A Hybrid Prediction and Search Approach for Flexible and Efficient Exploration of Big Data

Abstract—This paper presents a Hybrid Prediction and Search approach (HPS) for building visualization systems of big data. The basic idea is training a regression model to predict a coarse range on the dataset and then searching target records that satisfy the query conditions within the range. The prediction reduces the storage cost without preprocessing a data structure storing aggregate values of queriable attribute range combinations. Meanwhile, the search eliminates the prediction bias inevitable for machine learning models. Experiments on multiple open datasets demonstrate HPS's comparable query speed to existing techniques and the potential of continuous performance improvement by investing more hardware resources. In addition, the feature of returning original records instead of aggregate values brings better use flexibility, enabling to construct visualization systems with display/query functions that are unavailable for existing techniques.

Index Terms—interactive data exploration, visual query, machine learning, neural network, visual analytics.

1 Introduction

Interactive data exploration (IDE) is a traditional research topic in visualization. In a typical scenario, the user selects value ranges on attributes to generate views showing aggregate patterns of filtered records. When the dataset is large, the query latency becomes long due to time-consuming record traversals.

There are many techniques for accelerating the workflow by preprocessing a data structure to store aggregate values of queriable attribute range combinations [19, 22, 23, 27, 29]. Correct values can be fetched from the data structure immediately, thus achieving efficient IDE by avoiding any record traversal. However, the pre-storing strategy suffers from the problem of high storage overhead. On the other hand, the success of deep learning has inspired researchers to apply neural network-based models to improve IDE efficiency. A recent work trains a neural network that encodes the user's query as the input and predicts the corresponding aggregate values [35]. The predictive manner avoids preprocessing the data structure. However, the inevitable prediction bias of any machine learning models affects their applicability in scenarios with strict accuracy requirements.

In this paper, we propose a **H**ybrid **P**rediction and **S**earch approach (HPS) to the interactive exploration of big data. HPS follows a two-step workflow that (1) training a regression model to predict a coarse search range for each user-specified query condition and (2) searching target records within the range. We then aggregate these records using GPU-accelerated algorithms to generate visualizations. HPS has the advantages of both two types of techniques. The predictive manner avoids constructing the huge data structure storing aggregate values, thus reducing the storage overhead. The search eliminates prediction bias inevitable for any machine learning models.

We test the performance of HPS on multiple open datasets. The feature of returning target records brings better use flexibility. We develop three systems with display/query functions that are unavailable for existing techniques to demonstrate this point, which supports (1) constructing views showing either records or aggregate values, (2) changing the aggregation function during the exploration, and (3) querying on many (e.g., >2) finely-binned attributes. Quantitative results show that HPS achieves comparable query speeds to existing techniques on datasets with million-scale records using a single GPU, while its storage cost is much lower than those of existing techniques. Moreover, we can further improve HPS's query speed on larger datasets by adding more hardware resources.

In summary, our main contribution is a hybrid prediction and search approach for building interactive visualization systems of big data, which enables (1) better use flexibility, and (2) much lower storage cost than existing techniques, and (3) continuous improvement in query speed by adding more hardware resources.

2 Related Work

This section reviews three categories of techniques that support interactive exploration of large datasets.

2.1 Structure-Based Approach

Data Cube is a classic OLAP (Online Analytical Processing) [2] technique for processing big data, which allows retrieving aggregate values on any attribute combination in real-time. Its main problem is the unsuitability for high-dimensional data, as the combinatorial explosion of attributes results in unacceptable storage overhead.

How to reduce the storage cost becomes the key to the subsequent techniques. Liu et al. [22] limited the maximum number of indexed attributes of a spatiotemporal dataset to four, based on an observation that spatiotemporal exploration involving more than four attributes is uncommon. Miranda et al. [24] made use of an implicit temporal hierarchy to construct a memory-efficient index structure. Zhao et al. [40] proposed a KD-Tree-based index to support interactive exploration of large time series. Pahins et al. [28] proposed QDS that supports richer aggregate measures, such as quantiles and cumulative distribution. Ghosh and Eldawy [9] proposed AID* that indexes tiles describing the data distribution rather than original data. Lins et al. [19] proposed a quadtree-based structure, entitled Nanocubes, to index large spatiotemporal datasets. Nanocubes' branches corresponding to the regions that do not contain any data object are removed to reduce the memory usage. Nanocubes achieve optimized storage costs on sparse datasets and many essential features, such as high-resolution 2D query/display, bias-free outputs, etc. Nanocubes' success makes further reducing index sizes through data structure optimizations without degrading other performance aspects difficult.

As a trade-off between storage overhead and query speed, Liu et al. [20] utilized a "pre-fetching "strategy to reduce the size of Nanocubes by storing values most likely to be queried in the future. Pahins et al. [29] propose Hashedcubes that stores beginnings and ends of a list of ordered records instead of aggregate values of attribute combinations. It may, however, not ensure the display accuracy by using a leaf-sized parameter. Mei et al. [23] design a data representation for querying tabular datasets, that returns only approximate results. Gaining the performance or functionality in one aspect, the above techniques may loose performance in others. Maximally reducing the index size while enabling high-resolution query/display on many finely-binned attributes is still a challenge, one of our approach's primary goals.

There are also techniques designed for specific requirements, such as Semantics-space-time Cube [16] for exploring large spatiotemporal texts, Gaussian Cubes [36] for generating visualization involving complex algorithms, Concave Cubes [18] for drawing clustering results, TopKube [25] for showing top-ranked objects, and Trip Cube [38] for

indexing large trajectories. These techniques, however, are designed for the professional data analysis field, which is different from ours.

2.2 Sampling-based Approach

Sampling means the selection of a subset of elements to estimate characteristics of the entire dataset. By reducing the amount of data in the calculation, a database engine can return results with a low time latency [1,13]. Kwon et al. [15] systematically summarized the value of sampling for visual analytics. Sampling-based techniques generate only approximate results. As singular data points may be excluded from the calculation, sampling-based techniques cannot be used in some applications, such as anomaly detection. Moreover, because the calculations are performed on selected samples, the results are uncertain, and an estimate of the uncertainty is needed [3,8].

Progressive exploration is an extension of the sampling approach. This approach calculates results step by step on an incrementally enlarged sample set. Users can thus gradually obtain more accurate results [10,12,34]. In recent years, progressive exploration has gained attention in human-computer interaction [8] and visualization [39]. Many visualization systems utilize progressive strategies to improve system response speed when handling large datasets [14,17]. As a representative progressive IDE technique, Falcon [27] constructs index structures only for the active view (the user is manipulating) to reduce the index size. Research has demonstrated that progressively refined approximate results are often sufficient for exploratory analysis [7,8,26]. Most current works focus on constructing an initial subset of data [30], and how to develop a strategy for tailoring candidate queries [34]. Recent studies have analyzed the effectiveness [39] and the main challenges [33] of progressive data exploration.

HPS is not a sampling-based technique and does not have the above issues. It enables low storage costs and high-resolution 2D query/display simultaneously. Furthermore, it also enables to show original records by storing records in stored arrays.

2.3 Prediction-Based Approach

There is an increasing trend of applying deep learning in visualization. Utilizing learning-based techniques in data exploration receives much attention. Rossi et al. [31] combine visual representations with graph mining and learning techniques to aid in revealing important data insights. Chen et al. [5] design a deep neural network for lasso selection of 3D point clouds. He et al. [11] train a deep learning model as a surrogate to enable in-site exploration of the large parameter spaces of ensemble simulations. Chen et al. [4] utilize the generative model to extract density maps' dynamics by producing interpolation information. Xie et al. [37] present a hypergraph learning algorithm for visual analytics of heterogeneous datasets. However, these works are different from our approach that aims to form a common framework for exploring various large datasets.

There have been learning-based techniques for interactive data exploration. For example, NeuralCubes [35] trains neural networks to approximate datacube. The significant difference between HPS and NeuralCubes [35] is that HPS trains models on individual attributes, while NeuralCubes sets a single model to handle all multi-dimensional queries. We propose the structure for two reasons. First, fitting a multi-dimensional distribution is more complicated than a linear one. HPS thus enables a higher prediction accuracy, fewer parameters, and easier model tuning. Second, it avoids massive training sets (HPS's models take single keys as inputs, Section 4.2). NeuralCubes [35] has to include training samples of attribute range combinations. The number exponentially increases with that of attributes and the binning granularities. NeuralCubes [35] thus supports low-resolution queries and returns approximate results only.

Prediction-based techniques have many inherent issues despite small sizes, such as inevitable prediction bias, huge training set, slow training, and lengthy model tuning. HPS avoids them through a series of design optimizations.

3 Problem Statement

We first define the IDE query and introduce the query decomposition, based on which a first binary search-based IDE solution is given. By carefully analyzing the pros and cons of the solution, we identify HPS's design goal.

3.1 IDE Query Definition

HPS is used to construct IDE systems with multiple coordinated views. Users can select arbitrary value ranges on multiple attributes as a query through common interactions, such as panning, zooming, brushing, etc. After the user's selection on any view, all other views are updated correspondingly.

Let $D = (a_1,\ a_2,\ ...,\ a_n)$ be a dataset with n attributes. Query can be defined as $q = (r_1(a_1),\ r_2(a_2),\ ...,\ r_n(a_n))$, where $r_i()$ is an operation that returns a continuous value range on the attribute a_i. In all existing techniques, value ranges of attributes are discretized into bins [19, 22, 23, 29], as in Fig. 1. A bin is the smallest unit for value range selection, as in Fig. 1. Each attribute range $r_i(a_i)$ thus is a continuous bin interval.

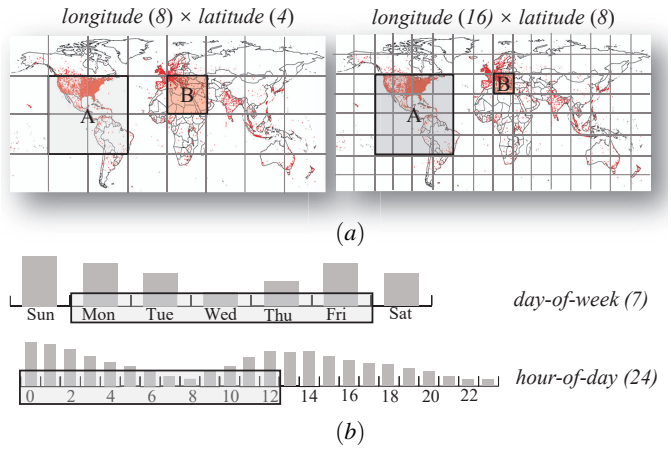


Fig. 1. Illustration of IDE query. Rectangles with black borders represent the selected attribute ranges, which should cover integral bins. (a) Two binning granularities on longitude and latitude to support different query/display resolutions. (b) Temporal attributes are discretized according to two relevant time cycles.

We can divide value ranges of attributes into an arbitrary number of bins to support either high or low query/display resolution, see the two maps in Fig. 1(a). There is no standard for how many bins an attribute range has to be called finely-binned. For example, we consider longitudes and latitudes in Nanocubes [19] are finely-binned, whose value ranges are divided into 2^{25} bins, thus enabling the identification of individual objects in streets. There are also naturally-discretized attributes, such as day-of-week and hour-of-day having 7 and 24 bins respectively, as in Fig. 1(b). Their binning granularities are relatively low.

3.2 Query Decomposition

We can divide a query into many 1D queries, i.e. $q = (q_1, q_2, ..., q_n)$, where $q_i = (r_i(a_i))$ is a 1D query with a single specified attribute range. Assume RS is the record set satisfying q, RS_i is the record set found by executing q_i. It is obvious that $RS = \cap_{i=1}^{n} RS_i$. We thus can get the same records of q by separately executing all 1D queries and calculating the intersection of their results. Fig. 2 shows an example of decomposing a spatial query into two 1D queries on longitude and latitude respectively.

3.3 A First Search-based Solution

We propose a method to find records for each 1D query quickly. As in Fig. 3, we can establish a sorted array for the attribute, which stores all records that are sorted according to their values on the attribute. For each query, we execute the binary search twice to find two positions for

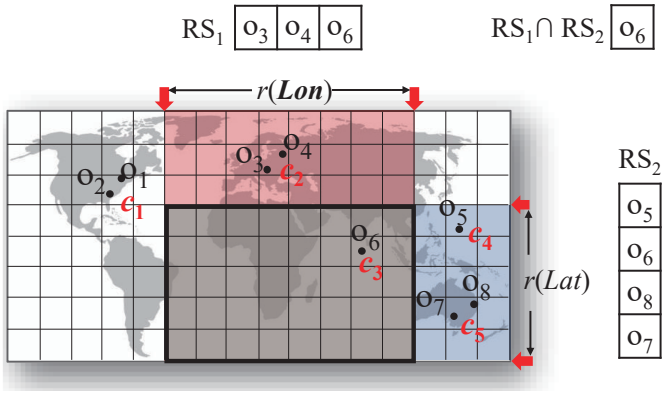


Fig. 2. Example of the query decomposition. The user brushes a rectangle on a map (marked with a black box) to select objects in it. We decompose the query into two 1D queries specifying value ranges on longitude (marked red) and latitude (marked blue). Two record sets RS_1 and RS_2 can be found by executing the two 1D queries, while their intersection contains target records within the brushed region.

the endpoints of the queried attribute range. Records between the two positions on the sorted array are targets. Because records are sequential in the sorted array, the binary search can be efficient.

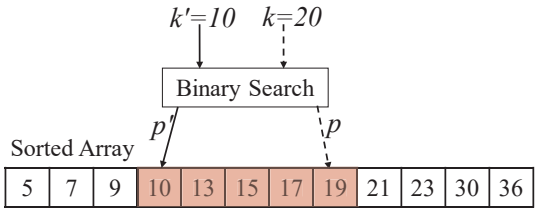


Fig. 3. Finding records in (10, 20) of an attribute using the binary search. A sorted array is necessary, in which all records are sorted according to their values on the attribute.

Along the line, we can decompose each query into multiple 1D queries and execute all decomposed 1D queries separately. Thus, target records satisfying the IDE query can be quickly obtained by executing all decomposed 1D queries and calculating the intersection of records found in all 1D queries. Fig. 4 shows the case of applying the method on a dataset with five attributes.

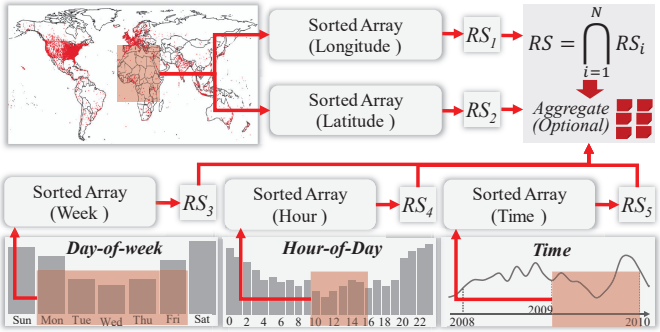


Fig. 4. IDE of a dataset with five attributes. We create a sorted array for each attribute to find records within the value range of the attribute. Target records can be obtained by calculating the intersection of record sets fetched from different sorted arrays.

The advantages of the solution are obvious. First, it can return records satisfying the IDE query. Second, the storage cost is low. We construct a sorted array consisting of all records for each attribute. The overall storage cost is linearly proportional to the number of attributes, unlike most existing techniques, whose sizes will exponentially increase as the number and binning-granularity of attributes to be jointly queried. Third, its outputs are always correct without any bias. In other words, the solution is not a progressive technique that has to sacrifice query precision to achieve a smaller storage cost. Fourth, it adapts to both sparse and dense datasets, unlike most existing techniques designed for sparse datasets exclusively [19, 23, 29]. Sorted arrays store original records, unrelated to the distribution of records, and we can always find correct records through the binary search. Finally, it applies to frequent data changes. For inserting or deleting an attribute, we simply add or remove the corresponding sorted arrays. For receiving or removing records, we just need to insert (or delete) affected records in sorted arrays, which can be accomplished efficiently, since all records in sorted arrays are sequential.

An overlooked issue is the time overhead of finding records of 1D queries is not fixed and will increase with the number of records. Moreover, the above solution involves on-the-fly calculations, i.e. intersection and aggregation, which are time-consuming when the dataset contains many records. We thus propose HPS to resolve the problems. We will introduce HPS's principle and design optimizations in the next Section.

4 Approach

We introduce the hybrid approach after a simple description of the principle of predicting the search range. We then give a series of design optimizations to improve the usability of the approach.

4.1 Prediction Principle

HPS is used to fetch records within the value ranges of 1D queries. Each HPS is coupled with a sorted array to find records within value ranges of an attribute. The sorted array contains all the data records. These records are sorted according to their values on the attribute. The length of the sorted array is equal to the number of records in the dataset. An important observation is that we can draw a curve to reflect the variation of positions along with their attribute values, as in Fig. 5(a). The variation is monotonically increasing. Thus, we can set a regression model to fit the curve. For each lookup attribute value k (below we call it a **key** for simplicity), the model can predict the corresponding **position** p on the sorted array, as in Fig. 5(b). The attribute value of p is the first one equal to or higher than k. The model should run twice to map the two endpoints of each queried range to two positions. Records between the two positions are what we want to find on the attribute.

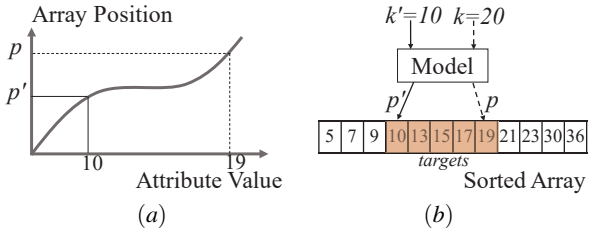


Fig. 5. Principle of predicting records within an attribute range, depending on two components, i.e. a sorted array and a regression model. (a) The sorted array stores all records. The relationship between positions and their keys on a sorted array is illustrated as a curve. The model is fitting the curve. (b) Finding records within $(10, 20)$. The regression model runs twice to map two endpoints to the two positions on the sorted array. Records between the two positions are the targets.

Training a regression model to fit a monotonically increasing curve is easy. We traverse all positions in a sorted array to collect training samples. Each training sample is in the form of $\langle k, p \rangle$, where k is the queried key and the p is the corresponding position. We include all key-position pairs into the training set. The training set size thus is equal to the number of bins divided on the attribute. For example, the longitude with 2^{25} bins results in about 33M training samples for the regression model, which most ordinary GPU servers can handle.

For an attribute with fewer bins, such as day-of-week (7) and hour-of-day (24) (see Fig. 4), we design a hash-map storing all key-position

pairs to support the query instead of training a model. For example, hash-maps of day-of-week and hour-of-day contain 7 and 24 key-position pairs, respectively. The hash-map can return the true position immediately without any bias for a key. This design aims at improving efficiency by skipping unnecessary training processes. Moreover, it can reduce the storage cost, since a hash-map with a few pairs is much smaller than a model.

4.2 Hybrid Prediction and Search

Regression models are good at learning a high-level trend. The prediction result can be very close to the true value, but it is impossible to eliminate the small biases, as in Fig. 6(a), affecting the correctness of the generated views. We utilize a hybrid prediction and search method to find the true position of a key. First, we use the model to predict a position p. Second, we set an initial search range threshold *thres*, and use binary search to find the true position (the first position with the attribute value equal or larger than the queried key) within $[p-thres, p+thres]$. **If the true position is not in the interval, we double *thres* and search again. This process repeats until finding the true position.** The binary search can be very efficient. Fig. 6(b) shows a case of finding the position of key 24.

In addition to avoiding the storage of aggregate values, the prediction shortens the search range, making the search time a constant (Section 6.5). The method also reduces the training time, because we can stop training whenever the current bias is less than the initial search range (we actually implement a regression model as a number of small models, which will be introduced in the next section, and the method is also suitable for each small model to reduce the training time). The effects of the initial search range *thres* on training time have been tested and summarized in Table 2.

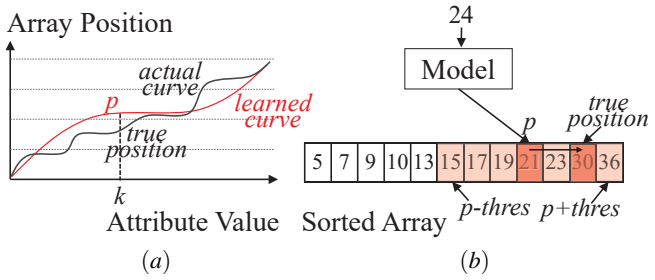


Fig. 6. Hybrid prediction and search. (a) The prediction bias is inevitable even for a model well capturing the general trend of the curve. (b) An example of predicting the position of 24. Having obtained a predicted position P, HPS searches for the true position within $[p-3, p+3]$, where 3 is the preset initial search range threshold.

4.3 Design Optimizations

We propose the following optimizations to resolve prediction-related issues, which better adapt the HPS to real-world scenarios.

4.3.1 GPU-based Parallel Computation Framework

Challenge 1: *HPS involves on-the-fly Intersection and aggregation, which are time-consuming when the data volume is huge. Improving their efficiency via algorithm optimizations is difficult.*

We design the GPU-based **P**arallel **C**omputation **F**ramework (PCF) to accelerate these operations.By running the algorithms on GPU, the intersection and aggregation can be computed efficiently. More information can be obtained from supplementary materials.

4.3.2 Segmented Curve Fitting: Implement a Regression Model as Many Parallel-Aligned Small Models

Challenge 2: *The key-position curve of a sorted array always show a nonlinear and fluctuating trend (although monotonic increasing). Fitting such a curve with a single model always causes large prediction biases, and requires more model parameters and tedious tuning [32], as in Fig. 7(a).*

Challenge 3: *The single model structure is not efficient in the model update. Once the data is updated, the whole model may need to be retrained with all training samples.*

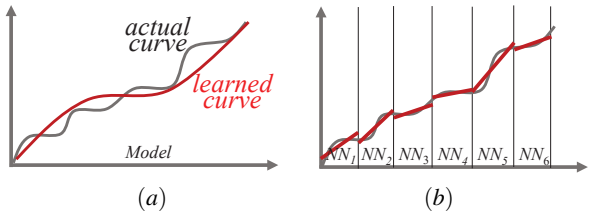


Fig. 7. The basic idea of the segmented curve fitting. (a) Fitting a key-position curve with a single model always causes large prediction biases. (b) We divide the curve into multiple segments and set a small model to fit each segment to improve the prediction accuracy.

We thus propose segmented curve fitting (SCF) that divides an entire value-position curve into a large number of segments and set a group of small models (denoted as NN_i), each fitting a segment of the data curve, as in Fig. 7(b). As the number of divided segments increases, the trend of each segment gradually approximates a straight line. Thus, each small model can have a simple structure with few parameters, which reduces the difficulty of model tuning. The simple structure also enables frequent data update scenario in IDE. For the data change affecting the curve shape, only small models of the affected segments need to be retrained. Thus, the cost of retraining models is significantly reduced.

Fig. 8(a) shows the SCF method. SCF integrates many small models (NN_1-NN_k). Each small model works on one segment. As in Fig. 8(b), (s_1, s_2) is for NN_1, (s_2, s_3) is for NN_2, (s_3, s_4) is for NN_3, and so on. There is a model selector. The model selector assigns a key to a small model for predicting its array position according to value of the key (i.e. the key is assigned to the small model whose segment covers the attribute value of the key). For example, in Fig. 8(a), the queried value is within (s_2, s_3), thus NN_2 is picked and the predicted position is between (p_1, p_2). The model selector is only a thread that keeps all split points, i.e. s_1-s_4.

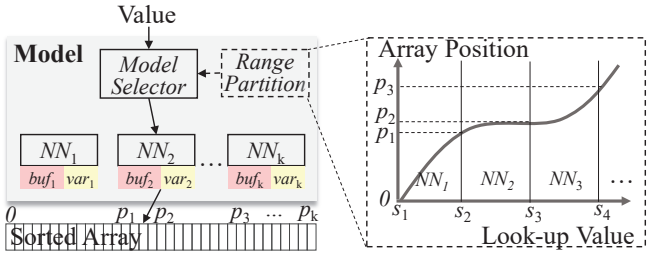


Fig. 8. Segmented curve fitting. (a) A model is implemented as a set of parallel-aligned small models. At the first level, the model selector assigns a key to a small model for predicting the position. (b) The value range of an attribute is equally-divided into multiple segments, each containing an equal number of bins. Keys whose attribute values are in a segment are assigned to the corresponding small model.

As in Fig. 8(b), we make all segments to cover equal value ranges, i.e. (s_1, s_2), (s_2, s_3), (s_3, s_4),..., have the same width (bin interval). Thus, each segment has the same number of bins, making the number of keys (bin indexes) assigned to each small model equal. We can thus use a unified structure for all small models. Moreover, when a small model completes the training session, its weights can be used to initialize the next small model, and so forth. This relaying process accelerates the entire training.

We need to divide the curve into a sufficient number of segments, such that each segment can be approximated as a straight line. Simple linear regression can then be used to capture the linear trend with high

accuracy. The "number of small models" is a parameter that will affect the training time and prediction accuracy. We test these effects in an experiment (Section 6.6).

Each small model has a "buffer" (see pink rectangles in Fig. 8(a)) and a "variable" (see yellow rectangles in Fig. 8(a)) to facilitate model updates. We will introduce their usages and the algorithm to the update of the small models in supplementary material.

SCF has many benefits: (1) Each small model learns a segment of the trend. As the number of small models increases, trends in individual segments become more linear, resulting in higher predicting accuracy. (2) Each small model has a simple structure to fit a short segment. Running such a small model for a query consumes few resources, such as GPU time and memory. (3) For any data change, only the small models affected by the change may need to be retrained, thus avoiding retraining the entire structure and ensuring high update efficiency.

4.3.3 Replace Records with Cells in Sorted Arrays

Challenge 4: *HPS has two types of storage cost, i.e. regression models and sorted arrays. When the dataset contains a large number of records, the sizes of sorted arrays are still huge.*

We thus store cells that contain at least one record instead of original data records in sorted arrays. A cell has 1-bin width on each attribute. In Fig. 9, $c_1 - c_5$ are five 2D cells. Because many records are in the same cells, the number of cells is much less than the original records, thus taking up less space. This design works for both sparse and dense datasets. For sparse datasets, because records are in fewer cells, more space can be saved. Fig. 9 shows a case of storing five cells rather than eight records in sorted arrays.

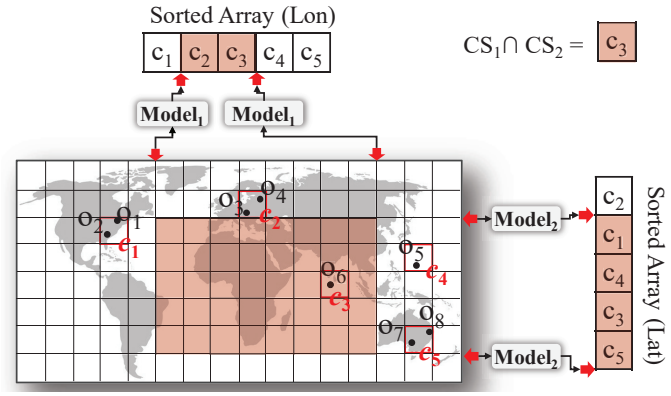


Fig. 9. Replacing eight records with five cells in sorted arrays to save space. We thus run each model twice to find a set of cells rather than original records. CS_1 and CS_2 are two cell sets found by executing the two 1D queries.

We need to calculate one or more cell attributes for each cell based on its inner records according to what the views will show. For example, if the view shows "Count" patterns, we need to calculate the number of records in each cell as a cell attribute. We can get the measure values by performing algebraic operations based on the selected cells. Note that replacing records with cells is optional, which does not affect the display/query effects because a cell is the smallest unit for query and display. Records in a cell will overlap and should be selected or unselected together in views.

5 Implementation

Each small model fits a segment of a curve with an approximately linear trend. We thus implement each small model as a neural network without any hidden layer, which is equivalent to a linear regression model. We can directly use linear regression or other classic machine learning techniques to achieve the same effects. We choose neural networks since developers can conveniently add a hidden layer where the curve has an extremely irregular trend. In most case, a dozen of neurons per layer is sufficient to ensure high accuracy, while training such a neural network takes only a few seconds (or $<1s$). Even with hundreds of thousands of neural networks, the overall storage cost and training time are still acceptable.

The MSE (Mean Squared Error) loss and ReLu activation function are used. Optimization strategies, such as learning rate decay and dropout, are optional. The initial search range threshold is set to 512 by default. The training of each small model can be terminated early when the average prediction bias is less than the threshold to accelerate the SCF building. In general, all used parameters are standard, integrated into existing deep learning frameworks.

6 Evaluation

The evaluation consists of three parts. First, we use HPS to develop visualization systems with different display/query functions (Section 6.2). Second, we test the storage usage, training time and execution speed of HPS, and compare HPS's measures to those of Nanocubes (Sections 6.3-6.5). Finally, we analyze the impacts of the parameter, i.e. the number of small models in a single regression model, on HPS's performance (Section 6.6). All the experiments are conducted on a GPU Server (XEON E5-2680, 196G, 2080Ti×4).

6.1 Experiment Preparation

Dataset. We collect six real-world open datasets and one synthetic dataset in evaluation, including: (1) brightkite social media checkins [6], (2-3) New York taxi trajectories (yellow and green), (4) Chicago crime records, (5) gowalla social media dataset, (6) US flight on-time statistics, and (7) ScatterPlot matrix (SPLOM) synthetic dataset. Table 1 summarizes the details of the first six datasets. The SPLOM dataset is synthesized according to the setting of imMens [22]. Among the 6 real-world datasets, (2), (3) and (4) have relatively dense distributions, while the other 3 are sparse.

Attribute Selection. We select 5-7 attributes for each dataset, as in Table 1. Four datasets (brightkite, crime, gowalla and flight) have the same attributes as Nanocubes [19], while those of other two datasets (taxi-yellow and -green) are the same as Hashedcubes [29].

Discretization. We uniformly divide longitudes and latitudes of all the six datasets into 2^{25} bins. Temporal attributes are discretized according to the natural division of time. Other attributes have the same binning granularities as [19] or [29], as in Table 1.

Model and Sorted Array. We train models for longitudes and latitudes. These attributes are all divided into 2^{25} bins. Other temporal attributes use hash-maps. A sorted array is created for each attribute. We store cells rather than original records in a sorted array.

Collection of Training Samples. We collect all key-value pairs as the training samples of a model. Since longitudes and latitudes are divided into 2^{25} bins, each model has about 33M training samples.

The Number of Small Models. According to the experimental results, we uniformly set 20K small models for longitudes and latitudes of all the datasets, which can achieve relatively good performance in most cases (Section 6.6). Each small model thus has about 1677 training samples ($2^{25}/20K$).

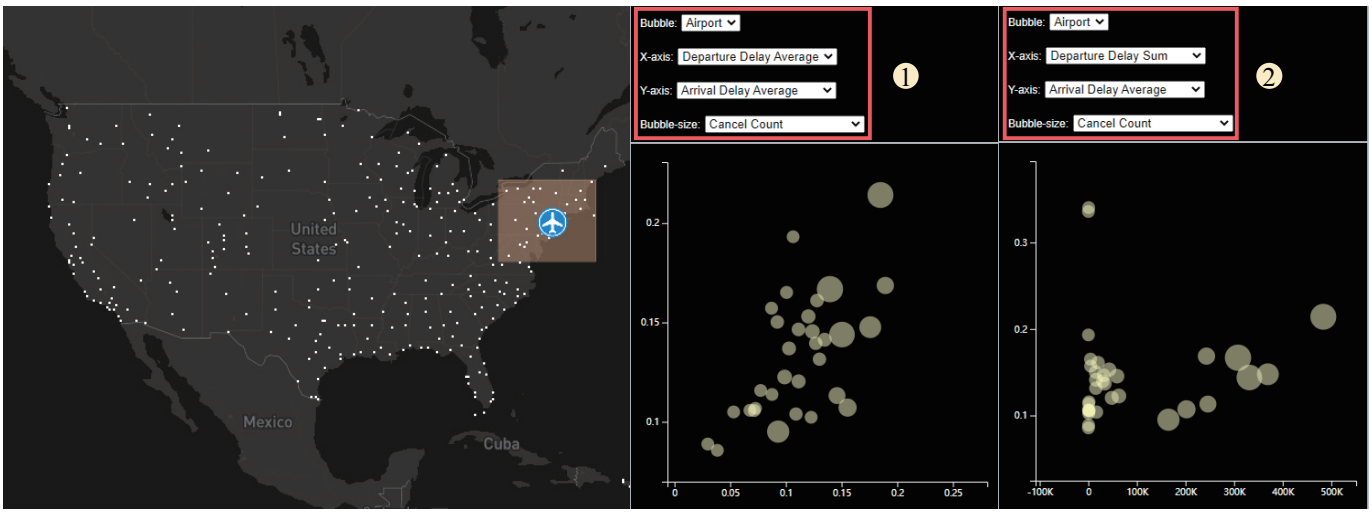
Training Setting. Each small model is implemented as a neural network without hidden layer. All the small models have the same structure: a 1×20 input layer and a 20×1 output layer, 40 weights in total. The MSE (Mean Squared Error) loss is used. The learning rate is set to 0.05, and the batch size is set to 16. The learning rate decay is used to change the learning rate to 0.7 times of the last one every five epochs. The initial search range threshold is set to 512. The training stops whenever its epoch number reaches 120, or the average actual prediction bias is less than 512.

6.2 Visualization Systems

We develop three visualization systems, each with a unique display/query function, on three representative real-world datasets to demonstrate HPS's better use flexibility, as follows:

Showing both aggregate patterns and original records. We constructed a system on the Brightkite dataset. It contains a map and four views showing temporal patterns, as in Fig. 10(a). The map can show

(a)



(b)

Fig. 10. Two IDE systems developed using HPS. (a1) A system integrates five views showing aggregate patterns of the brightkite dataset. The map in (a1) can be switched to show the filtered records' positions directly (a2). (b) A system for analyzing the US flight dataset, in which users can interactively change the mappings of scatters and two axes in the scatter-plot during the exploration.

Table 1. Dataset information and the storage cost summary.

Records(N)		Interaction schema (2^N)	Model(N)	Cells(N)	ModelSize	ArraySize	Σ
brightkite	4.5M	lon(25), lat(25), day-of-week(3), hour-of-day(5), time(16)	2	3.5M	6.1MB	66.5MB	72.6MB
taxi-yellow	258.1M	lon1(25), lat1(25), lon2(25), lat2(25), day-of-week(3), hour-of-day(5), time(10)	4	249.3M	12.2MB	6.5GB	6.5GB
taxi-green	28.3M	lon1(25), lat1(25), lon2(25), lat2(25), day-of-week(3), hour-of-day(5), time(10)	4	28.1M	12.2MB	751.1MB	763.3MB
crime	6.8M	lon(25), lat(25), type(6), day-of-week(3), hour-of-day(5), time(13)	2	6.6M	6.1MB	152.2MB	158.3MB
gowalla	6.4M	lon(25), lat(25), hour-of-day (5), day-of-week (3), time (15)	2	6.1M	6.1MB	116.9MB	123.0MB
flight	123.5M	lon(25), lat(25), time(16), carrier(5), takeoff-delay(4)	2	43.5M	6.1MB	829.0MB	835.1MB

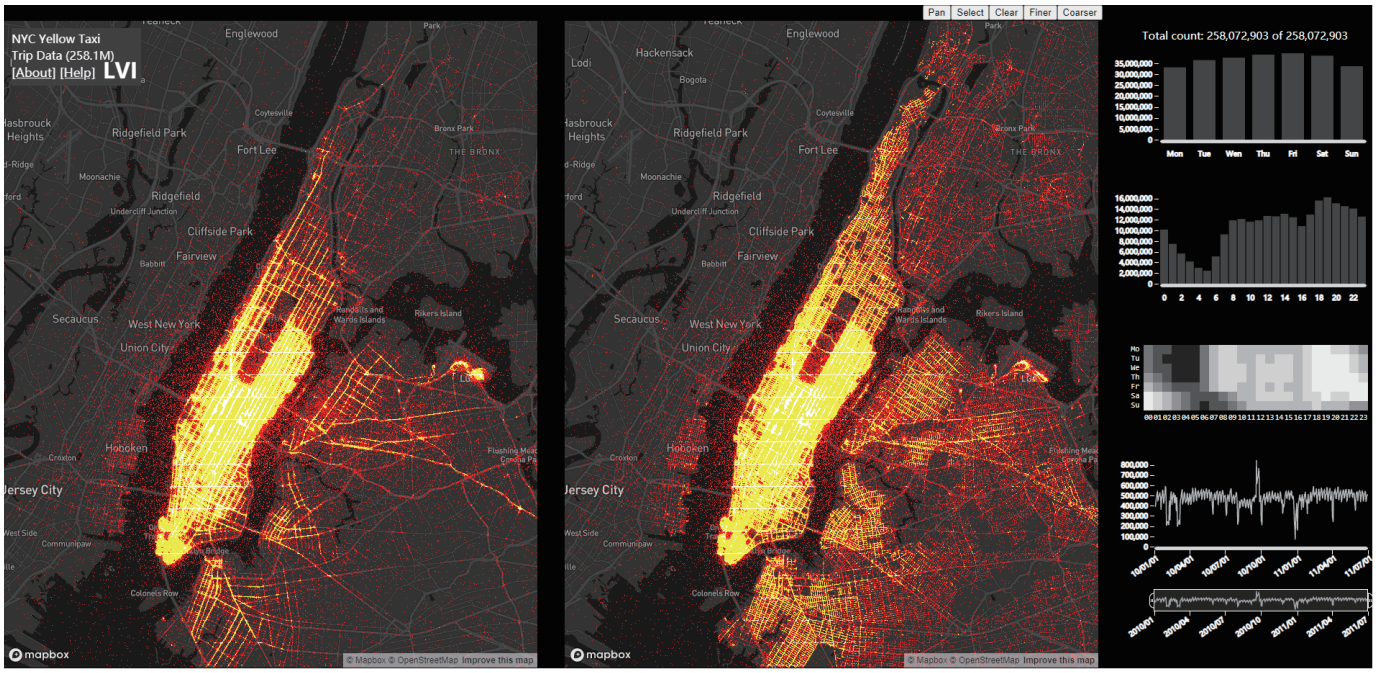


Fig. 11. A system for analyzing O-D patterns of the taxi-yellow dataset. Longitudes and latitudes of the two maps are uniformly divided into 2^{25} bins to enable high-resolution query/display.

aggregate spatial patterns (Fig. 10(a1)) or filtered records' distribution (Fig. 10(a2)).

Changing aggregate measures interactively. We develop a system on the US flight dataset that records on-time statistics. As in Fig. 10(b), the system includes a map showing airports' positions and a scatter-plot showing the flight delay situations. Notably, users can interactively change what scatters and axes map in the scatter-plot. For example, each scatter can be an airport, a state, or an airline; the scatter size and the two axes can map different aggregate measures, such as the count of departures, the average of arrival delays, etc.

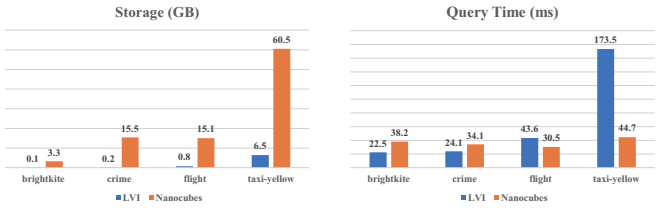
Querying on more finely-binned attributes.We develop a visualization system to enable O-D analysis on the yellow-taxi dataset with 258.1M records. The interface is shown in Fig. 11, which has two maps for users to select origins and destinations of taxi trips, and four small views showing temporal patterns. Latitudes and longitudes of the two maps are uniformly divided into 2^{25} bins. That is users can query on four finely-binned attributes at the same time.

6.3 Memory Usage

We record the storage costs of HPS constructed on six open datasets. Table 1 shows the dataset information and the storage costs of HPS on different datasets. The number of data records and the interaction schema are in the first two columns. "Model(N)" indicates the numbers of SCFs. "Cells(N)" represents the number of cells in a sorted array. "ModelSize" and "ArraySize" indicate the sizes of all models and sorted arrays, respectively. "Σ" represents the overall storage cost, i.e. the sum of "ModelSize" and "ArraySize".

In general, the storage cost of HPS is low. A model is about several MBs. Sorted arrays take up most of the storage, ranging from tens of MBs to several GBs, depending on (1) the number of cells, (2) the number of attributes, and (3) the data distribution. Specifically, more records result in more cells, and the number of attributes determines that of sorted arrays to be stored, which explains why taxi-yellow, taxi-green, and flight datasets have significantly higher storage costs than those of the other three datasets. Moreover, data distribution slightly affects this aspect. Two datasets with similar numbers of records may have different storage costs, since their records can be in different numbers of cells, see the crime and gowalla datasets in Table 1.

Fig. 12(a) compares the sizes of HPS and Nanocubes [19] on the four datasets which are representative with varying scales of records. As expected, HPS are significantly smaller on all the datasets. Most existing structure-based techniques are designed to store aggregate values of attribute combinations, unnecessary for HPS.



(a) (b)

Fig. 12. Comparison of HPS with Nanocubes on (a) storage cost and (b) execution time. HPS is significantly lower than Nanocubes in storage and has a faster execution speed on million-scale datasets. However, as the number of records increases, the query speed of HPS will decrease. We use Nanocubes of V4, larger than V1 used in [19] since V4 treats different types of attributes (spatial, temporal, categorical, numerical) uniformly. We test the execution time using a single-GPU condition.

6.4 Training Time

Table 2 shows the training time of a single SCF. We set different initial search range thresholds (training is terminated when the current average bias is less than the threshold, Section 4.3.2) and observe the training time variations.

Table 2. Training time of a single SCF under different initial search range thresholds (512 by default for other tests). Numbers in parentheses are percentages of keys with biases larger than the threshold.

	1024 (>*thres*)	**512** (>*thres*)	256 (>*thres*)	128 (>*thres*)
brightkite	1.5h (0.30%)	**1.9h (0.91%)**	2.6h (1.36%)	3.8h (1.54%)
taxi-yellow	1.6h (0.16%)	**1.7h (0.19%)**	1.8h (0.21%)	1.8h (0.26%)
taxi-green	1.7h (0.16%)	**1.7h (0.17%)**	1.7h (0.18%)	1.7h (0.19%)
crime	1.5h (0.18%)	**1.5h (0.20%)**	1.5h (0.21%)	1.5h (0.21%)
gowalla	1.5h (0.39%)	**2.1h (1.11%)**	2.9h (1.59%)	4.0h (1.83%)
flight	2.1h (1.28%)	**2.5h (1.35%)**	2.5h (1.42%)	2.6h (1.46%)

Table 2 shows the experiment results. We find that SCF's training is fast. Specifically, the training time is between 1-3 hours when using the default initial search range threshold 512, as many training sessions can be terminated quickly, no need to execute all the epochs. Thus, we can set a larger threshold to reduce the training time further.

The percentages of keys with prediction biases larger than the initial search range threshold are low (see numbers in parentheses in Table 2), implying high overall accuracy. Like the training time, the percentage is also inversely proportional to the initial search range threshold, which can be further reduced by setting a larger search range threshold.

6.5 Execution Speed

We test the execution time of PCF with different numbers of GPUs. We find that the execution of HPS can be further improved by using more GPUs. See supplementary materials for details.

We then extract 10000 actual user queries from logs of the visualization systems and test the execution time of HPS in real world scenarios by running them. We require all the selected attribute ranges to cover hot places of different zoom levels, such as countries, cities, blocks, etc., and diverse periods. The task is the same as RSATree [23].

Table 3. Execution time of HPS consisting of three parts: position prediction and search, copy cells from main memory to GPU, and calculation of intersection and aggregation (at initial search range threshold of 512).

		Prediction	Copy	Calculation	Total
brightkite	Mean	1.2μs	1.0ms	10.9ms	22.5ms
	Max	10.0μs	4.0ms	19.1ms	36.1ms
	Std	3.8μs	1.1ms	2.8ms	4.5ms
	Median	1.4μs	0.8ms	11.0ms	22.7ms
taxi-yellow	Mean	0.8μs	11.7ms	146.8ms	173.5ms
	Max	9.9μs	20.0ms	296.0ms	335.0ms
	Std	3.2μs	4.9ms	17.4ms	23.8ms
	Median	1.1μs	12.0ms	145.5ms	172.9ms
taxi-green	Mean	1.3μs	2.8ms	30.4ms	46.8ms
	Max	10.6μs	7.0ms	46.9ms	65.4ms
	Std	4.6μs	1.9ms	3.9ms	5.3ms
	Median	1.4μs	2.6ms	30.0ms	46.6ms
crime	Mean	0.8μs	1.2ms	12.2ms	24.1ms
	Max	10.0μs	4.7ms	25.0ms	43.0ms
	Std	2.7μs	1.4ms	3.1ms	3.9ms
	Median	0.8μs	1.1ms	12.2ms	24.3ms
gowalla	Mean	1.8μs	1.4ms	12.4ms	24.8ms
	Max	18.0μs	5.1ms	29.1ms	45.2ms
	Std	4.0μs	2.0ms	3.5ms	4.1ms
	Median	1.5μs	1.5ms	14.8ms	26.0ms
flight	Mean	1.4μs	4.1ms	30.8ms	43.6ms
	Max	10.0μs	18.2ms	42.9ms	72.2ms
	Std	3.5μs	2.4ms	3.2ms	3.9ms
	Median	0.9μs	11.9ms	31.0ms	48.5ms

We test the query speed using a single GPU. The execution time has three parts, i.e. predicting and searching positions, copying cells from main memory to GPU, and calculating intersection and aggregation, as in Table 3. The total time overhead also includes a certain amount of network latency. In general, the total time is short, supporting our claim, i.e. HPS supports interactive data exploration [21]. Specifically, prediction and search take almost a constant time, while the other two take longer when datasets contain more cells.

We also establish Nanocubes on four different datasets to compare Nanocubes' execution speed with HPS. We deploy all the established Nanocubes in the GPU server and execute the same 10000 queries on Nanocubes and HPS (using a single GPU). The result is shown in Fig. 12(b). In general, HPS executes a little faster than Nanocubes on million-scale datasets. However, HPS is slower than Nanocubes on larger datasets. Moreover, the speed of Nanocubes is almost constant on different datasets. In contrast, HPS involves on-the-fly calculations, resulting in varying query time on datasets of different scales.

We finally test HPS's query speed under different numbers of GPUs (using the same 10000 queries). We find the average query time can greatly decrease by using multiple GPUs for larger datasets, which illustrates HPS's better scalability. Details can be found in the supplementary material.

6.6 Parameter Setting

Our model involves a hyperparameter, i.e. the number of small models in a single SCF. In order to help users set this parameter, we conduct a quantitative experiment. The results show that the difference in training time of different numbers of small models is not large. It is feasible to set tens of thousands of small models for tens of millions of queriable keys. Details can be found in the supplementary material.

7 Limitation Analysis

We now discuss the limitations of HPS and the effects on applying HPS in real-world scenarios, as follows:

Query speed deterioration. As the number of records increases, the response speed of queries will decrease, interactive operations may become less smooth. Although using more GPUs can mitigate this problem, it cannot change the fact that HPS is not a technique with $O(1)$ query speed. A good aspect is that HPS can achieve comparable performance to existing techniques on million-scale datasets on a single GPU, while the query speed can be further improved by using more GPUs, illustrating HPS's usability in most common scenarios. However, HPS's query speed is not always slow, even for an extremely huge dataset. The response time of a query depends on the number of selected records. Users always explore local patterns (e.g. specific cities, blocks, hour-of-days, day-of-weeks, etc.) after seeing an initial overview. The frequency of selecting large attribute ranges, such as the entire map or period, to include all records is relatively low. Therefore, most queries in visualization systems involve parts of records and are fast to execute.

GPU dependence. HPS depends on GPU to train models and accelerate calculations. It is a trend for IDE techniques to utilize GPUs. For example, Falcon [27] depends on a GPU-based database to accelerate the calculation of aggregate values. Of course, we can implement and apply HPS in devices without GPU when hard real-time is not necessary. We can exclude all the models and find correct positions of queried attribute values through the binary search. The search time thus depends on the number of records. We can also use CPU-based multi-thread algorithms to accelerate the intersection and aggregation.

Requiring model training expertise. HPS requires developers to know neural networks (or linear regression) to set parameters, such as learning rate, number of iterations, activation function, etc. However, HPS only uses ordinary neural networks without hidden layer. The simple structure reduces the difficulty of model tuning. In most cases, users can achieve good performance through a few trials. Besides, the parameters we have given is reusable in most common cases, although not always optimal.

8 Conclusion and Future Work

This paper has presented a novel IDE technique. Unlike existing techniques using either pre-storage or learning-based methods, our approach follows a hybrid strategy to obtain the advantages of both types of techniques, which, according to our knowledge, is the first attempt in visualization. Our approach has smaller storage overheads and better use flexibility than existing techniques but involves intersection and aggregation for each query. We thus design a GPU-based framework to accelerate these on-the-fly operations. Our approach is not for improving existing techniques. Instead, its technical characteristics are suitable for scenarios with special hardware conditions and visualization requirements, such as low memory devices and querying on multiple high-resolution views. Our approach brings a new research idea, which may promote the emergence of more works that combine traditional data structures and learning models for efficient IDE.

In the future, we plan to make two improvements. First, we will further test the scalability of HPS on larger datasets. Second, we will use other AI techniques (e.g., CNN, RNN, and GNN) to support more complex visualization, such as trees and graphs.

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*, pp. 29–42. ACM, 2013.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, Mar. 1997.

[3] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 511–519, 2017.

[4] C. Chen, C. Wang, X. Bai, P. Zhang, and C. Li. Generativemap: Visualization and exploration of dynamic density maps via generative learning model. *IEEE transactions on visualization and computer graphics*, 26(1):216–226, 2019.

[5] Z. Chen, W. Zeng, Z. Yang, L. Yu, C.-W. Fu, and H. Qu. Lassonet: Deep lasso-selection of 3d point clouds. *IEEE transactions on visualization and computer graphics*, 26(1):195–204, 2019.

[6] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1082–1090. ACM, 2011.

[7] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: interactive analytics through pen and touch. *Proceedings of the VLDB Endowment*, 8(12):2024–2027, 2015.

[8] D. Fisher, I. Popov, S. Drucker, et al. 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*, pp. 1673–1682. ACM, 2012.

[9] S. Ghosh and A. Eldway. Aid*: A spatial index for visual exploration of geo-spatial data. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[10] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD Record*, 28(2):287–298, 1999.

[11] W. He, J. Wang, H. Guo, K.-C. Wang, H.-W. Shen, M. Raj, Y. S. Nashed, and T. Peterka. Insitunet: Deep image synthesis for parameter space exploration of ensemble simulations. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):23–33, 2019.

[12] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis: The control project. *Computer*, 32(8):51–59, 1999.

[13] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In *2014 IEEE 30th International Conference on Data Engineering*, pp. 472–483. IEEE, 2014.

[14] T. Kraska. Northstar: An interactive data science system. 2021.

[15] B. C. Kwon, J. Verma, P. J. Haas, and C. Demiralp. Sampling for scalable visual analytics. *IEEE Computer Graphics and Applications*, 37(1):100–108, 2017.

[16] 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.

[17] J. K. Li and K.-L. Ma. P5: Portable progressive parallel processing pipelines for interactive data analysis and visualization. *IEEE transactions on visualization and computer graphics*, 26(1):1151–1160, 2019.

[18] M. Li, F. M. 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.

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

[20] C. Liu, C. Wu, H. Shao, and X. Yuan. Smartcube: An adaptive data management architecture for the real-time visualization of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 2019.

[21] 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, 2014.

[22] Z. Liu, B. Jiang, and J. Heer. immens : real-time visual querying of big data. *eurographics*, 32:421–430, 2013.

[23] H. Mei, W. Chen, Y. Wei, Y. Hu, S. Zhou, B. Lin, Y. Zhao, and J. Xia. Rsatree: Distribution-aware data representation of large-scale tabular datasets for flexible visual query. *IEEE Transactions on Visualization and Computer Graphics*, 2019.

[24] 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.

[25] 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, 2017.

[26] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pp. 2904–2915, 2017.

[27] D. Moritz, B. Howe, and J. Heer. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pp. 1–11, 2019.

[28] C. A. Pahins, N. Ferreira, and J. L. Comba. Real-time exploration of large spatiotemporal datasets based on order statistics. *IEEE transactions on visualization and computer graphics*, 26(11):3314–3326, 2019.

[29] C. A. Pahins, S. A. Stephens, C. Scheidegger, and J. L. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE Transactions on Visualization & Computer Graphics*, 23(1):671–680, 2016.

[30] S. Rahman, M. Aliakbarpour, H. K. Kong, E. Blais, K. Karahalios, A. Parameswaran, and R. Rubinfield. I've seen enough: incrementally improving visualizations to support rapid decision making. *Proceedings of the VLDB Endowment*, 10(11):1262–1273, 2017.

[31] R. A. Rossi, N. K. Ahmed, R. Zhou, and H. Eldardiry. Interactive visual graph mining and learning. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 9(5):1–25, 2018.

[32] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[33] C. Turkay, N. Pezzotti, C. Binnig, H. Strobelt, B. Hammer, D. A. Keim, J.-D. Fekete, T. Palpanas, Y. Wang, and F. Rusu. Progressive data science: Potential and challenges. *arXiv preprint arXiv:1812.08032*, 2018.

[34] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. S ee db: efficient data-driven visualization recommendations to support visual analytics. *Proceedings of the VLDB Endowment*, 8(13):2182–2193, 2015.

[35] Z. Wang, D. Cashman, M. Li, J. Li, M. Berger, J. A. Levine, R. Chang, and C. Scheidegger. Neuralcubes: Deep representations for visual data exploration. In *2021 IEEE International Conference on Big Data (Big Data)*, pp. 550–561. IEEE, 2021.

[36] Z. Wang, N. Ferreira, Y. Wei, A. S. Bhaskar, and C. E. Scheidegger. Gaussian cubes: Real-time modeling for visual exploration of large multi-dimensional datasets. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):681–690, 2017.

[37] C. Xie, W. Zhong, W. Xu, and K. Mueller. Visual analytics of heterogeneous data using hypergraph learning. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(1):1–26, 2018.

[38] T. Xu, X. Zhang, C. Claramunt, and X. Li. Tripcube: A trip-oriented vehicle trajectory data indexing structure. *Computers, Environment and Urban Systems*, 67:21–28, 2018.

[39] E. Zgraggen, A. Galakatos, A. Crotty, J.-D. Fekete, and T. Kraska. How progressive visualizations affect exploratory analysis. *IEEE transactions on visualization and computer graphics*, 23(8):1977–1987, 2016.

[40] Y. Zhao, J. Zhang, C.-W. Fu, M. Xu, D. Moritz, and Y. Wang. Kd-box: Line-segment-based kd-tree for interactive exploration of large-scale time-series data. *IEEE Transactions on Visualization and Computer Graphics*, 2021.

Supplementary Material

1. Parallel Computation Framework

1.1 GPU-based Intersection and Aggregation

Both intersection and aggregation are time-consuming when the data volume is large. Improving their efficiency via algorithm improvement is difficult. Therefore, we design the **P**arallel **C**omputation **F**ramework(PCF) to accelerate them via GPU-based algorithms. The framework adopts a divide-and-conquer strategy, partitioning each computational operation into multiple independent sub-operations, and starts multiple threads to execute these sub-operations in parallel, thereby reducing the time overhead. A GPU can always have thousands of CUDA cores, and each CUDA core can run thousands of threads simultaneously [1], allowing the framework to support parallel execution of a large number of such sub-operations. Moreover, the framework can automatically assign sub-operations to multiple GPUs, further increasing concurrent threads. Specifically, the framework involves three kinds of operations, as follows:

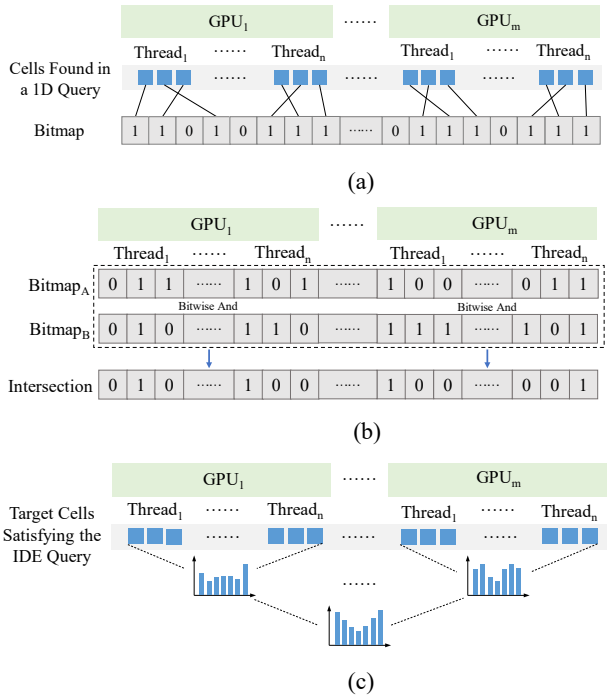


Fig. 1. Three kinds of operations in the Parallel Computation Framework, i.e. (a) bitmap generation, (b) bitwise AND-based intersection, and (c) aggregation. All three operations enable multi-GPU-based parallelization.

Bitmap Generation. We create a bitmap [2] for records (or cells) found in a 1D query. The length of the bitmap is equal to the number of all records (or cells) of a sorted array. We traverse all found records (or cells) in the 1D query and map each record (or cell) to a specific bit (set the bit 1). We can divide records (or cells) into multiple parts and execute traversal of each part using a thread for acceleration, as in Fig. 1(a). This step will form multiple bitmaps, each corresponding to a decomposed 1D query.

Bitwise AND-based Intersection. We perform a bitwise AND operation to get the intersected records (or cells) for each pair of bitmaps, as in Fig. 1(b). A bitwise AND operation can be divided into multiple sub-operations on divided shorter bitmaps for parallelization, and reduces the memory overhead by encoding each record (or cell) as a bit. This step will form a single bitmap encoding target records (or cells) of the IDE query.

Aggregation. An aggregation operation can be divided into several sub-operations running on independent threads, as in Fig. 1(c). Outputs of sub-operations can be further aggregated until obtaining desired aggregate values. This recursive structure can make full use of the computing power of each GPU.

1.2 Experiments on the Parallel Computing Framework

We calculate the intersection of randomly-generated cells using the PCF and test the execution time when using different numbers of GPUs. Experiment results are in TABLE 1. We find the time overhead significantly decreases when using more GPUs in most cases. However, for smaller datasets (the first row in TABLE 1), using more GPUs leads to the slight increase of the time overhead since single GPU is sufficient to make the operations fully-parallelized. Adding GPUs brings more synchronization and scheduling overhead.

TABLE 1

Time overhead of intersection operations on randomly-generated cells.

	1 GPU (ms)	2 GPUs (ms)	3 GPUs (ms)	4 GPUs(ms)
10 million	4	4	4	5
50 million	26	14	10	8
100 million	50	28	18	14
200 million	92	55	37	30
500 million	233	124	90	47
1 billion	476	244	178	122
2 billion	996	538	349	249

We also test HPS's query speed under different numbers of GPUs (using the same 10000 queries mentioned in the manuscript). Fig. 2 shows the results. We find the average query time can greatly decrease by using multiple GPUs for larger datasets, i.e. taxi-yellow, taxi-green, and flight, illustrating HPS's better scalability. However, the other three million-scale datasets' query time, i.e. brightkite, gowalla, and crime, increase slightly. This result also supports the conclusion we made about the PCF, that using a single GPU already makes the computation of millions of records fully parallel while adding GPUs will bring additional time overhead in device scheduling and synchronization.

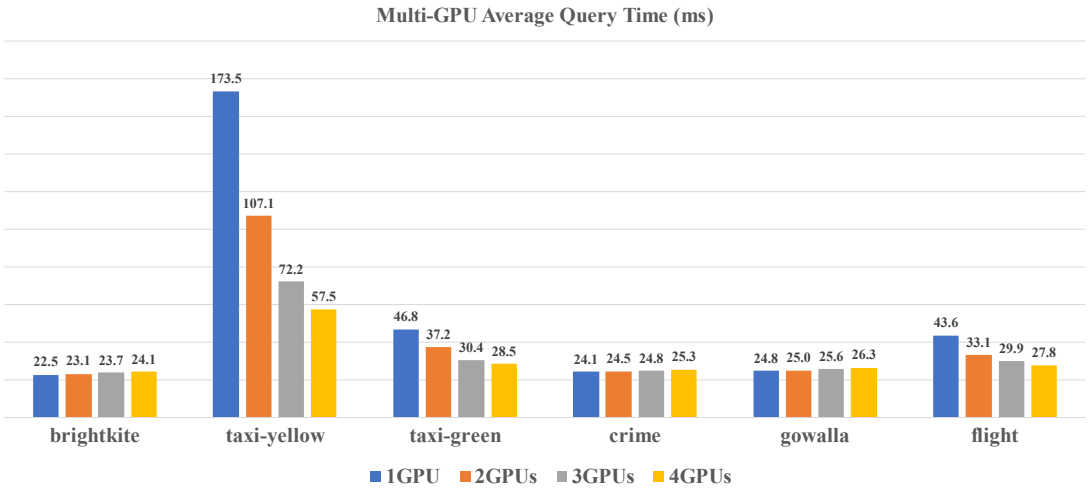


Fig. 2. The average query time of the six open datasets when using different numbers of GPUs.

2. Parameter Setting

We test the effects of the parameter, i.e. the number of small models in a single SCF, on HPS's training time and prediction accuracy. The experiment is conducted on three real-world datasets (brightkite, taxi-green, taxi-yellow) and one synthetic dataset (SPLOM). The four datasets vary significantly in data amount. We select one attribute from the three real-world datasets(lon for brightkite, lon1 for green-taxi and yellow-taxi), and generate a normally-distributed attribute for the SPLOM dataset. All the selected attributes are divided into 2^{25} bins. We test their performances with different numbers of small models (5K, 10K, 20K, 50K 100K). Experiment results are shown in TABLE 2.

TABLE 2

The training time and prediction accuracy when setting different numbers of small network in each SCF. The best result in each column is marked red, and the initial search range threshold is 512.

	Brightkite		Green-Taxi		Yellow-Taxi		Splom	
Number of NNs	Training Time(h)	Samples beyond thres(%)	Training Time(h)	Samples beyond thres(%)	Training Time(h)	Samples beyond thres(%)	Training Time(h)	Samples beyond thres(%)
5k	2.86	0.53	1.70	0.09	1.82	0.19	1.57	0.05
10k	2.18	0.25	1.67	0.07	1.77	0.13	1.56	0.03
20k	1.94	0.12	1.69	0.06	1.75	0.10	1.60	0.02
50k	1.84	0.04	1.80	0.06	1.86	0.07	1.72	0.01
100k	1.87	0.02	1.86	0.05	1.89	0.05	1.80	0.01

A common trend for all datasets is the prediction accuracy (the percentage of samples with biases smaller than the initial search threshold) gradually increases as the number of small models and peaks at 100K. However, the optimal settings to achieve the fastest training speeds vary greatly for different datasets, while neither too few (5K) nor too many (100K) small models can achieve fast training speeds. Thus, we set 20K small models for all experiments mentioned in the paper, which

achieves good results in most cases. We should note that it is challenging and unnecessary to find the optimal value of the parameter. As long as the value is within a roughly reasonable range, HPS will perform well. The experiment results show that the difference in training time of different numbers of small models is not large. It is feasible to set tens of thousands of small models for tens of millions of queriable keys.

3. Update Algorithm

3.1 Update Optimization

We introduce the usage of the "buffer" and "variable" mentioned in the paper (Section 4.3.3).

First, each small model has a "buffer" (see pink rectangles in Fig. 3) to facilitate data updates. The buffer stores all keys with biases larger than the initial search range threshold and their true positions. Each buffer (organized as a hash-map) can immediately return the true position of a queried attribute value. The execution process of a small model is changed with the buffer. Before running a small model, we first check whether the queried attribute value is in the buffer. If so, the true position can be fetched from the buffer immediately without running the small model. The retraining is triggered only when the buffer is full (see Algorithm 1). The buffer can reduce the retraining frequency and avoid performance degradation before the retraining.

Second, each small model has a "variable" (see yellow rectangles in Fig. 3). Each variable stores the array position corresponding to the start of the segment assigned to the small model (var_1 is 0, var_2 is p_1, and so on). Each small model only needs to predict the position offset from the starting position. For example, the value range of outputs of NN_2 is changed from $(p_1,\ p_2)$ to $(0,\ p_2 - p_1)$. This narrows the value range of labels, thus improving the prediction accuracy. It also improves update efficiency. The small models behind the inserting position of the new cell only need to modify their variable values (lines 2-3 in Algorithm 1)

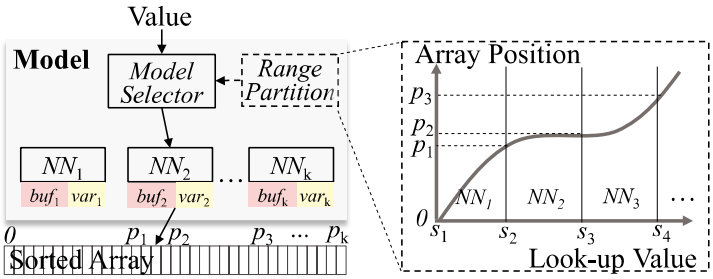


Fig. 3. Parallel Index Structure. (a) A model is implemented as a set of parallel-aligned small models. (b) The value range of an attribute is equally divided into multiple segments, each containing an equal number of bins. Attribute values in a segment are assigned to a small model.

3.2 Update Algorithm

We introduce the model update details due to a data change, which involves three types of scenarios.

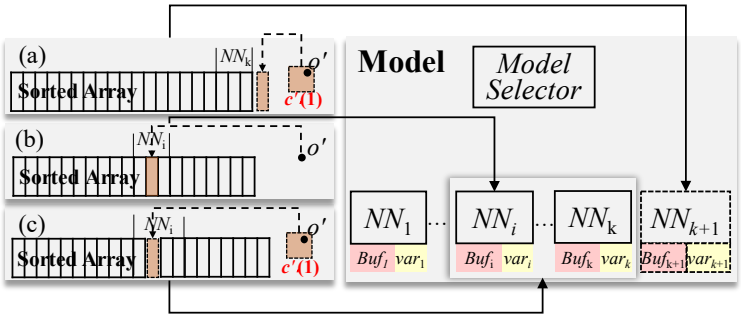


Fig. 4. Three update scenarios when receiving a new record o'.

New Cell Appending. Fig. 4(a) shows the new record o' falling in a new cell being appended to the end of the sorted array. We only need to train a new small model, and existing small models are not affected. We can first establish the small model buffer and put value-position pairs of the new cell in it. The new small model is trained when the buffer is full.

Cell Updating. Fig. 4(b) shows that the new record's cell may already have records and is in the sorted array. In that case, we only need to update the cell's attribute value, without a need to retrain any small model.

New Cell Inserting. Fig. 4(c) shows the new record o' falling in a new cell that should be inserted in the middle of the sorted array. This scenario involves a more complex update process. Let *model[]* be the set of all small models, *target_model* is the index of the small model whose inputs(i.e. the queriable keys) cover the new cell. The update process is in Algorithm 1.

We first find the *target_model* (line 1). The small models after *target_model* are not retrained. Instead, their outputs are shifted one step to the right to adapt to the array change (lines 2-3). The training samples of *target_model* with biases larger than the threshold are stored into the buffer (lines 4-6). When the buffer is full, we clear it and retrain the *target_model* (lines 7-9).

Algorithm 1: Index Update

Input: int train_set[], model[], new cell/deleted_cell

Output: start_pos[], buff[]

```
target_model = model_selector(new_cell/deleted_cell);
for i from target_model+1 to k do
    start_pos[i]  ±= 1 (+ for inserting, - for deleting);
for data in train_set[target_model] do
    if model[data.key] ≥ threshold then
        buff[target_model].add(data);
if sizeof(buff[target_model]) ≥ threshold2 then
    buff[target_model].clear();
    call retrain_process(train_set[target_model]);
return start_pos and buff;
```

The algorithm improves the update efficiency in three aspects. First, only two of the three scenarios need to retrain small models. Second, each small model has a small number of training samples, making the training process fast. Third, retraining is triggered only when the buffer is full. A larger buffer could reduce retraining frequency.

The parallel index structure also supports the update for deleting cells. In that case, we may only need to retrain the small model that covers the deleted cell, see Algorithm 1.

3.3 Update Efficiency

We used the six real-world datasets to test the update efficiency, as in TABLE 3. For each dataset, we take out the last 200K records to evaluate the update time when different amounts of records are added back. Specifically, we set the buffer size to 100, and test the update time and numbers of neural networks to be retrained when 10K, 50K, 100K and 200K records are added back. Buffer size of 100 is a strict condition. Better results can be obtained by using a larger buffer.

TABLE 3

Update times when 10K, 50K, 100K and 200K records are added. Numbers in parentheses are those of neural networks (small models) to be retrained. The buffer size is 100.

	10K (N)	50K (N)	100K (N)	200K (N)
brightkite	0.2h (120)	0.5h (178)	0.7h (186)	0.9h (210)
taxi-yellow	0.2h (109)	0.5h (113)	0.5h (113)	0.5h (115)
taxi-green	0.3h (118)	0.6h (125)	0.6h (126)	0.6h (126)
crime	0.1h (42)	0.2h (52)	0.2h (58)	0.3h (62)
gowalla	0.3h (130)	0.4h (148)	0.4h (153)	0.6h (170)
flight	0.6h (172)	0.7h (187)	0.8h (196)	1.0h (201)

As in TABLE 3, the updating time is much less than rebuilding the entire index. Moreover, the

number of neural networks to be retrained is small even when many records arrive due to the Parallel Index Structure.

4. Storage Configuration

There are mainly three manners for deploying HPS and sorted arrays in a computer as in Fig. 5.

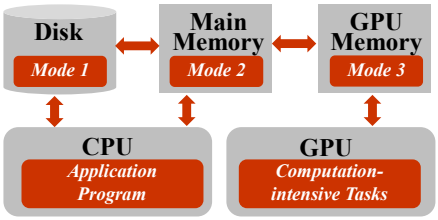


Fig. 5. Three types of storage configurations. Mode1 refers to the disk mode. Mode2 refers to the memory mode. Mode3 refers to the GPU mode.

Disk Mode. HPS is stored on the hard disk. There is no memory cost for this mode. For each query, the selected neural networks and the filtered parts of the sorted arrays are dynamically loaded into the main memory for the following calculation. Because no data traversal is needed, the main performance bottleneck derives from I/O operations.

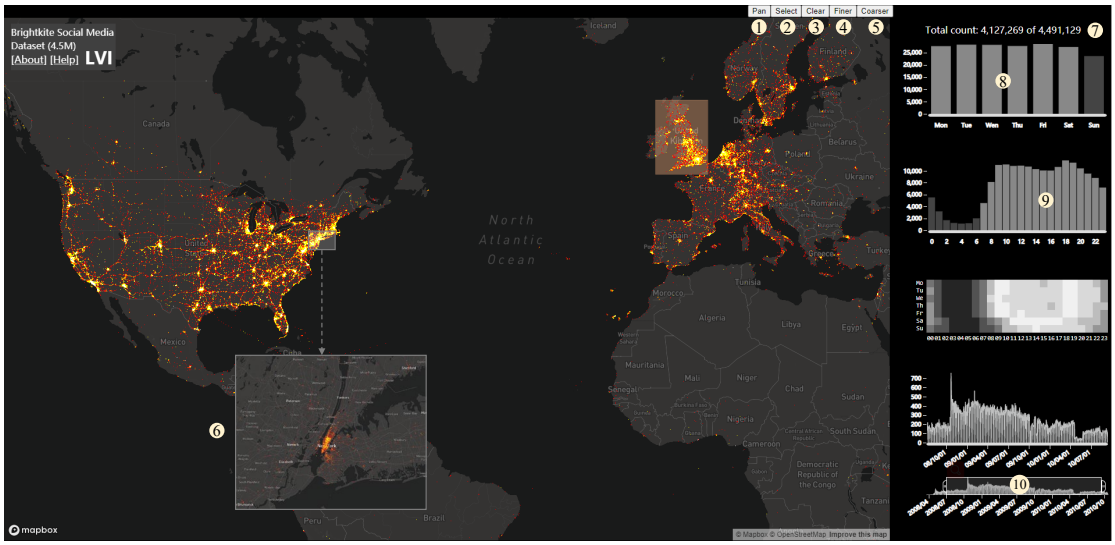
Memory Mode. HPS is stored in the main memory. There is no I/O operation in this way. The selected parts of sorted arrays can be directly fetched from memory and transferred to the GPU for subsequent calculations. All calculations are done on the GPU, ensuring the response speed of the approach. However, copying data between the GPU and the main memory takes up some additional execution time.

GPU Mode. HPS is stored in the video memory of the GPU. This mode has the best performance. All calculations are conducted on GPU, and there is no need to transfer data between the main memory and the GPU. However, this mode will take up GPU video memory, which is more expensive than the main memory.

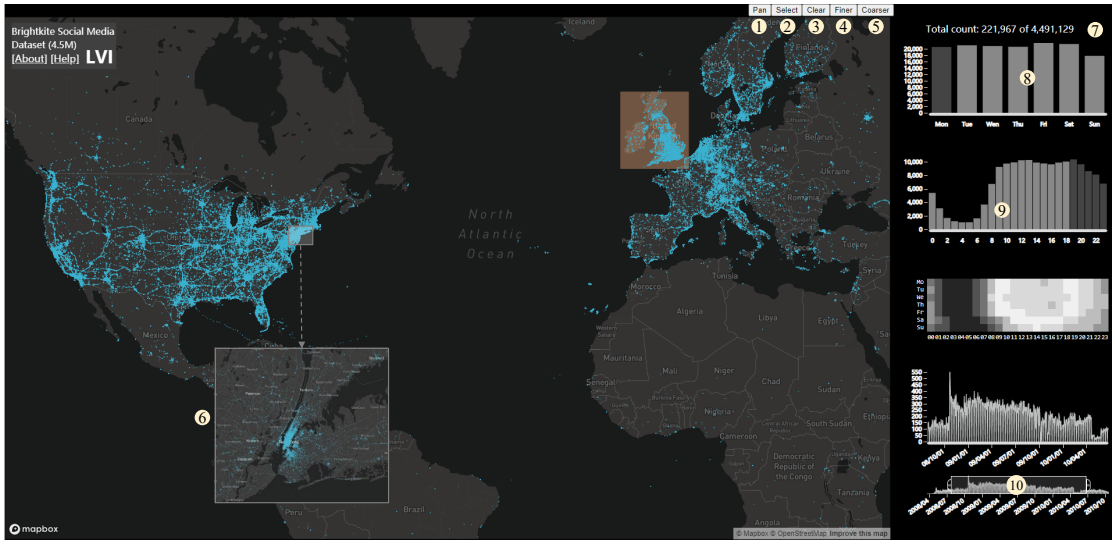
There are other storage configurations. For example, we can separately store HPS in different devices, such as SCF in memory to ensure a high response speed and sorted array in the disk to save space. Moreover, we can only keep parts of sorted arrays with higher access frequency in memory. Users can flexibly configure the two components according to the actual application environment.

5. Interface

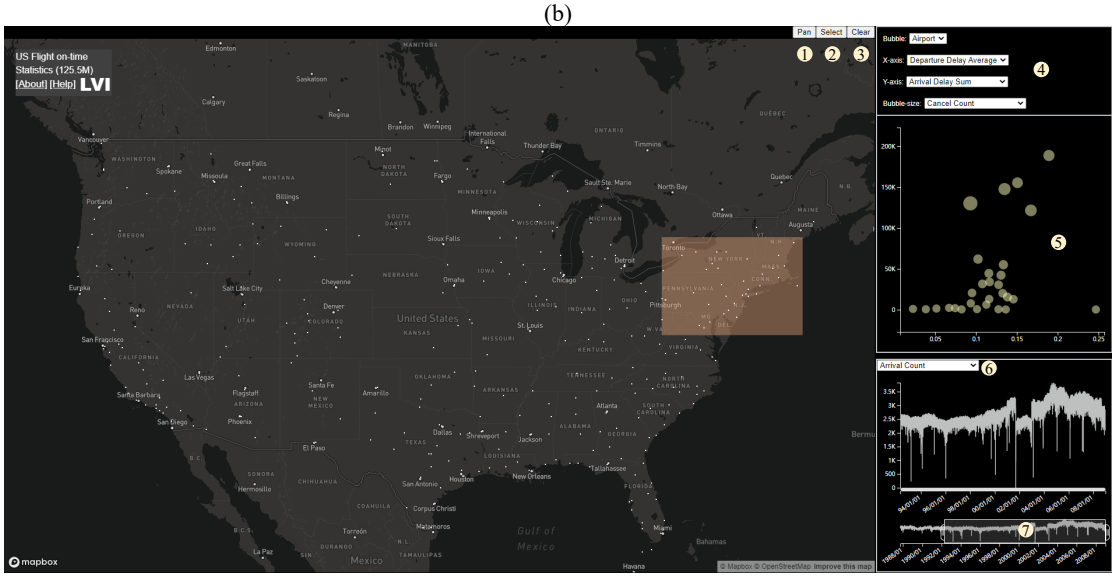
We show the interfaces of three IDE systems.


Brightkite Social Media
Dataset (4.5M)
[About] [Help] LVI
Pan
Select
Clear
Finer
Coarser
Total count: 4,127,269 of 4,491,129
North Atlantic Ocean
mapbox

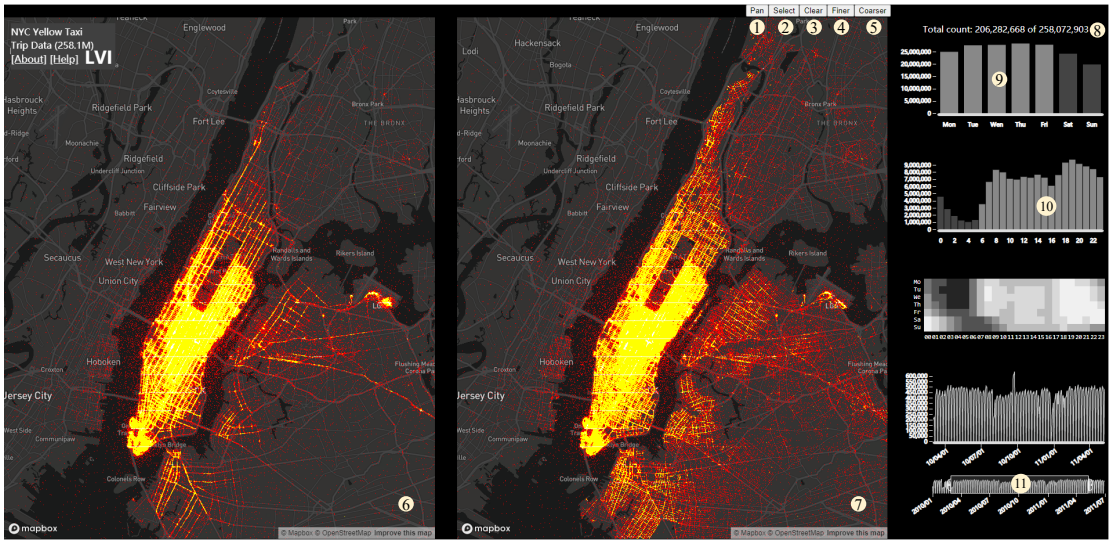
(a)


Brightkite Social Media
Dataset (4.5M)
[About] [Help] LVI
Pan
Select
Clear
Finer
Coarser
Total count: 221,967 of 4,491,129
North Atlantic Ocean
mapbox

(b)


US Flight on-time
Statistics (125.5M)
[About] [Help] LVI
Pan
Select
Clear
Bubble: Airport
X-axis: Departure Delay Average
Y-axis: Arrival Delay Sum
Bubble-size: Cancel Count
Arrival Count
United States
Gulf of Mexico
mapbox

(c)



(d)

Fig. 6. Interfaces of developed IDE systems using HPS. (a) Exploring spatiotemporal patterns of brightkite dataset. The map is showing an aggregate heat-map and can be switched to show filtered records (b). (c) Analyzing on-time statistics of US flights. The scatter-plot supports interactive changes of statistic measures encoded by scatters and axes during the exploration. (d) Analyzing O-D patterns of New York yellow taxis. Longitudes and latitudes are all divided into 2^{25} bins.

Fig. 6(a) shows the interface for exploring spatiotemporal patterns of brightkite dataset. The heat-map on the map is showing aggregate patterns of filtered records. Users can pan the map to any area (Fig. 6(a1)), select or deselect a region (Fig. 6(a2-a3)), adjust display resolution of the heat-map (Fig. 6(a4-a5)), and scroll the map to differet zooming levels (Fig. 6(a6)). The number of tweets filtered by the current operation is shown (Fig. 6(a7)). Users can select time intervals of different scales on multiple views (Fig. 6(a8-a10)). We can switch the map to show filtered records without aggregation, as in Fig. 6(b).

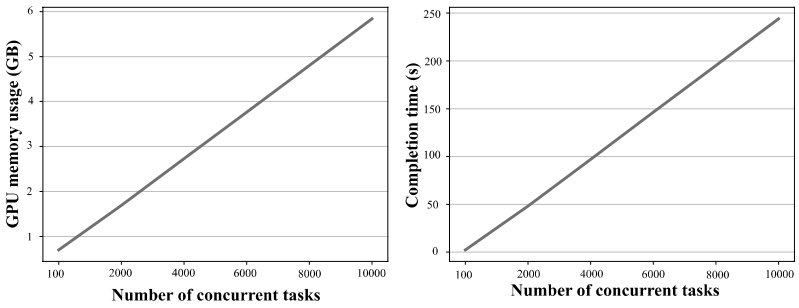
Fig. 6(c) shows the interface for analyzing on-time statistics of US flights. The map supports the same operations as in Fig. 6(a) (compare Fig. 6(a1-a5) and Fig. 6(c1-c3)). The system integrates a scatter-plot (Fig. 6(c4)) that supports interactive changes of the statistic measures mapped by scatters and the two axes. Similarly, users can change the statistic measure of the y-axis of the line-chart, see Fig. 6(c6). When clicking on a scatter, the corresponding object's on-time statistics will show, as Fig. 6 (c5). Users can select a time interval in a line-chart, as in Fig. 6(c7).

Fig. 6(d) shows the interface for analyzing the O-D patterns of New York yellow taxis. Operations of all views are the same as in Fig. 6(a) (see Fig. 6(a1-a10) and Fig. 6(d1-d11)). The two maps are showing pick-up and drop-off patterns, as in Fig. 6(d6) and Fig. 6(d7). Longitudes and latitudes of the two maps are divided into 2^{25} bins to enable high-resolution query/display.

6. Concurrent Performance

We test HPS's concurrence performance by executing a query task, i.e. calculating the intersection

of the two sets and aggregating the intersected cells into 24 groups. We generate two sets by conducting queries using the brightkite system. Each set contains about 200K cells, while the intersection contains about 150K cells. We simultaneously initialize different numbers of tasks and observe the execution time and GPU memory usage.



(a) (b)

Fig. 7. Results of the concurrent performance test. (a) and (b) show the variations of GPU memory usage and completion time as the number of tasks increase.

Fig. 7 shows the experiment results. In general, both memory usage and time have a linearly-increasing trend. Specifically, 10000 tasks will take about 6GBs memory and 250s time overhead, i.e. 60M memory and 20ms for each task. We can further improve the concurrent performance through a series of engineering methods, such as adding more GPUs and a scheduling modular, using a connection pool, etc.

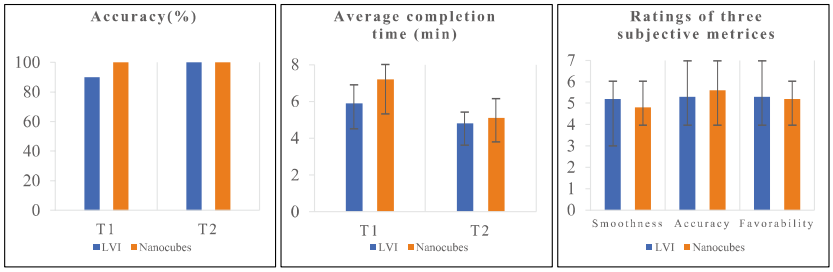
7. User Study

We compare user experiences of systems developed on HPS and Nanocubes [3]. The purpose is to illustrate the performance of HPS in usage scenarios fully. We recruited 20 participants (12 males and 8 females; aged 22-26 years, median: 23), who are graduate students majored in computer science. They have experience in data analysis and website development but are not involved in this project. We divided the participants into two groups (10 participants per group), each conducting the same pre-defined tasks using either our system or Nanocubes'. We choose the Brightkite system for the experiment, as Nanocubes has a system with the same visual design, ensuring the experiment's fairness.

For each trial, we demonstrated the online system to the participants. They then freely used the system and were encouraged to ask any system questions. We provided guidance and make sure they can use the system smoothly.

Each participant should use the system to complete two pre-defined tasks: (1) sort three time periods (0:00-7:00, 8:00-17:00, 19:00-24:00) according to the numbers of tweets posted in New York City during the three periods; (2) sort three places (Manhattan, Queens and Brooklyn) in New York City according to the numbers of tweets posted from19:00 to 24:00 Sunday in the three regions. The two tasks focus on testing spatial and temporal views, respectively. We collect the completion time and accuracy of each group.

After completing the two tasks, each participant filled out a questionnaire, by rating three aspects on 7-point Likert scales. The three aspects are: (1) how smooth the operations are; (2) how clear and correct the shown information is; (3) how much the participant like the system. Participants were also encouraged to leave any comments about the systems on the questionnaires.



(a) (b) (c)

Fig. 8. User study results. (a-b) Accuracy and average completion time of objective tasks. (c) Ratings of three subjective metrics.

Fig. 8(a) and (b) show the accuracy and the mean completion time of the two groups (we measure the accuracy according to the objective ground truth extracted from the original data). We find only one significant effect using ANOVA, i.e., the first task's average completion time ($p<0.05$, $\eta 2= 0.353$). This explains that the participants using our system can complete the task faster. We assume the network latency causes a significant effect. Test results of other tasks are not significantly different. In general, the results illustrate that systems developed with HPS can achieve similar usability to those of Nanocubes.

ANOVA test results of the three subjective metrics show consistent trends with the objective tasks. The differences in all the three metrics between the two groups are tiny, and we do not find any significant effect. This further illustrates the similar usability of HPS to Nanocubes.

We also identify several places to be improved from the comments of questionnaires. Two participants pointed out that the heat-map temporarily disappears when zooming the map to a new level. This is because we use open-source online map control. We plan to develop a light-weight map control to resolve this problem. Another participant found our systems to only run in Chrome browser. We explained the map control relies on WebGL. On a browser without WebGL, the user can only see the four temporal views. This problem can also be solved by developing a new map library.

REFERENCES:

[1] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. M. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of ACM SIGPLAN*, 2008, pp. 73-82.

[2] C. Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *Proceedings of ACM SIGMOD*, 1998, pp. 355-366.

[3] L. Lins, J. T. Klosowski, and C. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *IEEE Transactions on Visualization and Computer Graphics*, 2013.