Nanocubes for Real-Time Exploration of Spatiotemporal Datasets

Lauro Lins, James T. Klosowski, and Carlos Scheidegger

Fig. 1. Example visualizations of 210 million public geolocated Twitter posts over the course of a year. The data structure we propose enables real-time (these images above were rendered faster than the typical screen refresh rate) visual exploration of large, spatiotemporal, multidimensional datasets. The visual encodings built using nanocubes are within a controllable difference to ones rendered by a traditional linear scan over the dataset. They naturally support linked navigation and brushing, and include choropleth maps, time series over arbitrary regions and scales of space and time, parallel sets, histograms, and binned scatterplots. The color scale of the choropleth map is a diverging scale in which blue corresponds to iPhones being relatively more popular, and red corresponds to higher relative popularity of Android devices.

Abstract—Consider real-time exploration of large multidimensional spatiotemporal datasets with billions of entries, each defined by a location, a time, and other attributes. Are certain attributes correlated spatially or temporally? Are there trends or outliers in the data? Answering these questions requires aggregation over arbitrary regions of the domain and attributes of the data. Many relational databases implement the well-known data cube aggregation operation, which in a sense precomputes every possible aggregate query over the database. Data cubes are sometimes assumed to take a prohibitively large amount of space, and to consequently require disk storage. In contrast, we show how to construct a data cube that fits in a modern laptop’s main memory, even for billions of entries; we call this data structure a nanocube. We present algorithms to compute and query a nanocube, and show how it can be used to generate well-known visual encodings such as heatmaps, histograms, and parallel coordinate plots. When compared to exact visualizations created by scanning an entire dataset, nanocube plots have bounded screen error across a variety of scales, thanks to a hierarchical structure in space and time. We demonstrate the effectiveness of our technique on a variety of real-world datasets, and present memory, timing, and network bandwidth measurements. We find that the timings for the queries in our examples are dominated by network and user-interaction latencies.

Index Terms—Data cube, Data structures, Interactive exploration

1 INTRODUCTION

As datasets get larger, exploratory data visualization becomes more difficult. Consider a dataset with a billion entries. We can compute a small summary of the dataset and visualize the summary instead of the dataset, but as Anscombe’s famous quartet shows [3], summaries themselves cannot ascertain their own validity. Summaries might help, but in order to understand if that is the case, we will inevitably find ourselves having to visualize one billion residuals. As far as scale goes, we are back to square one. In other words, data summarization alone will never solve the problem of scale in exploratory visualization. As visualization practitioners, what then can we do? Even drawing the simplest scatterplot is not straightforward. If we decide to produce the visualization by scanning the rows of a table, we will either need non-trivial parallel rendering algorithms or significant time to produce a drawing. Neither of these solutions is attractive or scales well with dataset size.

Data cubes are structures that perform aggregations across every possible set of dimensions of a table in a database, to support quick exploration [15, 31]. Many visualization systems are built on top of data cubes, concretely or conceptually. Still, only recently have researchers started to examine data cube creation algorithms in the context of information visualization [33, 18, 21].
Section 4, we show how to construct a data cube that fits in the main memory of a modern laptop computer or workstation, extending the work of Sismanis et al. [31]. In addition, the query times to build the visual encodings in which we are interested will be at most proportional to the size of the output, which is bounded by the number of screen pixels (within a small factor). This is an important observation: the time complexity of a visualization algorithm should ideally be bounded the number of pixels it touches on the screen. Our technique enables real-time exploratory visualization on datasets that are large, spatiotemporal, and multidimensional. Because the speed of our data cube structure hinges partly on it being small enough to fit in main memory, we call it a nanocube.

By real-time, we mean query times on average under a millisecond for a single thread running on computers ranging from laptops, to workstations, to server-class computing nodes (Section 6). By large, we mean that the datasets we support have millions to billions of entries.

By spatiotemporal, we mean that nanocubes support queries typical of spatial databases, such as counting events in a spatial region that can be either a rectangle covering most of the world, or a heatmap of activity in downtown San Francisco (Section 4.3.1). By the same token, nanocubes support temporal queries at multiple scales, such as event counts by hour, day, week, or month over a period of years (Section 4.3.3). Data cubes in general enable the Visual Information-Seeking Mantra [29] of “Overview first, zoom and filter, then details-on-demand” by providing summaries and letting users drill down by expanding along the wanted dimensions. Nanocubes also provide overviews, filters, zooming, and details-on-demand inside the spatiotemporal dimensions themselves.

By multidimensional, we mean that besides latitude, longitude, and time, each entry can have additional attributes (see section 6) that can be used in query selections and rollups.

As we will show, nanocubes lend themselves very well to building visual encodings which are fundamental building blocks of interactive visualization systems, such as scatterplots, histograms, parallel coordinate plots, and choropleth maps. In summary, we contribute:

- a novel data structure that improves on the current state of the art data cube technology to enable real-time exploratory visualization of multidimensional, spatiotemporal datasets;
- algorithms to query the nanocube and build linked and brushable visual encodings commonly found in visualization systems; and
- case studies highlighting the strengths and weaknesses of our technique, together with experiments to measure its utilization of space, time, and network bandwidth.

2 RELATED WORK

Relational databases are so widespread and fundamental to the practice of computing that they were a natural target for information visualization almost since the field’s inception [20]. Mackinlay’s Automatic Presentation Tool is the breakthrough result that critically connected the relational structure of the data with the graphical primitives available for display [23] and ultimately lead to data cube visualization tools like Polaris [34, 35] and Show Me [24]. Nanocubes are specifically designed to speed up queries for spatiotemporal data cubes, and could eventually be used as a backend for these types of applications.

In contrast, some of the work in large data visualization involves shipping the computation and data to a cluster of processing nodes. While parallelism is an attractive option for increasing throughput, it does not necessarily help achieve low latency, which is essential for fluid interactions with a visualization tool. As a result, sophisticated techniques such as query prediction become necessary [6]. Leveraging the enormous power of graphics processing units has also become popular [25, 21], but without algorithmic changes, linear scans through the dataset will still be too slow for fluid interaction, even with GPUs.

Another popular way to cope with large datasets is through sampling. Statistical sampling can be performed on the database backend [26, 10, 14], or on the front-end [11]. Still, the techniques we introduce with nanocubes can produce results quickly and exactly (to within screen precision) without requiring approximations, which we believe is preferable. In addition, as Liu et al. argue, sampling by itself is not sufficient to prevent overplotting, and might actually mask important data outliers [21].

Fekete and Plaisant have proposed modifications of traditional visual encodings which use the computer screen more efficiently [13]. These scale better with dataset size, but nevertheless require a traversal of all input data points that renders the proposal less attractive for larger datasets. Carr et al. were among the first to propose techniques replacing a scatterplot with an equivalent density plot [5]; nanocubes enable these visualizations at a variety of dataset sizes and scales.

Careful data aggregation [17], then, appears to be one of the few scalable solutions for low-latency large data graphics. While Elmqvist and Fekete propose variations of visualization techniques that include aggregation as a first-class citizen [12], in this paper we show how to issue queries such that, at the screen resolution in which the application is operating, the result is indistinguishable (or close to) from a complete
3 Data Cubes

Following common practice, we will call the table in Figure 4 a relation, its columns attributes, its lines records, and its entries values. An aggregation represents the idea of selecting a certain group of records from a relation and summarizing this group using an aggregation function (e.g., count, sum, max, min). For example, a possible aggregation for the relation A could be to select all its records and summarize those using count, yielding five as the aggregation result. If we allow a special value All to be a valid attribute value, we could represent this aggregation as relation B in Figure 4. A record that contains the special value All is an aggregation record. Using this notation, it is easy to understand some conventional ways of describing aggregations for a given relation: GROUP BY, CUBE, and ROLLUP.

A GROUP BY operation is one in which a relation is derived from a base relation given a list of attributes and an aggregation function. For example, GROUP BY on attributes Device and Language with the count aggregation function results in the relation C in Figure 4. Note that for every different combination of values present in the attributes of a base relation, an aggregation record is added to the resulting relation. In our running example, these combinations are (Android, en), (iPhone, en), and (iPhone, ru).

The CUBE operation is the result of collecting all possible GROUP BY aggregations into a single relation for a given list of attributes (i.e., \(2^n\) GROUP BYs for \(n\) input attributes). In our running example, the CUBE for count on Device and Language is the union of four GROUP BYs: on (1) no attributes; on (2) Device only; on (3) Language only; and (4) on Device and Language, shown in relation D in Figure 4. Finally, a ROLLUP is a constrained version of the CUBE operation where the order of the input attributes is important. A ROLLUP on Device and Language (in this order) means the union of GROUP BYs on: (1) no attributes; (2) Device; and (3) Device and Lang-
Fig. 5. A simplified set of queries supported by nanocubes. The column $s$ represents space; $t$, time; $c$, category. R means “rollup”, D means “drilldown”. The value next to $R$ or $D$ contains the subset of that dimension’s domain being selected. $U$ represents the entire domain (“universe”).

4 NanoCube: A Compact, Spatiotemporal Data Cube

Data visualizations in a computer are necessarily bounded by display size, and so we would like to be able to quickly collect subsets of the dataset that would end up in the same pixel on the screen. However, spatiotemporal navigation is inherently multiscale. The same data structure should support quick indexing for a visualization over multiple years of time series and for drilling down into one particular hour or day. Similarly, the data cube should support aggregation queries over vast spatial regions covering entire continents, as well as very narrow queries covering only a few city blocks.

The database notion of roll-up, in a sense, aligns nicely with the notion of Level of Detail. For example, if the records of a table (relation) contain a location attribute, one can design a roll-up query whose resulting relation encodes the same information as the one encoded in a level of detail data structure. More concretely, suppose $\ell_1, \ldots, \ell_k$ are attributes computed from the original location attribute and yield “quadtree addresses” of increasingly higher levels of detail (from 1 to $k$). A roll-up query on these (computed) attributes results in, essentially, the same information as the one contained in a quadtree (given that we are keeping the same summary in both, e.g. count).

The second important notion in the design of nanocubes is the idea that we want to combine aggregations of independent dimensions at independent levels of detail. For example we might want to know for a whole country, what is the spatial distribution of tweets generated by an iPhone: coarse on the spatial dimension, but specific on the device dimension. Conversely, we might want to know the distribution of tweets (coarse on device) in a small city block (fine in space). In relational database terminology, this model has a name: it is a CUBE of roll-up, or a roll-up CUBE. With the terminology set, we can state: a nanocube is a data structure to efficiently store and query spatiotemporal roll-up CUBE. Besides implementation tricks, the main difference between nanocubes and previously published sparse coalesced data cubes such as Dwarf cubes [30] is in the design of aggregations across spatiotemporal dimensions (see Sections 4.3.1 and 4.3.3). Next, we present a formal description of the components that make up our nanocube index, pseudo-code for building nanocubes, an illustrated example, and how queries are made against our index.

### 4.1 Definitions

Let $O$ be a set of objects. A labeling function $\ell : O \rightarrow L$ associates a label value to the objects of $O$. We can think of $\ell$ as an attribute in a relational database. In connection with the level of detail discussion above, if $\ell_1$ and $\ell_2$ are two labeling functions for $O$, we say $\ell_1$ is coarser than $\ell_2$ or that $\ell_2$ is finer than $\ell_1$ if for any two objects $o, o’ \in O$ the implication $\ell_2(o) = \ell_2(o’) \Rightarrow \ell_1(o) = \ell_1(o’)$ holds. We denote this fact by $\ell_1 \geq \ell_2$.

A sequence of labeling functions $c = [\ell_1, \ell_2, \ldots, \ell_k]$ for objects $O$ is a chain for $O$ if every labeling function is coarser than the next labeling function in the sequence: $\ell_i \geq \ell_{i+1}$. The number of levels of a chain is defined by $\text{levels}(c) = |c| + 1$. An indexing schema for objects $O$ consists of a sequence of chains $S = [c_1, c_2, \ldots, c_n]$. The dimension of an indexing schema $S$ is the length of its sequence of chains and is denoted by $\text{dim}(S)$. The multiplicity of a schema $S$ is the product of its chains’ number of levels: $\mu(S) = \prod_{i=1}^{\text{dim}(S)} \text{levels}(c_i)$.

A full assignment for a sequence of labeling functions $[\ell_1, \ell_2, \ldots, \ell_k]$ is a sequence of label values $[v_1, v_2, \ldots, v_k]$ where $v_i$ is a label value under $\ell_i$. Any prefix of a full assignment for a sequence of labeling functions, including the empty one, is referred to as a partial assignment. Note that a full assignment is also a partial assignment since a sequence is also a prefix of itself. An address on a schema is a sequence of partial assignments for its chains, more formally, if $S = [c_1, c_2, \ldots, c_n]$ is an indexing schema, then $a = [p_1, p_2, \ldots, p_n]$ is an address of $S$ if $p_i$ is a partial assignment for chain $c_i$. The set of possible addresses of $S$ is denoted by $\text{addr}(S)$.

The addresses of an object $o$ under indexing schema $S$, denoted by $\text{addr}(o, S)$ are all the addresses in $\text{addr}(S)$ whose partial assignments are consistent with the label values associated to $o$ and it is easy to see that the size of $\text{addr}(o, S)$ is always $\mu(S)$. Besides a schema $S$, the definition of a nanocube requires a separate labeling function, $\ell_{\text{time}}: O \rightarrow T$, which we refer to as the time labeling function since we use it to encode the temporal aspect of our datasets. Thus, a nanocube for objects $o_1, \ldots, o_n$ is denoted by:

$$\text{Nanocube}([o_1, \ldots, o_n], S, \ell_{\text{time}})$$

A key in a nanocube is any pair $(a, t)$ where $a \in \text{addr}(S)$ and corresponds to a full assignment (see definition above) and $t \in T$ is a possible time label. If we remove the requirement of $a$ being a full assignment, we say that pair $(a, t)$ is an aggregate key. Note that every key is also an aggregate key. The set of all possible keys and the set of all possible aggregate keys of a nanocube are respectively referred to as its key space, or $K^*$, and its aggregate key space, or $K_{\text{agg}}$. The size of the key space, $|K^*|$, is referred to as its cardinality.

### 4.2 Building the Index

To ease the remaining exposition, we assume that a nanocube maps an aggregate key to a count. Nevertheless, nanocubes support any kind of summary that is an algebra with weighted sums and subtractions. Notably, this includes linear combinations of moment statistics, with which we can compute means, variances and covariances.

The pseudo-code for building a nanocube is presented in Figure 3. The main idea of the algorithm is for every object $o_i$ to first find the finest address of the schema $S$ hit by this object, update the time series associated with this address and from there on update in a deepest first fashion, all coarser addresses also hit by $o_i$. Note that the content of the last dimension of schema $S$ is always a time series and that is why, in line 21 of ADD, we insert the time label of the current object. The important trick used is to, when possible, allow for shared links.
We next cross the dimension boundary line by traversing the (blue) edge pointing to the next dimension. We first illustrate how simple queries are conducted on nanocubes using a sum example. Recall that the end result of the query will be to return a count of elements inside the leaf (2) is the answer for the query. Note that, for each dimension, a simple query only traverses a single path of its tree before jumping to the root node of a tree in the next dimension (or to a leaf node which encodes time and is treated differently). In general, higher level queries might traverse multiple paths of a single tree, and may also report single aggregates, multiple aggregates, or even combine aggregates from multiple branches. To abstract and classify how a general nanocube query processes a dimension, we use the terminology of rollups and drilldowns (the ROLL_UP relational operation is related but has a different meaning than the one we intend here). The dimension that is the basis of a rollup should report a single aggregate value as a result. This aggregate might be a single existing aggregate in the nanocube or a combination of multiple aggregates from different branches of that dimension. A drilldown reports aggregate values for multiple branches in that dimension. In a single nanocube query, each dimension is independently set to be used as the basis for either a rollup or a drilldown. In Figure 5, we provide a set of example queries and their mapping to the server query URL (see Section 5).

It is worth noting that the order of the $d$ dimensions does not impact the worst-case query run-time. For example, a marginal barchart of a categorical dimension (with $k$ bars), requires $O(kd)$ time, regardless of the category chosen or the ordering of the dimensions.

### 4.3 Querying the Cube

Nanocubes support three distinct dimension types, which are always traversed in a fixed order: spatial, categorical, and finally temporal. Before describing queries for each of these specific dimension types, we first illustrate how simple queries are conducted on nanocubes using an example. Recall that the end result of the query will be to return precomputed aggregates across one or more dimensions.

In Figure 2(5), assume we are interested in the count of all tweets that occurred in the northwest quadrant of the world, regardless of the device type and time. The aggregate key $k_w = ((p_1, p_2), t)$ for this query consists of: (1) the partial assignment for the northwest quadrant in the spatial dimension: $p_1 = [0, 1]$; (2) the empty partial path for the device dimension $p_2 = [],$ indicating any device; and (3) a time label $t$ indicating any time. Finding the precomputed aggregate for a given aggregate key is called a simple query. In this example, we start at the top-most node and traverse all black parent-child links described in the partial assignment $p_1$; in this case only the black $[0, 1]$ link. We next cross the dimension boundary line by traversing the (blue) content link of the current node. The traversal process is repeated for the device dimension using the partial assignment $p_2$. In this specific case, no restrictions are made on the device, and we can jump to the next dimension by traversing the content link. At this point, we reach a leaf node containing $\{o_1, o_2\}$. Since no time constraint is imposed, the count of elements inside the leaf (2) is the answer for the query.

Note that, for each dimension, a simple query only traverses a single path of its tree before jumping to the root node of a tree in the next dimension (or to a leaf node which encodes time and is treated differently). In general, higher level queries might traverse multiple paths of a single tree, and may also report single aggregates, multiple aggregates, or even combine aggregates from multiple branches. To abstract and classify how a general nanocube query processes a dimension, we use the terminology of rollups and drilldowns (the ROLL_UP relational operation is related but has a different meaning than the one we intend here). The dimension that is the basis of a rollup should report a single aggregate value as a result. This aggregate might be a single existing aggregate in the nanocube or a combination of multiple aggregates from different branches of that dimension. A drilldown reports aggregate values for multiple branches in that dimension. In a single nanocube query, each dimension is independently set to be used as the basis for either a rollup or a drilldown. In Figure 5, we provide a set of example queries and their mapping to the server query URL (see Section 5).

It is worth noting that the order of the $d$ dimensions does not impact the worst-case query run-time. For example, a marginal barchart of a categorical dimension (with $k$ bars), requires $O(kd)$ time, regardless of the category chosen or the ordering of the dimensions.

### 4.3.1 Spatial Queries

In our current implementation, the first dimension to be traversed in a nanocube is always the spatial dimension. It is helpful to think of this dimension as being represented by a traditional quadtree [28], where each quadtree node is enriched by an extra pointer (content pointer) that jumps to the next dimension of the nanocube. If a query matches exactly the region represented by a quadtree node, then the content pointer of that node is the gateway for all aggregates that refers precisely to that region. If the query includes categorical restrictions (or drilldowns), then these can be found by traversing down the following categorical dimensions, as described below. However, spatial regions will very rarely match exactly one node in the quadtree; therefore, we use the traditional region quadtree intersecting algorithms to compute the minimal disjoint set of quadtree nodes that exactly cover the query region [28], and sum the resulting rollups across the nodes.

Arbitrarily shaped regions are not currently supported for spatial queries because of the additional complexity that is introduced, but there is no intrinsic barrier in the framework which prevents them from working. For spatial rollups, we support arbitrary rectangular regions. For drilldowns, we currently support regions defined by the tiling scheme of most mapping services on the WWW. For example, the widest tile in the world in OpenStreetMap [16] has coordinates...
With this data structure, we can compute a temporal rollup of event counts in the period. A temporal drilldown happens similarly, and we can compute a time series with $t$ entries by performing $t + 1$ binary searches. Each determines the breaking points in the cumulative array, and the final value is computed by stepwise differences.

This scheme for storing time entries is attractive for several reasons. First, it ensures that we can store time series of any granularity without requiring a nested tree structure like our spatial indexing scheme. Second, the running time is essentially optimal (up to a log $n$ factor), and the algorithm is extremely fast in practice.

## 5 Implementation

We use a client-server architecture for the current implementation of nanocubes. The server reads the multidimensional data, builds a nanocube, and then processes queries on the nanocube from client applications. The server is a C++11 template-based implementation which makes it easy to plug in different data structures for each dimension of the nanocube. For example, for the Twitter data, we use a 2d quadtree indexed on location for our spatial dimensions (latitude and longitude), and flat trees for each categorical dimension (e.g., language, device, application), and our summed-area table variant for the time dimension.

The nanocube construction algorithm has not been optimized for speed (results are included in section 6) but there are several possible improvements that we could make: using multiple threads, or using memory pools to avoid the overhead of repeated memory allocations and deallocations. Due to the scale of the input data, most of our effort has been spent on optimizing memory usage, including optimized libraries for memory allocation (libtcmalloc) and tagged pointers, which allow us to use the 16 most significant bits in a 64-bit pointer to quickly identify different types of nodes in our data structure.

The nanocube server exposes its API for queries via HTTP. More specifically, it provides a web service through which queries can be issued [27]. After the data cube is built, the data structures are no longer mutated, and so the server is easily parallelizable (it also means that nanocubes are add-only: they cannot be updated if a record is removed from the base relation). Our implementation uses the Monogoose library for handling multiple HTTP requests in separate threads concurrently [22]. We have built two front-end visualization clients to query the nanocube server. One client is written in C++ and uses OpenGL for efficient rendering. The other client is browser-based and is written in Javascript, HTML5, SVG, WebGL, and D3 [4].

## 6 Experiments

To study the behavior of nanocubes, we collected six datasets that ranged in size from four million records up to over one billion records. Each dataset includes geospatial, temporal, and domain-specific categorical dimensions with up to 30 distinct values. For all but the
Fig. 10. Highlights of a visual analysis session of the CDR dataset, with 1,043,884,027 records. We noticed the different patterns in call volume by interacting with the dataset and trying different regions and category selections. Notice the patterns occur at different spatial and temporal scales.

In the following sections, we provide a brief overview of each of the datasets, followed by an overall summary of our experimental results in section 6.8. For each of the experiments, we paid particular attention to how much memory was required to build and store the nanocube index, as well as the overall complexity of the dataset itself, which varied greatly from one to the next. Once the nanocubes were constructed, we queried them using one or both of our front-end clients to highlight the ease with which analysts could explore the data.

The query times and bandwidth usage across all experiments are consistent, so we report them in aggregate here. The mean query time was 800µs (less than 1 millisecond) with a maximum of 12 milliseconds. The output size per query averaged 5KB, with a maximum size of 50KB (geographical tiles dominated bandwidth usage). Our server currently uses no compression, although we plan to support transparent gzip stream encoding. The mean number of queries for the C++ client was 100 requests per second. The HTML5 client is much quieter, at around 1 query per second, since linked views are only updated when a brush is released. The C++ client was designed for LANs, and its bandwidth usage is around 5Mbps, well within current capacities.

6.1 Twitter

Between November 2011 and June 2012, we collected about 210 million tweets that originated in the United States using Twitter’s public feed which provides a representative sampling of all tweets. The rate of tweets obtained averaged about one million per day. The data was streamed in the form of JSON objects, from which we extracted the following attributes: latitude and longitude of the device, the time the tweet occurred, the client application used, the type of device, and the language of the tweet. The categorical dimensions in our data (application, device, language) had respectively 4, 5, and 15 distinct values. With a nanocube built using this data, we could quickly explore the data to better understand the areas in which one device is more popular than another, where each of the languages is most prevalent, and how that information changes over time (see Figure 7).

6.2 Airline Commercial Flights History

This publicly available dataset contains data for every commercial flight in the United States over a 20 year period (1987-2008) [2, 36]. For over 120 million flights, the records include the scheduled departure and arrival times, the actual departure and arrival times, the origin and destination airports, the airline, and other fields. For this experiment, we built our index using the origin airport (for latitude and longitude), scheduled departure time, the departure delay, and the airline. This allows us to answer queries related to overall departure delays for any airports, airlines, time of day, or combinations thereof. In Figure 8 we present an overview of the weekly percentages of total commercial flights in the U.S. for a 20 year period of Delta and American Airlines.
Fig. 11. Selecting different geographical regions highlights how different populations interacted with the Brightkite social network. While in the US and UK there is no substantial difference between weekday and weekend traffic, in Japan weekday usage is markedly lower.

6.3 Call Detail Records

For each cellular phone call, telecommunications companies collect information about the call including time, duration, and the sequence of cell towers that carried the call. This information is organized into what are known as Call Detail Records (CDRs). A large U.S. service provider (privately) shared with us over one billion CDRs generated from a one month period in July 2010. Due to the sensitivity of CDR data, our data has been completely anonymized. No personally identifiable information was gathered or used in conducting this study. To the extent that any data was used, it was anonymous and aggregated data. The nanocube was built using the geospatial temporal data (of first cell tower), as well as the duration (transformed to a categorical dimension) of each call (see Figure 10).

6.4 Location-Based Social Networks

The next dataset is also publicly available, and consists of location-based checkins in the Brightkite social network collected by Cho et al. [7]. The dataset comprises all data checkins from the (now-defunct) website between April 2008 and October 2010. In addition to latitude, longitude, and the time of each checkin, we redundantly encoded hour of day and day of week as extra categorical dimensions, since we expected there to be interesting periodic day-to-day and weekday vs. weekend patterns (see Figures 11 and 12).

6.5 Customer tickets

This dataset contains a record of about 8 million records of customer interactions of a large U.S. service provider over a period of 2.5 years. The dataset contains latitude, longitude, time and report type (one of eight categories). The same measures taken to anonymize CDR data in Section 6.3 were used here. In Figure 9, we highlight the use of nanocubes to detect relative changes in category in the time series plot, and how choropleth maps restricted to different time regions show the change in geographical distribution of the report types.

6.6 SPLOM

This is a collection of synthetic datasets (each with five dimensions) designed by Kandel et al. [18] to exercise data cube technology (SPLOM stands for ScatterPLOT Matrix, the visual encoding used to explore the dataset in that work), also used by Liu et al. [21]. To compare resource usage to that of these other proposals, we built nanocubes using five different bin sizes per dimension, from 10 to 50.

6.7 Memory Usage

To understand the memory requirements to build a nanocube, it is important to remember that objects are not inserted directly into the nanocube, but rather through their corresponding keys (see Figure 3). An object’s key identifies the most specific bin in the nanocube that contains that object. Thus, depending upon the resolutions defined for the dimensions of a nanocube, two different objects may or may not be distinguishable. For example, if the time resolution of a nanocube is one hour, two objects with timestamps at 20h10m and 20h50m will both have keys with the same time label rounded to 20h. As a result, new occurrences of keys that were already inserted into a nanocube do not require additional storage space.

Figure 14(A) shows the memory usage growth for the SPLOM dataset as we insert from zero to one billion objects into the five nanocubes of increasing bin size. In all cases there is an initial rapid growth that quickly flattens out. In the case of SPLOM 50, the index grew from 0 to 300MB with the first 200 million object insertions, but grew less than 100MB larger as a result of the next 800 million object insertions. The explanation for this behavior is that, by a characteristic of the synthetic object generator (samples from a normal distribution for each dimension) a key set of high probability was quickly generated making it harder and harder for a new object with an unseen key to be generated. Thus, later in the process, most inserted objects will not require more memory since their keys were already inserted into the nanocube. We refer to this phenomena as key saturation.

In Figure 14(B), we present curves for memory usage and number of keys for the CDR dataset, both relative to the final nanocube numbers. To test for a key saturation effect, we excluded the time dimension present in the original data. Once again, we observe an initial rapid growth on memory usage explained by the large number of combinations of cell locations and call durations not yet inserted. Once the bulk of the keys corresponding to these combinations are inserted, a relatively small but steady rate of new keys are inserted reflecting a small but steady growth in the cell tower infrastructure. Similarly defined curves for the Flights dataset are shown in Figure 14(C). The first part of this experiment follows the same trend as before: rapid initial growth, followed by a saturation of keys and a steady but much slower growth reflecting the small rate of new airport locations and carriers. At about 80M inserted flights (circa 1995), we again observe a regime of rapid growth, which corresponds to a burst of new carriers.

6.8 Performance Summary

In Figure 13, we summarize the relevant information for building our nanocubes on the previously described datasets. The number of input objects N, the memory requirements, and the build times are reported in the first three columns, while the exact schema used for each dataset...
### Table 13. Summary of resource usage for our reported experimental results (K=10^3, M=10^6, B=10^9). The numbers in parentheses on the schema column denote the number of bits necessary to refer to a value of that dimension, and their sum is the exponent of 2 in the [K^*] column.

<table>
<thead>
<tr>
<th>dataset</th>
<th>objects (N)</th>
<th>memory</th>
<th>time</th>
<th>size</th>
<th>sharing</th>
<th>keys ([K])</th>
<th>[K^*]</th>
<th>schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>brightkite</td>
<td>4.5 M</td>
<td>1.6 GB</td>
<td>3.30 M</td>
<td>149.0 M</td>
<td>3.00x</td>
<td>3.5 M</td>
<td>2^41</td>
<td>lat(25), lon(25), time(16), weekday(3), hour(5)</td>
</tr>
<tr>
<td>customer tax</td>
<td>7.8 M</td>
<td>2.5 GB</td>
<td>8.47 M</td>
<td>213.0 M</td>
<td>2.93x</td>
<td>7.8 M</td>
<td>2^41</td>
<td>lat(25), lon(25), time(16), type(3)</td>
</tr>
<tr>
<td>flights</td>
<td>121.0 M</td>
<td>3.3 GB</td>
<td>31.13 M</td>
<td>234.0 M</td>
<td>16.50x</td>
<td>43.3 M</td>
<td>2^55</td>
<td>lat(25), lon(25), time(16), carrier(5), delay(4)</td>
</tr>
<tr>
<td>twitter-small</td>
<td>210.0 M</td>
<td>10.2 GB</td>
<td>1.23 h</td>
<td>1.2 B</td>
<td>3.72x</td>
<td>116.0 M</td>
<td>2^35</td>
<td>lat(17), lon(17), time(16), device(3)</td>
</tr>
<tr>
<td>twitter</td>
<td>210.0 M</td>
<td>44.6 GB</td>
<td>5.87 h</td>
<td>5.2 B</td>
<td>4.00x</td>
<td>136.0 M</td>
<td>2^50</td>
<td>d1(4), d2(4), d3(4), d4(4), d5(4)</td>
</tr>
<tr>
<td>splom-10</td>
<td>1.0 B</td>
<td>4.3 MB</td>
<td>4.13 h</td>
<td>51.2 K</td>
<td>5.67x</td>
<td>7.4 K</td>
<td>2^20</td>
<td>d1(6), d2(6), d3(6), d4(6), d5(6)</td>
</tr>
<tr>
<td>splom-50</td>
<td>1.0 B</td>
<td>166.0 MB</td>
<td>4.72 h</td>
<td>8.8 M</td>
<td>16.00x</td>
<td>1.9 M</td>
<td>2^10</td>
<td>lat(25), lon(25), time(16), duration(3)</td>
</tr>
<tr>
<td>cdrs</td>
<td>1.0 B</td>
<td>3.6 GB</td>
<td>3.08 h</td>
<td>271.0 M</td>
<td>18.60x</td>
<td>96.3 M</td>
<td>2^29</td>
<td>lat(25), lon(25), time(16), duration(3)</td>
</tr>
</tbody>
</table>

Fig. 13. Summary of resource usage for our reported experimental results (K=10^3, M=10^6, B=10^9). The numbers in parentheses on the schema column denote the number of bits necessary to refer to a value of that dimension, and their sum is the exponent of 2 in the [K^*] column.

is included in the last column. Column “size” indicates the number of nodes in our data structure (in the nanocube of Figure 2(5) this number would be 41: 22 circles + 19 entries in the leaves). The “sharing” column indicates how much larger the nanocube would be without the sharing mechanism (dashed lines in Figure 2). For example, the twitter dataset nanocube would use at least 4 × 46.4 GB = 185.6 GB without sharing. Column “keys” is the number of distinct keys induced by the N objects (note here that the time dimension is included). Finally, column [K^*] reports the cardinality of the key space of each nanocube. All but two of the datasets fit in 4GB of RAM, and only one of them would not fit in 16GB, the amount of memory in a high-end laptop. The multiscale nature of spatiotemporal datasets make the cardinality of the key space impractically large for any dense storage scheme.

### 7 Discussion

As a sparse scheme to store aggregates, nanocubes suffer from the same drawbacks of any sparse data structure. Namely, when the occupancy (i.e. key space covered by the inserted objects) is large, the extra logic and memory needed to handle pointers is wasteful. A dense mechanism using implicit addressing on arrays has simpler logic, faster access, and uses less memory. When considering multidimensional datasets, however, the memory requirements of a dense mechanism quickly become impractical (see cardinality of Figure 13). Except for the SPLOM experiments, every other dataset would require at least 2^55 memory locations to represent the finest bin summaries (e.g. counts), without even considering the memory needed to represent aggregates. This requirement is simply overwhelming. A sparse scheme like a nanocube’s can handle those datasets with present day technology. Nanocubes will not avoid the combinatorial explosion that happens when an arbitrary number of dimensions is considered, but they still push the boundary of interactive visualizations as far as scale is concerned.

It is enlightening to compare nanocubes to recent data cube visualization proposals: Datavore [18] and imMens [21] (see Figure 15). For this discussion, we assume input objects inducing keys K and aggregate keys K_a. Datavore’s algorithms behave well in the sparse regime, but cannot handle very large datasets, because its querying time appears to be proportional to |K|, while imMens, on the other hand, has extremely fast query times (below 1ms per query, and apparently O(1)), but is designed for the dense regime, and uses memory proportional to the cardinality of its key space, O(|K^*|). This limits the size of the key space and we observe that although imMens reports experiments varying N from 10^6 to 10^9, the value of |K^*| in all experiments were within a factor of 5 of one another [21]. The hierarchical nature of a nanocube’s spatiotemporal index provides advantages in the fidelity of resulting visualizations for a much larger set of scales than imMens (Datavore supports exact visual encodings across any dimensions, but cannot cope with large-scale datasets). This same hierarchical nature provides nanocubes with natural offscreen culling: the region visible onscreen can be interpreted as a spatiotemporal selection, reducing the total processing necessary.

### 8 Limitations and Future Work

Nanocubes offer efficient storage and querying of large, multidimensional, spatiotemporal datasets, but are not without limits. Nanocubes do not allow queries down to any individual record, like a traditional database. Our index was designed specifically to answer queries from interactive visualization systems that explore massive datasets.

Our current server API encourages much chattier communication than is necessary, peaking at hundreds of HTTP requests a second. This happens when brushes are being moved in the C++ client, but could be avoided by techniques like query queuing and prediction [6].

The current nanocube implementation allows for only one spatial dimension and one temporal dimension. It would be clearly useful to allow schemata that included multiple spatial dimensions so that one could visualize, for example, the distribution of geographical locations of flights leaving a certain different geographical area. Similarly, phone calls have two natural geographical dimensions.

Nanocubes still take more memory than we would like. One of our examples clearly demonstrates this: when indexing all six dimensions, the 210 million points from Twitter take around 45GB of memory. This is enough memory for a present-day server, but above that of typical laptops and workstations. We envision dynamic control over the cardinality of dimensions, but leave that for future work. We would also like to explore hybrid solutions that utilize both on-disk and in-memory data structures to enable more complex nanocubes.

Acknowledgments We would like to thank Stephen North, Luciano Barbosa, Divesh Srivastava and Jeffrey Heer for their gracious support and input, as well as the anonymous reviewers.

### References


---

Fig. 14. (A) Nanocube memory usage growth with number of elements, using the SPLOM benchmark by Kandel et al [18]. Notice the plateauing in memory usage due to key saturation. On the right, the growth of memory usage and number of keys when inserting objects into nanocubes for the Call Detail Records (B) and Flights (C) datasets.
Key space, $K$, is a set that grows quickly with the number of dimensions. Function $f$ reflects nanocube’s sharing mechanism and has an important compression effect on the already sparse set $K_s$ (see sharing col. in Figure 13): $|f(K_s)| \leq |K_s|$.

Fig. 15. Comparison of observed asymptotic resource usage of recent methods. Set $K$ corresponds to the input keys and set $K_s$ to the aggregate keys induced by $K$. Key space, $K^*$, is a set that grows quickly with resolution and number of dimensions. Function $f$ reflects nanocube’s sharing mechanism and has an important compression effect on the already sparse set $K_s$.