphics*, 24(3):1394–1407, March 2018. doi: 10.1109/TVCG.2017.2671341

[23] C. A. L. Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):671–680, Jan 2017. doi: 10.1109/TVCG.2016.2598624

[24] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pp. 10–18. ACM, New York, NY, USA, 1981. doi: 10.1145/582318.582321

[25] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[26] Y. Sismanis, A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical dwarfs for the rollup cube. In *Proceedings of the 6th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '03, pp. 17–24. ACM, New York, NY, USA, 2003. doi: 10.1145/956060.956064

[27] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the petacube. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pp. 464–475. ACM, New York, NY, USA, 2002. doi: 10.1145/564691.564745

[28] C. D. Stolper, A. Perer, and D. Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1653–1662, Dec 2014. doi: 10.1109/TVCG.2014.2346574

[29] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, Jan. 2002. doi: 10.1109/2945.981851

[30] C. Stolte, D. Tang, and P. Hanrahan. Multiscale visualization using data cubes. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):176–187, April 2003. doi: 10.1109/TVCG.2003.1196005

[31] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pp. 441–448, June 1996. doi: 10.1109/MMCS.1996.535011

[32] Z. Wang, D. Cashman, M. Li, J. Li, M. Berger, J. A. Levine, R. Chang, and C. Scheidegger. Nncubes: Learned structures for visual data exploration. *arXiv preprint arXiv:1808.08983*, 2018.

[33] Z. Wang, N. Ferreira, Y. Wei, A. S. Bhaskar, and C. Scheidegger. Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):681–690, Jan 2017. doi: 10.1109/TVCG.2016.2598694

[34] Wei Wang, Jianlin Feng, Hongjun Lu, and J. X. Yu. Condensed cube: an effective approach to reducing data cube size. In *Proceedings 18th International Conference on Data Engineering*, pp. 155–165, Feb 2002. doi: 10.1109/ICDE.2002.994705

[35] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka. Dynamic load balancing based on constrained k-d tree decomposition for parallel particle tracing. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):954–963, 2018.

[36] J. Zhang, H. Guo, and X. Yuan. Efficient unsteady flow visualization with high-order access dependencies. In *Proceedings of the IEEE Pacific Visualization Symposium (PacificVis)*, pp. 80–87. IEEE, 2016.