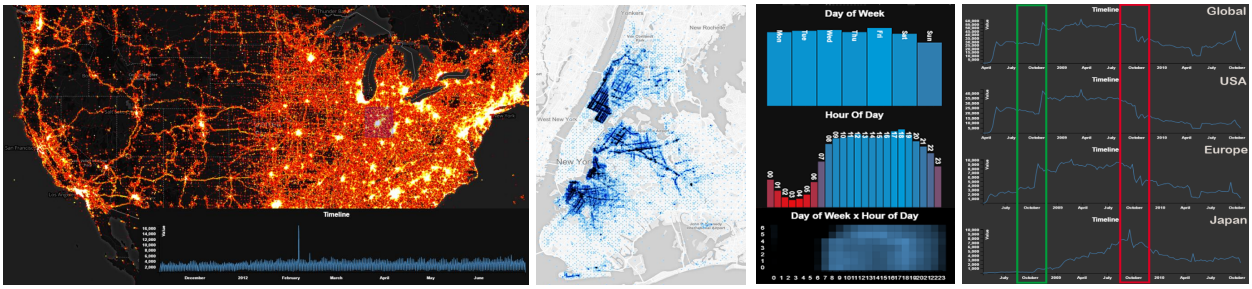
Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data

Cícero A. L. Pahins, Sean A. Stephens, Carlos Scheidegger, João L. D. Comba



Overview of USA tweets between Nov 2011 and Jun 2012 | NYC Green Taxis pick-up | Brightkite in Europe | Brightkite temporal series

Fig. 1. Hashedcubes accelerates queries used in a wide range of interactive exploratory visualizations, such as heatmaps, time series plots, histograms and binned scatterplots, and supports brushing and linking across spatial, categorical and temporal dimensions. In this figure, we show some example visualizations backed by Hashedcubes. The left image shows 210.6 million tweets from November 2011 to June 2012, highlighting the activity during Superbowl XLVI. The central image shows 24.5 million pick-up locations of NYC green taxis rides from January 2014 to June 2015. On the right, the visualizations show different aspects of 4.5 million Brightkite check-ins, a social network. Hashedcubes balances low memory usage, fast running times, and simple implementation; it allows interactive exploration of datasets that previously either required a prohibitive amount of memory or uncomfortably large latencies.

Abstract—We propose Hashedcubes, a data structure that enables real-time visual exploration of large datasets that improves the state of the art by virtue of its low memory requirements, low query latencies, and implementation simplicity. In some instances, Hashedcubes notably requires two orders of magnitude less space than recent data cube visualization proposals. In this paper, we describe the algorithms to build and query Hashedcubes, and how it can drive well-known interactive visualizations such as binned scatterplots, linked histograms and heatmaps. We report memory usage, build time and query latencies for a variety of synthetic and real-world datasets, and find that although sometimes Hashedcubes offers slightly slower querying times to the state of the art, the typical query is answered fast enough to easily sustain a interaction. In datasets with hundreds of millions of elements, only about 2% of the queries take longer than 40ms. Finally, we discuss the limitations of data structure, potential spacetime tradeoffs, and future research directions.

Index Terms—Scalability, data cube, multidimensional data, interactive exploration.

1 Introduction

Designers of interactive visualization systems face serious challenges in the presence of large, multidimensional datasets. On one side, naive implementations of repeated linear scans of the dataset of interest no longer offer acceptable latencies: this makes simple data structures no longer attractive. On the other side, sophisticated implementations of precomputed indices built specifically for visualization have been proposed recently. These offer attractive query times, but their implementations are not trivial to integrate with existing systems, require GPU support, or have another similar downside. This paper provides an affirmative answer to the following question: is there a simple data structure that offers much of the performance of the more sophisticated indices, while maintaining a relatively-low memory footprint and implementation simplicity?

Specifically, we present *Hashedcubes*, a novel data structure that enables fast querying for interactive visualizations of large, multidimensional, spatiotemporal datasets. Hashedcubes supports spatial queries, such as counting events in a particular spatial region; categorical queries over subsets of attribute values; and temporal queries over intervals of any granularity. As we report on Section 6, a typical query is returned in under 30 milliseconds in single-threaded execution. As a practical matter, Hashedcubes was designed to target the amount of main memory of a modern desktop or laptop personal computer (on the order of 16 to 32GB of main memory). In summary, this paper contributes:

- a simple data structure for real-time exploratory visualization of large multidimensional, spatiotemporal datasets, advancing the state of the art especially with respect to implementation simplicity and memory usage,
- an experimental validation of a prototype implementation of Hashedcubes, including a suite of experiments to assess query time, memory usage, and build time of the data structure on synthetic and real-world datasets, and
- an extended discussion of the trade-offs enabled by Hashedcubes, including limitations and open research questions.

2 Related Work

In this section we will focus on work directly related to interactive visual analysis of big data. For a more comprehensive list of papers, we refer the reader to the surveys on big data analysis [15], big data visualization [3], geospatial big data analysis [31] and challenges in big data implementation [18, 36, 22].

The need for low latency in large databases is a popular theme in the literature [5, 40, 10]. *BlinkDB* [2] builds a carefully-constructed

- *Cícero A. L. Pahins and João L. D. Comba are with the Federal University of Rio Grande do Sul. E-mail: {calpahins,comba}@inf.ufrgs.br.*
- *Sean A. Stephens and Carlos Scheidegger are with the University of Arizona. E-mail: {seanastephens,cscheid}@email.arizona.edu.*

stratified sample of the dataset, which allows interactive latencies in approximate queries over multiple terabytes of data. In essence, *BlinkDB* provides infrastructure such that Hellerstein et al.'s online aggregation has fast convergence properties [20]. *ScalarR* improves performance by manipulating physical query plans and computing a dynamic reduction of query sets based on screen resolution [7]; it is an early, central example of explicitly taking peculiarities of a visualization setup in a DB account. *3W* is a search framework for geo-temporal stamped documents that allows fast searches over spatial and text dimensions [37]. *Forecache* [6] improves performance by predicting user actions ahead of the actual queries being issued.

The seminal paper of Gray et al. [17] introduced the *data cube* concept, which laid the foundation for many other methods [34, 32, 11], including our proposal. A data cube can be seen as a hierarchical aggregation of all data dimensions in an n-dimensional lattice. Its main disadvantage is its memory consumption, which becomes impractical as the number of dimensions increases. To address this problem, some approaches describe ways to compress data cubes, such as Dwarf [41], or build on distributed databases to cope with scale requirements [27].

VisReduce [23] is an approach to data aggregation which computes visualization results in a distributed fashion. It uses a modified *MapReduce* [13] algorithm and data compression. Its main drawback is that interaction operations require on-demand aggregations. Thus, the final result is obtained only after the costly transfer over the network of partial and final aggregations. As a rule of thumb, on-demand computation is problematic for visual analysis because of latency. As Liu and Heer describe [33], latencies of as little as half a second can affect the overall quality of an analyst's data exploration process. A popular alternative to hide latency is to use sampling, and report uncertainty estimates as soon as they are available [14]. Similarly, Stolper et al. describe a general framework for progressive approach for visual analytics [42].

The most recent trend in research at the intersection of data management and visualizations is the explicit acknowledgement of the human perceptual system. Wu et al. suggest that database engines should explicitly optimize for perceptual constraints, by for example including the visual specification into the physical query planning process [45]. Jugel et al. offer a technique that is one such example: the query algorithms described there return approximate results which nevertheless rasterize to the same image as the exact query result would [25, 26]; ScalarR [7] is another example, mentioned earlier in this section.

Closest to Hashedcubes are *imMens* [34] and *Nanocubes* [32]. The imMens approach combines data reduction, multivariate data tiles, and parallel query processing(using a GPU) to minimize both data cube memory usage and query latency. Its multivariate data tile methods are based on the observation that for any pair of 1D or 2D binned plots, the maximum number of dimensions needed to support brushing and linking is four. Thus, an n-dimensional data cube can be decomposed into a collection of smaller 3- or 4-dimensional projections. Furthermore, these decomposed data cubes are segmented into multivariate tiles, like the ones used by Google Maps. On the other hand, imMens lacks support for compound brushing in more than four dimensions. In comparison, Hashedcubes support any number of dimensions, even if at a potential cost in query latency. Nanocubes is a compact variation of a data cube that can handle a large number of dimensions. It defines a search key that is used to combine aggregations of independent dimensions at varying levels of detail and to maximize shared links across the data structure. *Hashedcubes* is an alternative to *Nanocubes* that eschews a large number of aggregations, allowing both a more compact representation and a much simpler implementation. Hashedcubes uses a partial ordering scheme combined with the notion of pivots [35, 38] to allow fast queries and a simple data structure layout.

BigVis [44] is an R package for the visualization of large datasets and statistical modeling that can store more sophisticated event statistics of events in its bins. Hashedcubes can be extended to include the additional functionality of BigVis. The support for the visualization of origin-destination (OD) data is requested in several applications that handle trajectory data. OD Taxi data visualization [24] and taxi trajectory data visualizations are discussed in [21]. One particularly favorable use case for Hashedcubes is in fact the visual analysis of origin-destination data. The interleaved scheme used in Hashedcubes allows sufficiently-fast queries, while requiring significantly less memory than Nanocubes and imMens.

3 HASHEDCUBES

In this section we will describe the algorithms for building and querying a Hashedcubes. Before giving the full algorithms, however, we will give some intuition on how it works. Hashedcubes combines a few different ideas, and it is easier to see how they work together by progressively building on the properties it exploits. These include hierarchical array partitions, stable sorting, and commutativity of the summaries of a list under permutations of the list.

3.1 Some intuition

First, we note that the fundamental unit we want to visualize in large-scale visualizations such as heatmaps and histograms is a *count*: "how many events happened within this region at some point in time?" "How many events happened on a Tuesday?", and so on. We describe below the intuition behind answering such queries from data stored in arrays.

The following observation is trivial but important: the size of an array does not change when we shuffle it, and so we have much freedom in choosing the order of its elements. The second observation is that when data is stored in a contiguous array, there is a convenient representation for some subsets of this array: we can represent a subset S of elements from an array A by a pair of indices (b,e) such that all elements $A[i]$ for which $b \leq i < e$ are considered to belong to S. We call this pair a *pivot*. If we partition the elements of an array in a certain set of non-overlapping subsets, we can always rearrange the elements such that the chosen subsets of the partition can be represented by pivots (i.e. the subsets are contiguous along the array). In other words, we can represent a partition by permuting the array and storing the corresponding array of pivots. This representation of a partition allows us to, among other things, quickly skip large runs of the data array, while remaining simple and compact. Thirdly, this rearrangement of a partition *also* has significant freedom in its choice: as long as the partition is respected, we can choose the internal order of each subset arbitrarily. Crucially, we can think of each subset of the partition as an array in itself —after all, its elements are all contiguously stored as well— and so we can impose *further* partitions on these subsets, hierarchically. This reordering does not invalidate the first pivot representation, as long as our sorting is *stable* with respect to the first partition.

Now imagine a hypothetical network logging dataset in which we log packets that reach a particular server, and that we are interested in three attributes: day of week (d), hour of day (h), and network port (p) requested. In order to build a Hashedcubes data structure, we need to decide on an *ordering* of these attributes with which to sort the array hierarchically (note that we discuss performance consequences of these choices in Section 7). For this example, assume we will sort in the order we just gave. As we partition the array along each of the attributes, we store the array of pivots that represents the partitions. Note that in dimensions other than the first, this means that the finer partitions will respect the previous sorting: for example, even though all events on a Monday (or any given day of week) will be laid out contiguously in an array, not all events with a given *hour of day* will be: only the events with a given hour *and* day of week. Thus, as we go down the list of dimensions in which we are partitioning, the array of pivots becomes larger, and the partitions themselves become smaller. When the sorting process is finally finished, we will have as many arrays of pivots as there are dimensions in which we are interested in querying the dataset. In our specific case, we will have three pivot arrays: one for the d partition, one for the (d,h) partition, and one for the (d,h,p) partition.

How does this hierarchical sorting help answer queries quickly? For example, if we are interested in plotting a histogram of requests in which bins represent different hours of the day, it is clear that the *second* pivot array is central for this query. Instead of scanning the data array one element at a time, we can scan the array of pivots that represent the sorting on (d,h). If we annotate the pivot arrays with information about the range of attributes of the data they contain, we will be able to make decisions about entire subsets of contiguous data

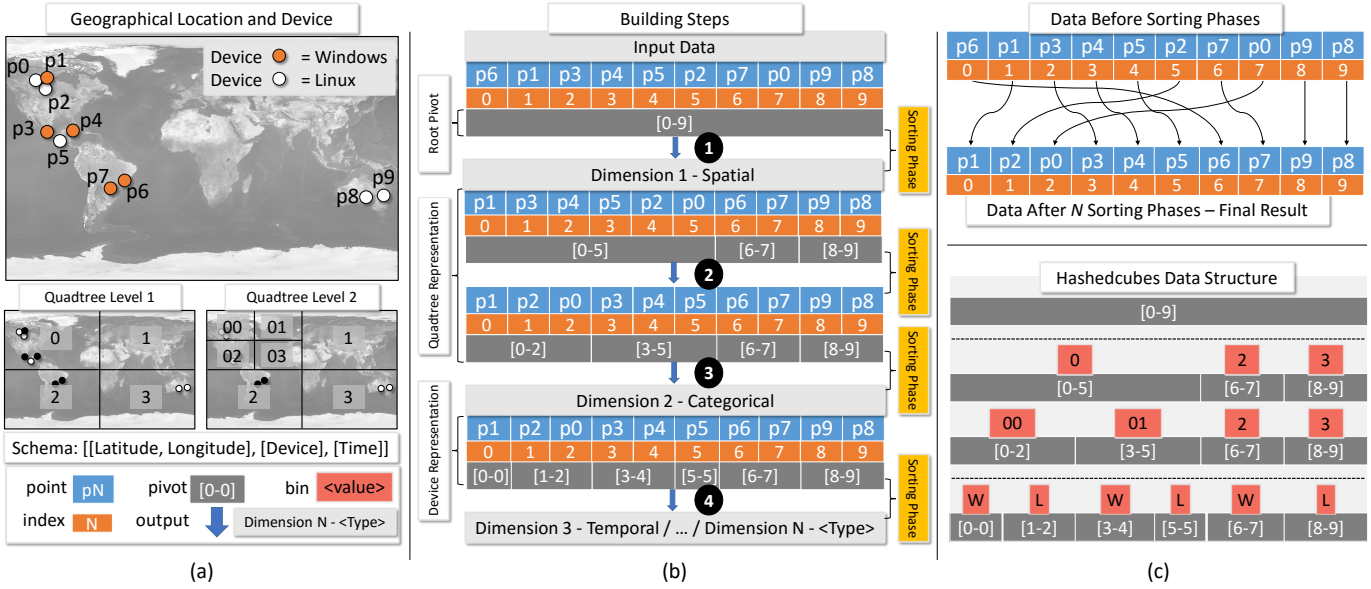


Fig. 2. Overall summary for building Hashedcubes. **(a)** Input dataset of points [p_0,...,p_9] under a spatial-categorical-temporal schema. The complete process is described in Section 3. **(b)** Step-by-step illustration of the process for building arrays of sorted partitions, as explained in Section 3.2. **(c)** Data is loaded (in any order) into a sequential memory and each record is associated with an index (rectangle in orange). The Hashedcubes *construction algorithm* executes multiple sorting phases that result in a array of sorted partitions. After building a Hashedcubes, every pivot delimits a partition. The stored Hashedcubes data structure is shown below. Its memory usage is mainly composed by pivots (each corresponding for two 32-bit integers) and attribute ranges (for the spatial dimension, the range is a 2-dimensional bounding box; for the categorical dimensions, the range is simply an integer value).

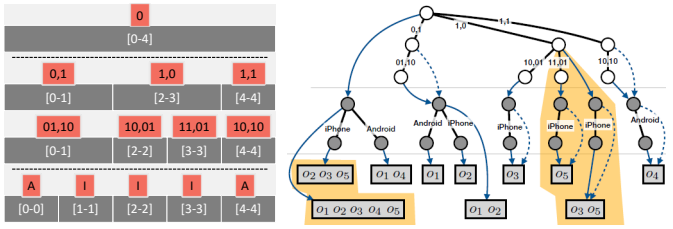
at once. This is already somewhat useful, but imagine, for example, a natural interactive query in which users are interested in studying the same histogram as before, but for a particular subset of days of the week. As we have currently described Hashedcubes, there is no connection between the different pivot arrays, and so we cannot use information about values in one dimension to speed up queries of a different dimension. But this is easy to fix: after sorting on a finer attribute, we annotate the "coarse pivots" with the range of pivots that they represent in the next finer dimension. In our example, the array of d pivots will be annotated with the boundaries they represent on the array of (d,h) pivots; the (d,h) pivots, in turn, will be annotated with the boundaries they represent in (d,h,p) pivots, and so on. Now consider our working queries above again. In the same way that we exploited the query attribute values to skip entire ranges of data values by scanning the (d,h) pivot array, we can scan the d pivot array to skip entire ranges of the (d,h) pivot array itself. This is the central insight behind Hashedcubes. The astute reader will have undoutedbly noticed that if we instead wanted to filter on network ports, we could not escape a scan of a relatively large (d,h,p) pivot array. This is correct, and we discuss this further in Section 3.6.

3.2 Construction Algorithm

The algorithm for building Hashedcubes requires an ordering of dataset dimensions (e.g. first spatial, then categorical, and finally temporal). In what follows, we will sometimes use terms like "above" and "below" to refer to precedence relationships in this ordering. Once defined, a linear array called *Hash* is associated with a root pivot $[0,n-1]$, which represents the initial partition containing the universe of n elements. Each element of the *Hash* array is an integer that points to a record in the dataset. The *Hash* array can be stored in a random or sequential ordering. For every dimension of the indexing scheme, each partition (here forth referred as a bin) of each object is indexed using pivots. Bins have different interpretations for each dimension. Bins represent regions for a spatial dimension, specific values or ranges for a categorical dimension, or time intervals for a temporal dimension.In a input array of n elements all entries belong to the same bin, represented by a pivot $[i_0,i_1]$. Each dimension receives as input a list of pivots and outputs a list of pivots. The first dimension receives as input the root pivot. Subsequent dimensions receive the list of pivots created from the previous dimensions. Sorting is performed in each bin to group elements. The bin delimited by a given pivot is further refined as necessary to create subset bins, represented by a new list of pivots. After processing each dimension a new list of pivots is generated. A hierarchy of pivot lists connects the bins created in each dimension.

Hashedcubes supports three distinct dimension types: spatial, categorical and temporal. The pivot hierarchy for these three dimension types can be built in any order. Since a bin at a given dimension is a subset of a bin in the previous dimension, a list of pivots represents subsets for all previously defined dimensions. This allows to remove dimensions from the representation, which is useful for managing memory consumption. The pivot hierarchy mimics a tree hierarchy since each pivot represents a set that can be further divided into a variable number of subset pivots, but notably, it does not store edges from one dimension to another. Sibling pivots (nodes) are stored as lists. Because each dimension stores collections of pivots, and pivot indices are always offsets into the data array, dimensions can be treated independently of each other. This allows the algorithm which executes queries to skip dimensions that are not referred to by in the query. Furthermore, the cardinality of the subset represented by a pivot can be directly obtained from the pivot indices; this way, the size of an aggregation can be directly determined by the list of pivots themselves.

We use the Figure 2 to illustrate different aspects of Hashedcubes. The input data consists of 10 points using the schema [[Latitude, Lonngitude], [Device], [Time]]. In Figure 2b step 1, the array is re-ordered along the first level of the quadtree and three partitions are created associated to quadrants 0, 2, and 3 (the quadrants that contain points). Three pivots are created ([0-5], [6-7], [8-9]) to delimit these partitions. In step 2 the array is re-ordered along the second level of the quadtree. Note that only the first quadrant of the quadtree is subdivided in this step, and therefore only the partition affected (associated to the pivot [0..5])) is updated, leading to two new pivots ([0..2] and [3..5]). In steps 3 and 4 the process is similar, but using the categorical and temporal dimensions to create further partitions in the data. In the top of Figure 2c we compare the input values of the array to the final re-ordering



(a) Hashedcubes (b) Nanocubes (adapted from [32])

Query	Hashedcubes	Nanocubes
Count[<0,1>] **or** Count[<10,01>,<11,01>]	Pre-computed	Pre-computed
Count[all<Android>] **or** Count[all<iPhone>]	Compute On-the-fly	Pre-computed

Fig. 3. A comparison between the computation of Nanocubes and Hashedcubes. Note that Nanocubes pre-compute more aggregations, which tends to lead to lower query times but larger memory consumption. Hashedcubes, in contrast, uses a sparser set of preaggregations in its query execution engine.

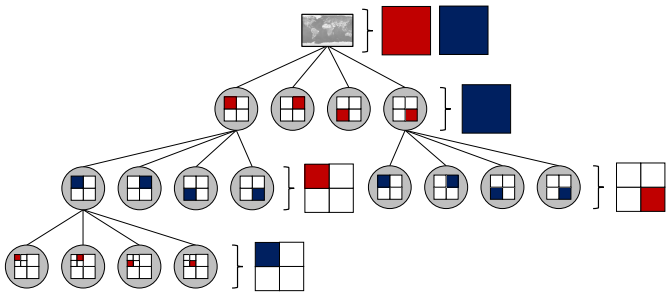


Fig. 4. Multiple spatial dimensions. In this example one quadtree is created for each of the two spatial dimensions, red and blue. The quadtrees are used alternately in Hashedcubes to partition the data.

obtained after successive partitions of the data. In Hashedcubes it suffices to keep the final array along the pivots created at each step to recover the partitions created during these steps. In the bottom of Figure 2c we show the list of pivots created at each step and stored by Hashedcubes. The list of pivots correspond to partitions induced in the first and second levels of the quadtree, and the categorical partition, in this case if device used was Windows (W) or Linux (L).

In contrast to other data cube alternatives [17, 32, 34], Hashedcubes does not precompute aggregations across every possible set of dimensions. Instead, it leverages the pivot hierarchy to compute missing pre-aggregations on-the-fly. Consider in Figure 3 the problem of computing the number of all objects labeled as *Android* or *iPhone* in the categorical dimension. Hashedcubes does not pre-compute this information. Although this means that such queries will require a scan over a potentially large portion of the array, the fact that Hashedcubes stores these in an array (as opposed to a pointer-based data structure) means that the aggregations can be computed relatively efficiently. In fact, allowing these worst-case scenarios to occur is precisely what is responsible for the low memory consumption in Hashedcubes. The query algorithm is described in Section 3.6.

3.3 Spatial Dimensions

Efficiently answering queries involving spatial attributes typically requires the use of hierarchical spatial data structures [39]. In Hashedcubes the spatial dimension is represented as a quadtree, a hierarchical data structure often used to represent geo-spatial data where the space is recursively divided into 4 regions [39]. Each quadtree node is associated with a pivot that delimits the objects contained in that quadrant. If a query matches the exact region represented by a node, then the pivot represents the aggregation result for that query. Otherwise, we compute the minimal disjoint set of nodes that cover the query region. We note that during an interactive session, the viewport region of the screen can be interpreted as a spatial query. Although Hashedcubes can process dimensions in any given order, in our experiments we chose to use the spatial dimension first in the ordering of dimensions to increase the speed in which geo-spatial queries can be answered.

The algorithm for building spatial dimensions associates each record within each pivot range to its current quadtree quadrant. Sorting is used to group records belonging to the same region, and consequently, quadtree nodes store the pivot that delimits the records for that specific subdivision. As we mentioned above, the schemas we use typically start with spatial dimensions. Therefore, the input is a single pivot (root) representing the data universe and only a unique quadtree is allocated.

Hashedcubes supports multiple spatial dimensions, but this process is different from single spatial dimensions. Each spatial dimension is associated with a quadtree. Instead of building each spatial dimension sequentially, Hashedcubes interleaves the construction of each quadtree, refining one level of each quadtree at a time. Consider a dataset of phone calls, with two geographical locations, one from the caller and another from the receiver. The root of the quadtree represents all data. At each level of the quadtree the records are subdivided according to the current spatial attribute (e.g. odd and even levels can be associated to origin and destination locations respectively). By using an interleaved quadtree, queries with multiple region constraints are answered by traversing a unique data structure, since quadtree nodes stores the bounding box and the pivot that matches precisely to all aggregates from that regions. Figure 4 illustrates this process.

Another important aspect of the Hashedcubes quadtree implementation is the minimum leaf size. Every dimension output is the input for the following dimensions, while each pivot is subsequently refined to represent subsets of specific attributes. Smaller pivots cause the creation of a greater number of subsets. Consider Figure 2d. For every input of the *spatial dimension*, it can at most output 2^{2n} subsets, where n is the maximum quadtree subdivision. For every input of *categorical dimension* it can output at most two subsets (*windows* or *linux*). Thus, the output size is directly dependent on the input size. The leaf size is a crucial factor for memory usage and performance of Hashedcubes, and is discussed in Section 7.

3.4 Categorical Dimensions

Categorical attributes of multidimensional datasets are usually divided into specific values or ranges. The processing of such attributes in Hashedcubes produces a list of pivots that groups data in bins for each categorical value or range. By varying the granularity of the Hashedcubes query results, categorical queries form the basis for histograms, binned scatterplots and time series plots.

To process a categorical dimension, each record attribute is tagged and a position in the output list of pivots is computed. This algorithm compares an element against all dimension attributes and returns a bin tag. Once this finishes, the sorted list of pivots is created. For a categorical dimension of n distinct values or ranges, at most n pivots can be created. Hashedcubes stores a structure called *CategoricalNode* which implements a dense vector based on the number of unique attributes. Consider the *categorical dimension* in Figure 2d, which has as input a list of pivots of size 4. Every input creates a *CategoricalNode* that has a vector with two pivots, representing either *Windows* or *Linux*. The result of processing this dimension creates a list of pivots of size 6, with 4 CategoricalNode objects (*object 1*: [0-0],[1,2]; *object 2*: [3,4],[5,5]; *object 3*: [6-7]; *object 4*: [8-9]).

Unlike the processing of multiple spatial dimensions (which are processed in an interleaving fashion), multiple categorical dimensions are generated in sequence.

3.5 Temporal Dimensions

We take advantage of the fact that a pivot represents an interval to represent temporal dimensions. Consider the example of a temporal dimension that needs to be processed to create bins for each different day. The building algorithm classifies each element of the input in the corresponding bin. The result of this process is a sparse list of sets since a bin is created if it has, at least, one record. From this list, a compact list of timestamped pivots is created, as illustrated in Figure 5.

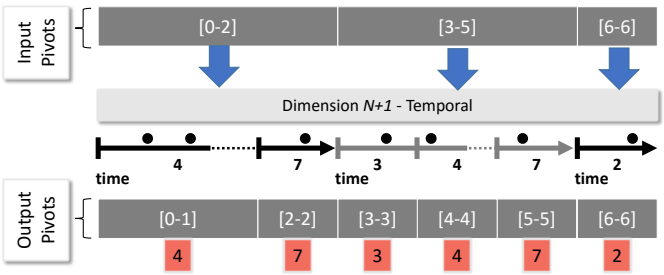


Fig. 5. Temporal dimension indexing. A period of time is represented by a dense list of timestamped pivots. Each black circle represents a record that has been tagged to a specific bin.

The algorithm for building the temporal dimension is similar to the one for categorical dimensions. It tags each record with its respective bin since epoch time. Hashedcubes supports any granularity multiple of milliseconds, and the time interval is defined by the building schema (e.g., 15 minutes, 1 hour, 4 hours, 1 week, etc). Take as an example a schema that aggregates time by the hour, and two records with a difference of 40 minutes. These records are tagged to the same bin, and consequently, represented by a single pivot.

This algorithm enables temporal queries to be efficiently answered without requiring a hierarchical data structure. This is accomplished with two executions of a binary search algorithm, which finds the pivot with the smallest and greatest values from the period of time. This is precisely the same algorithm used by Lins et al.'s Nanocubes [32].

3.6 Queries

A query into a Hashedcubes comprises a set of *clauses*. Each clause corresponds uniquely to a dimension, and defines either *constraints* on values or *group-by* directives (often a dimension will contain both a group-by directive and a value constraint). Constraint clauses specify regions of the dataset to be aggregated over, while group-by clauses indicate partition boundaries for the result, in direct analogy to SQL's group by clause (eg. different bins of a monthly histogram as in a "group by month" SQL clause, or nodes of a quadtree for a multiresolution heatmap plot).

The result of a Hashedcubes query is a list of aggregated pivots. As discussed in Section 3, Hashedcubes does not store precomputed aggregations across every possible set of dimensions. Instead, it materializes only a portion of all combinations (corresponding to a strict prefix ordering of the dimensions as alluded above). The query execution algorithm takes advantage of the pivot hierarchy to compute the missing aggregations on-the-fly, scanning subintervals of dimensions as necessary.

Most queries contain a group-by clause. In queries broken down by latitude and longitude (as in those which generate heatmaps), the spatial dimension is that clause. In queries broken down by categorical attributes or timestamps, any of the multiple categorical or temporal dimensions can be the group-by clause. Take as example the schema in Figure 2d. Assume we are interested in the count of all objects with quadrants 0 and 1 as spatial coordinates and categorical attribute *Windows*. In this case, the result of the query is exactly the contents of a single pivot in that dimension, and no aggregations are necessary. This query is efficient because the constraint clauses form a prefix over the ordering of the dimensions (in fact, it's the entire dimension set). Consider, on the other hand, a query that requests the count of all objects with categorical attribute *Windows*, regardless of spatial coordinates. In this case, there is no single pivot storing the final result, and so it is clear that some on-the-fly aggregation will be required.

The full algorithm proceeds as follows. Initially, the query range is the dataset universe represented by the root pivot $[0, n-1]$. The query result in each dimension is a delimiting list of pivots of the selected data, thus, these lists become the new range query, similar to a breadth first search algorithm that uses two lists, one for expanding and one for temporary storage. This process is iteratively repeated until the last dimension. Note that, unlike tree-based data structures, scans happen along arrays. Such approach tend to offer appealing performance, since the CPU cache automatically optimizes burst memory operations [16, 29].

4 Implementation

The current implementation of Hashedcubes uses a simple client-server architecture. The server reads the data from a file (e.g. CSV tabular files), builds the data structure and enters an event loop that waits for queries from the client. The server is implemented in C++. Since Hashedcubes uses linear-based memory structures such as sorted arrays, it preallocates chunks of memory to avoid the overhead of repeated memory allocations and deallocations, which are common operations in tree-based data structures. Besides the sorting of the index arrays, Hashedcubes does not require any data precomputation prior to building its data structure. The sorting of the data array dominates the construction time, as we discuss in Section 6.2.

For the representation of spatial values, Hashedcubes uses the spherical Mercator projection popular with map tile providers such as OpenStreetMap [19]. Typically, map tiles providers use coordinates (x, y, z) for each tile image. The tuple $[x, y]$ corresponds to integer addresses, while z represents the zoom level, in most cases varying from 0 (maximum zoom out) to 18 (maximum zoom in). Each zoom increment doubles the $[x, y]$ resolution, and consists of 4^n tiles. We choose to limit the spatial coordinates to a maximum of 26 levels: the maximum zoom value plus 8, corresponding to the typical tile size of 256x256. The 26-level subdivision naturally yields a 26-bit address for each of the x and y coordinates, and these addresses can be easily employed for the hierarchical sorting in spatial coordinates.

The server is easily parallelizable since the data structure does not change after building. It exposes the querying API via HTTP (as in Table 1) through a web service implementation that handles concurrent requests in multiple threads. In the front-end, the prototype client is written in Javascript, SVG, and HTML5; notable libraries include D3 [9] and Leaflet [1], as shown in Figure 6.

5 Datasets and Schemas

In this section, we report an evaluation of Hashedcubes using a collection of publicly-available datasets. We collected seven datasets that range from 4.7 million to 1 billion records, including some used in other data cube visualization proposals, as well as the schema they used. In addition, we introduced some variations on the schemata used in previous experiments in order to properly stress the features of both Hashedcubes and previous systems. We summarize all of the schema variations and datasets in Table 2.

5.1 Location-Based Social Networks

Brightkite and Gowalla are two former location-based social networks: users participated by sharing their locations via check-ins events. Both datasets are publicly available in Leskovec's *Stanford Large Network Dataset Collection* [30]. They consist of time and location information of user check-ins, collected by Cho et al. [12]. Brightkite check-ins range from April 2008 to October 2010, and Gowalla from February 2009 to October 2010. We built Hashedcubes using two different schemas for these datasets. The first one replicates the schema used by Nanocubes and encodes latitude and longitude as spatial information, hour of the day and day of the week as categorical variables, and check-in time as temporal variables. The second one replicates the imMens schema and encodes latitude and longitude as spatial information, hour of the day, and day of the month as categorical information. In Figure 1, we use Hashedcubes to visualize Brightkite check-ins in Europe and to highlight Brightkite releases of its iOS app and its 2.0 platform version.

5.2 Airline On-Time Performance

The U.S. Department of Transportation tracks the on-time performance of domestic flights by U.S. air carriers. This dataset was made publicly available in [4, 43], and covers over 121 million flights in a 20 year period, from 1987 to 2008. Records include over 29 fields. We used

Table 1. Subset of queries supported by Hashedcubes HTTP API.

Queries (in natural language)	Spatial	Categorical	Temporal	URL
heatmap of all check-ins in Mondays	drilldown	rollup	rollup	/tile/tile/0/0/0/0/8/where/day_of_week=Monday
hour of day histogram of check-ins in the USA	rollup	drilldown	rollup	/group/hour_of_day/region/0/USA
scatterplot of hour of day against day of week of check-ins in Europe	rollup	drilldown	rollup	/scatter/field/hour_of_day/field/day_of_week/region/0/Europe
time-series of check-ins in Fridays and between Jan and Feb of 2010	rollup	rollup	drilldown	/tseries/tseries/0/Jan-2010/Feb-2010/where/day_of_week=Friday

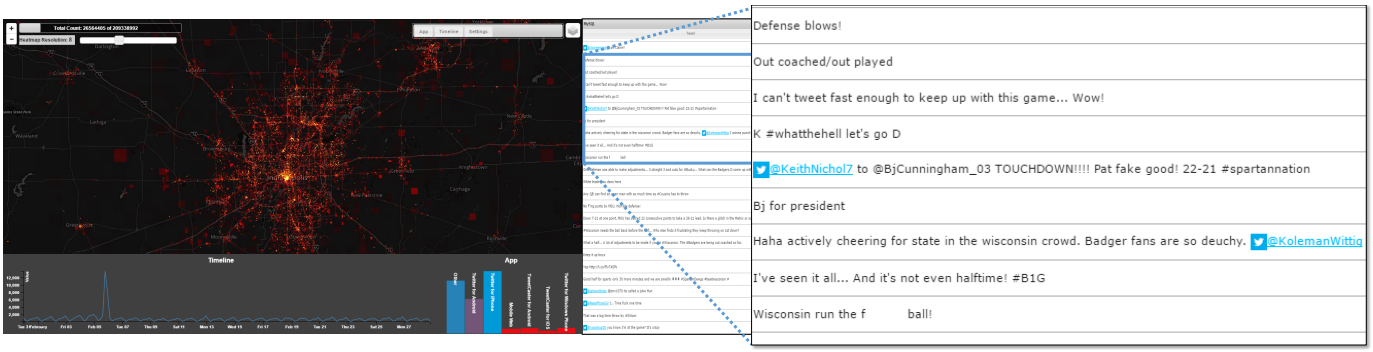


Fig. 6. Visual exploration of the twitter dataset during Super Bowl 2012. In addition to enabling real-time exploration using a wide range of visual encodings, with support to brushing & linking in any dimension, Hashedcubes allows the access to the text of tweets from an external SQL server.

three different schemas for this dataset. The first one encodes the origin airport as spatial information, departure delay and carrier delay as categorical information, and departure delay as temporal information. This is the same schema used in Nanocubes. The second schema is the one used by imMens, and encodes only categorical information. The day of the week, year, carrier, arrival delay and departure delay are the categorical information. Note that the arrival delay and departure delay are encoded as 15 minutes interval bins, and were designed to be visualized using a scatter plot. The last schema is designed to exploit the Hashedcubes ability to work with multiple spatial dimensions, so we encoded origin and destination airports as spatial information.

5.3 SPLOM

The ScatterPlot Matrix (SPLOM) benchmark [28] was designed to stress test the data cube technology, and has been used as validation in recent big data visualization proposals [34, 32]. It consists of a collection of synthetic elements with up to five dimensions. The first, second and fifth dimensions are independent and normally distributed. The third and fourth dimensions are, respectively, linearly and log-linearly dependent with the first. As a synthetic dataset, we used five different bin sizes per dimension, from 10 to 50, and varied the elements from 100 million up to 1 billion to stress test Hashedcubes (Figure 7a).

5.4 Twitter

The data consists of geolocated tweets collected from the (formerly open) Twitter API between November 2011 and June 2012 that originated in the United States. We used two different schemas, namely twitter-small and twitter. The first one encodes the record origin as spatial information, device used as categorical information, and record collection time as temporal information. The second schema adds the application and language, respectively 4 and 15 distinct values, as categorical informations. In Figure 1 we present and overview of tweets in USA, and a close-up in the date and region of Superbowl 2012.

5.5 NYC Yellow and Green Taxis

The NYC Taxi and Limousine Commission (TLC) collects and provides monthly trips records from yellow and green taxis from New York City. Records include over 21 fields that capture pick-up and drop-off times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, driver-reported passenger counts, and others. While yellow taxis are able to pick-up passengers in any of the five NYC boroughs, green taxis are only allowed to pick-up passengers in outer boroughs and in Manhattan above East 96th and West 110th Streets. For each dataset, we used two different schemas, both encoding pick-up and drop-off locations as spatial information. The first schema encodes time as week bins along with categorical information: day of the week and hour of the day. The second schema encodes time as hour bins. In Figure 1, we highlight the use of Hashedcubes to analyze pick-up locations from the green taxis dataset.

6 Performance Results

In this Section we discuss the performance results of Hashedcubes. We compare the Hashedcubes memory usage, construction and query time to recent data cube visualization proposals, namely Nanocubes [32] and imMens [34]. Table 2 summarizes benchmark results for all schema variations and datasets. The number of records (N) in the dataset, quadtree leaf-size, memory usage, time to build and the accumulated number of pivots (P) across all data structure are reported.

6.1 Memory Usage

Memory usage in Hashedcubes is directly proportional to the number of pivots, i.e., the number of used bins per dimension. Figure 7a shows the memory growth for the SPLOM dataset ranging from zero to one billion inserted records. We used five schema variations that range bin size from ten to fifty in each dimension. Records from this dataset are collected from synthetic generators that have a normal distribution, which means that the set of high probability values are quickly sampled, making harder for new records with an unseen bin. It highlights an effect known as *key saturation*. Due to the key saturation effect, most inserted records does not require additional memory since their pivots were already present in the Hashedcubes index, a phenomenon that performs an important role to reduce memory requirements.

When comparing Hashedcubes to recent data cube strategies, memory usage sees a breakthrough from current state-of-the-art data cube proposals, enabling the visualization of a much larger set of scales and more complex schema configurations than imMens and Nanocubes. Compared to Nanocubes, we find a reduction factor of up to 5.2x in the best case, as shown in Figure 7c. Building the Hashedcubes for brightkite, flights, twitter-small and twitter schemas, requires 366MB, 457MB, 4GB and 9.4GB of memory, respectively. For the same schemas, Nanocubes requires 1.6GB, 2.3GB, 10.2GB and 46.4GB, enough for present day servers, but above that of typical notebooks and workstations. imMens uses a dense indexing to speed up aggregation time and to simplify parallel query processing, but this implies that memory usage is proportional to the cardinality of its key space. Furthermore, it lacks support for compound brushing of more than four di-

Table 2. Overall summary of the relevant information for building Hashedcubes.

dataset	objects (N)	leaf-size	memory	time	pivots (P)	schema
splom-10[1,2]	1.0 B	N/A	5 MB	38:32 m	26 K	d1 (10), d2 (10), d3 (10), d4 (10), d5 (10)
splom-50[1,2]	1.0 B	N/A	349 MB	46:28 m	12.7 M	d1 (50), d2 (50), d3 (50), d4 (50), d5 (50)
brightkite[1]	4.5 M	32	366 MB	7 s	6.7 M	lat0, lon0, hour of day (24), day of week (7), time (week)
brightkite[2]	4.5 M	32	375 MB	10 s	6.8 M	lat0, lon0, month of year (12), hour of day (24), day of month (31)
brightkite-alternative	4.5 M	32	468 MB	8 s	8.0 M	lat0, lon0, time (week), hour of day (24), day of week (7)
gowalla[1]	6.4 M	32	743 MB	13 s	12.6 M	lat0, lon0, hour of day (24), day of week (7), time (week)
flights	121.2 M	32	1.5 GB	06:55 m	61.0 M	lat0, lon0, lat1, lon1, departure delay (9), carrier (29), time (4 hours)
flights[1]	121.2 M	32	457 MB	03:56 m	19.5 M	lat0, lon0, departure delay (9), carrier (29), time (4 hours)
flights[2]	50.3 M	N/A	18 MB	12 s	396 K	day of week (7), year (21), carrier (29), arr_delay (147), dep_delay (147)
twitter-small[1]	210.6 M	64	4.9 GB	10:53 m	137 M	lat0, lon0, device (5), time (4 hours)
twitter[1]	210.6 M	64	9.4 GB	12:04 m	203 M	lat0, lon0, app (4), device (5), language (15), time (4 hours)
green-taxis-small	24.5 M	64	788 MB	01:35 m	27 M	lat0, lon0, lat1, lon1, time (hour)
green taxis	24.5 M	64	3.0 GB	01:49 m	52 M	lat0, lon0, lat1, lon1, day of week (7), hour of day (24), time (week)
yellow-taxis-small	224.1 M	64	7.0 GB	18:14 m	243 M	lat0, lon0, lat1, lon1, time (hour)
yellow-taxis	224.1 M	64	12.6 GB	20:38 m	473 M	lat0, lon0, lat1, lon1, day of week (7), hour of day (24), time (week)

[1]Schema used by Nanocubes. [2]Schema used by imMens.

mensions, once it requires computing prohibitively large 5-dimensional data tiles for the adopted approach.

We also evaluated Hashedcubes for schemas with multiple spatial dimensions, a feature that was not supported by Nanocubes and imMens in their initial public releases. For that, we introduced schema variations and two unstudied datasets, namely, the green and yellow NYC taxis. These datasets are particularly hard because both have a very restrict spatial region, thus pushing spatial dimensions data structures to deeper levels of subdivision. Moreover, we tested two time resolutions, by hour and over a week along with day of week and hour of day categorical attributes. We have attempted to create Nanocubes for these schemas, but found them to take a prohibitively large amount of memory. Before killing the nanocube process, we estimated the eventual memory usage of the yellow-taxis-small schema to be around 124GB for a pair of 20-bit quadtree addresses, and 321GB for 25-bit addresses (and an estimated five hours of construction time). We made no attempt to generate a nanocube of the full yellow taxis schema.

6.2 Construction Time

Construction time was a relevant factor when designing Hashedcubes. The construction algorithm was optimized for speed by avoiding repeated memory allocations and deallocations. The bottleneck of this algorithm are the sorting phases, specially when handling spatial dimensions. The pivot hierarchy uses a sorting step for every quadtree, which can be very demanding for datasets with restricted geographical coverage and multiple spatial dimensions, since these cases tend to generate trees next to the maximum recursion depth supported. Compared to Nanocubes, we obtained a reduction factor of up to 30x in the best case, as shown in Figure 7b. On average, the construction time is about 10 times faster.

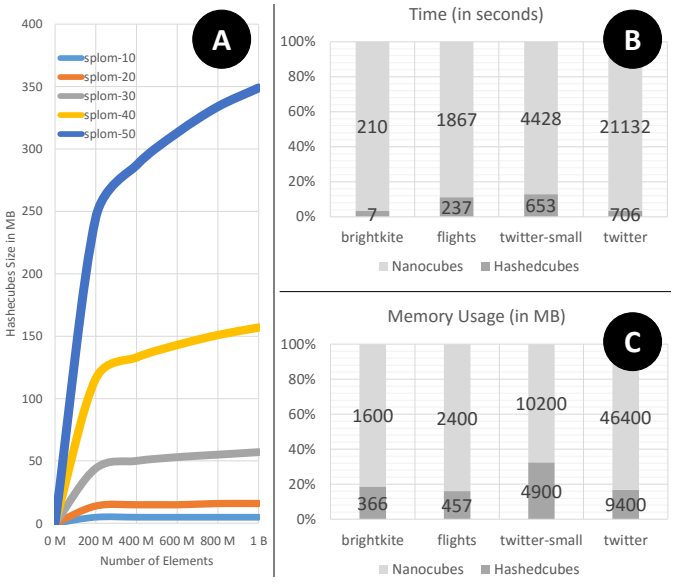
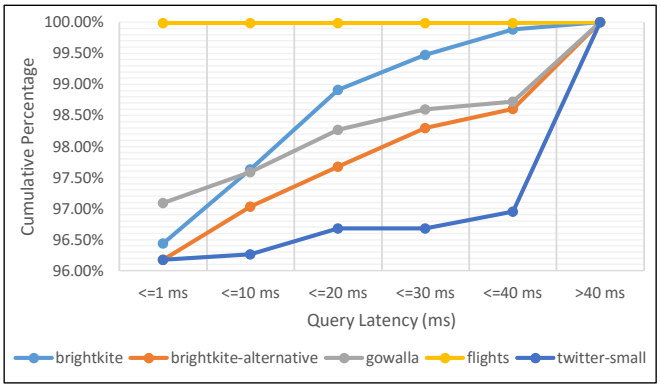


Fig. 7. **(a)** Hashedcubes memory usage growth while inserting SPLOM dataset elements. Notice the *key saturation* effect. **(b)** and **(c)** compare Hashedcubes construction time and memory usage to Nanocubes.



statistic/ dataset	brightkite	brightkite alt.	gowalla	flights	twitter-small
queries N	507880	507880	102430	215980	48190
median	0 ms	0 ms	0 ms	0 ms	0 ms
mode	0 ms	0 ms	0 ms	0 ms	0 ms
mean	0 ms	1 ms	1 ms	0 ms	4 ms
stdev	4.21 ms	11.02 ms	7.19 ms	1.03 ms	66.93 ms
maximum	94 ms	281 ms	114 ms	159 ms	1382 ms

Fig. 8. Cumulative percentages of query latency from real-world scenarios. The vast majority of queries are answered within the real-time budget (<40ms *or* >25fps) for different schemas and datasets.

6.3 Query Time

We used a set of real-world queries graciously provided by AT&T Research to assess query latency. Query requests were collected on the public Nanocubes [32] web site, in which users performed brushing and linking across dimensions of Brightkite, Gowalla, Flights and Twitter datasets. This set provides a sample of common actions when exploring real-time interactive systems using a wide range of visual encodings. Unlike synthetic benchmarks, it allows to validate Hashedcubes in an uncontrolled environment. We implemented a script that translates Nanocube queries to Hashedcubes queries and compares the results of both proposals. For that, we used the same schema from Nanocubes.

Figure 8 shows the cumulative percentages when the set of queries is executed in an Intel Core i7 4790 CPU. We report the median, mode, mean, standard deviation and maximum latency for each of the tested schemas. Typically, Hashedcubes performance level is within the real-time budget (<40ms *or* >25fps); only one in fifty queries takes more than 40ms. The most time-consuming queries are those which require a large number of aggregations of many small pivots. These typically happen when the query constraints are specified over a variable that has been "finely split" over a large range of indices, and yet no filtering in previous dimension rejects has occurred. In the worst case, this might degenerate to a linear scan over the dataset. For other schemas and datasets, Hashedcubes presented similar frequency distribution, consistently answering many queries under 40*ms* for various rollups and drill down test combinations. The server to client latency was dominated by transference of geographical tiles information.

Nanocubes have a very small worst-case value, around 12ms. imMens sustains a 20ms update time on average. It has to be noted, however, that both solutions uses pre-computation and a higher memory footprint in favor of faster queries. Hashedcubes, instead, balances these two variables and allows the real-time exploration and analysis of datasets that previously required a prohibitory amount of space. Moreover, it supports more flexible schema configurations that enables re-ordering and multiple spatial, categorical and temporal dimensions.

7 Discussion

The underlying concept behind Hashedcubes, the pivot hierarchy, can be constructed in any given order. In addition, it allows a natural integration with an external database to complement visual queries. In this Section, we discuss these two extensions and how the quadtree leaf size impacts memory usage and visual accuracy.

Exchanging the Pivoting Order: Exchanging the order in which the variables are sorted impacts both memory usage and running time of specific queries. In Figure 8 we compare two schemas of the same dataset: *brightkite* and *brightkite-alternative*. The set of real-world queries described in Section 6 was used to test the Hashedcubes implementation. The alternative schema, using a spatial-temporal-categorical ordering, notably increases both standard deviation and maximum query time from 4.21ms and 94ms to, respectively, 11.02ms and 281ms. Moreover, it increases memory consumption by 25%. On the other hand, in this schema temporal queries answer much faster since there are fewer pivots that need to be processed by the querying algorithm. Such tradeoffs can be considered by a database administrator to choose one layout over another. Automatically tuning the ordering of variables, or possibly creating redundant Hashedcubes instances to process different queries, is a natural area for future research.

Integration with Database of Record: Large data visualization systems like imMens and Nanocubes, along with Hashedcubes, can be considered *approximate databases*, which means that they use data aggregation which might discard some information of the original record. The underlying concept behind Hashedcubes allows a simple integration with external databases. The retrieval of complementary information can be useful, for example, when datasets have text attributes along with spatial, categorical and temporal values, or when these values are not relevant for the exploratory interactive system itself. All real-world datasets used to validate Hashedcubes contain additional information that is ignored by the schema configurations. In Figure 6 we show the visual exploration of a large dataset associated with the retrieval of complementary data from an external SQL server.

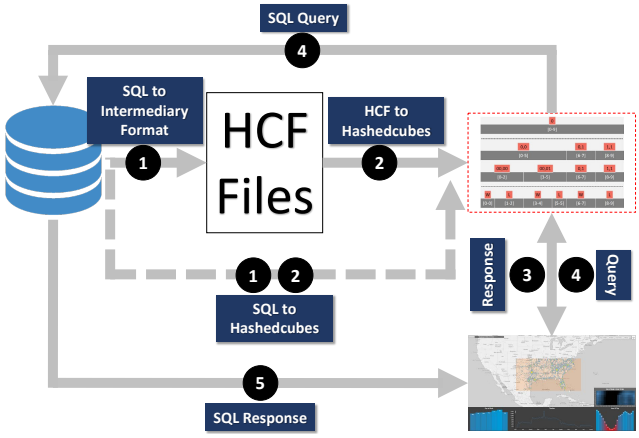


Fig. 9. Hashedcubes supports recovering the original data by using a linking structure. Pivots represent the values from the SQL index, which allows to efficiently match all rows of a given query. Hashedcubes can be built directly from a SQL database or from an intermediary format.

Hashedcubes allows to recover original data by associating the pivot indexes with an external index, for instance, an SQL index. As shown in Figure 9, data is loaded from our intermediary binary format (to obtain faster building times) or directly from the SQL server, and sorted out accordingly to the external ordering. Hashedcubes answers queries in real-time and simultaneously triggers asynchronous SQL queries based on the pivot selection. This natural extension encourages the complement of visual queries with external information.

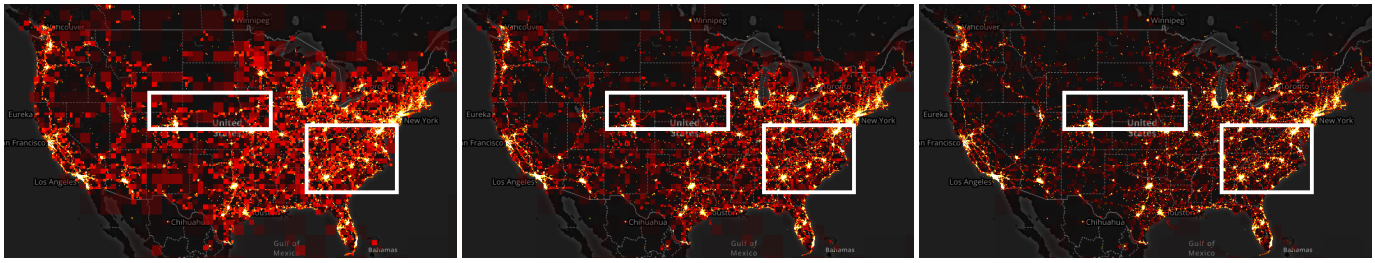
Leaf-Size Trade-off vs Visual Accuracy: During the construction of Hashedcubes, the output of every dimension serves as input for the following dimension, and each pivot is subsequently refined to represent smaller data subsets. Spatial dimensions adopt a minimum quadtree leaf-size to balance running time, memory usage and visual accuracy, as shown in Figure 10 (a), (b) and (c). The leaf-size threshold creates a phenomenon called *truncated pivot*. This indicates that a given spatial region will be no longer subdivided if a minimum leaf-size is reached. Since visual accuracy was a relevant factor when designing Hashedcubes, we implemented a specific heatmap visualization that allows identifying truncated pivot occurrences (Figure 10a).

Truncated pivots are typically found in smaller geographical regions with very low data sampling, an arrangement which might mask outliers. As a workaround to this issue, Hashedcubes users can integrate external databases to recover precise spatial information of a specific region, as discussed previously. It has to be noted, however, that Hashedcubes supports any leaf-size threshold. The default values for the schemas in Table 2 were chosen to achieve a good balance between running time and memory usage while producing a similar visual result when compared to the other data cube visualization proposals (Figure 11).

8 Conclusions and Future Work

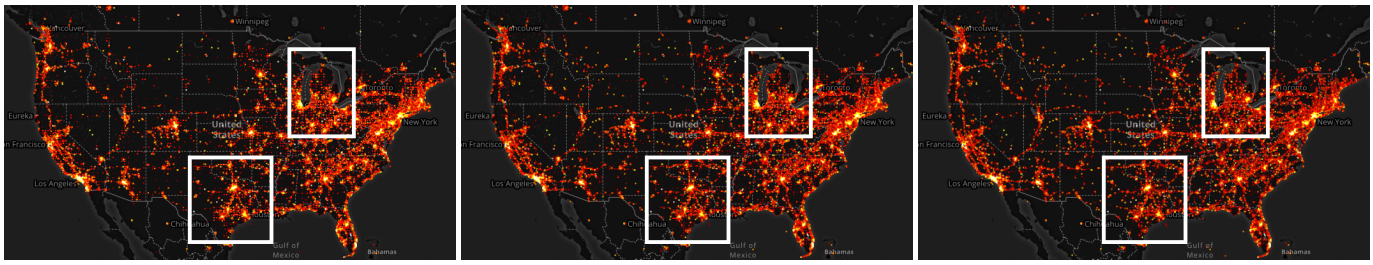
In this paper, we presented Hashedcubes, a fast, easy to implement and memory efficient data structure to answer queries from interactive visualization tools that explore and analyzes large multidimensional datasets. Pivot hierarchy, the underlying concept behind Hashedcubes, enables traversal in any order and allows to include multiple spatial dimensions, which is useful to visualize origin-destinations datasets. Furthermore, it supports access to the original data by integrating the data structure with an external database.

Our major contributions have shown that (i) is possible to represent hierarchical and flat data structures using an optimized pivot schema that is stored in a linear fashion way, and (ii) demonstrated that this leads to memory savings over other data cube visualization proposals, as shown in Section 6. Taking advantage of the performance level given by Hashedcubes, researchers can develop richer and seamless interactive



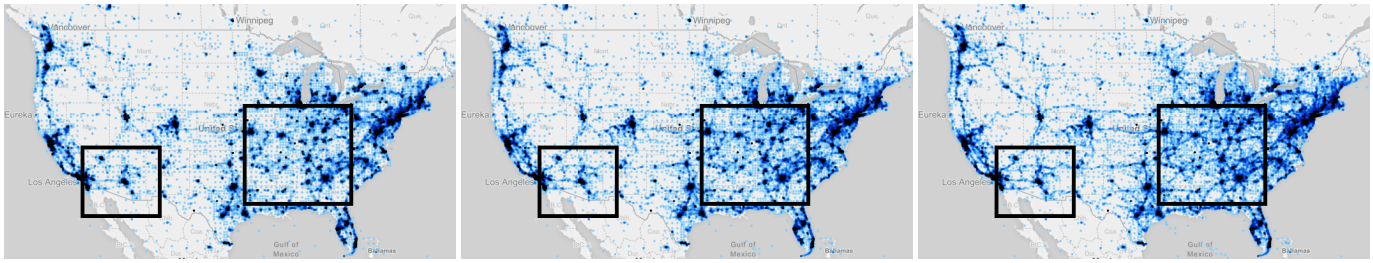
leaf-size: 32 leaf-size: 16 leaf-size: 8

(a) Brightkite overview. Primitive: rectangles, Colormap: red-yellow-white, Density Aware.



leaf-size: 32 leaf-size: 16 leaf-size: 8

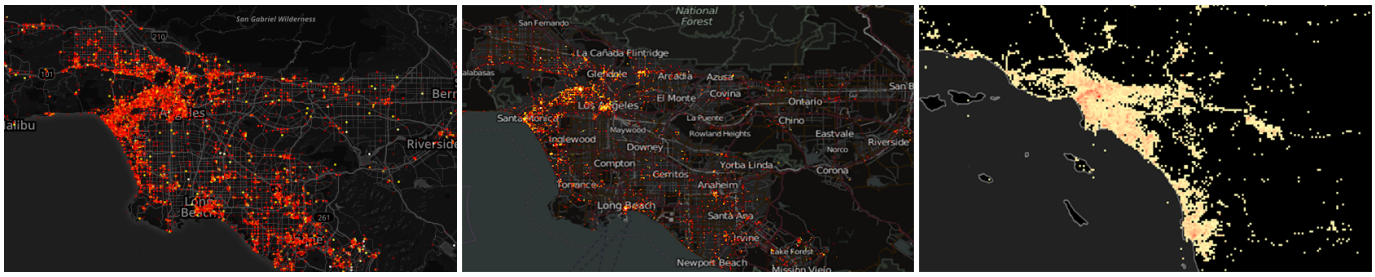
(b) Brightkite overview. Primitive: circles, Colormap: red-yellow-white, Density Aware.



leaf-size: 32 leaf-size: 16 leaf-size: 8

(c) Brightkite overview. Primitive: circles, Colormap: light-blue-dark, Not Density Aware.

Fig. 10. Hashedcubes different heatmap visualizations showcase. Notice the leaf size variation from 32 to 8 by looking into the highlighted regions. It impacts running time, memory usage and visual accuracy. **(a)** allows to identify truncated pivot occurrences by representing them as rectangles. Color is a factor of area and occupancy. **(b)** and **(c)** use circles to represent the center of an aggregated region (i.e., quadtree bounding box).



Hashedcubes (Circles, Density Aware, Leaf-size: 32) Nanocubes imMens (maximum supported zoom by public demo)

Fig. 11. Los Angeles (United States) city view of detailed Brightkite heatmaps from recent data cube visualization proposals. Apart from the use of different colormaps across Hashedcubes, Nanocubes and imMens, what produces a slightly dissimilar visual appearance, Hashedcubes pivot concept enables a high visual accuracy along with reduced memory consumption when compared against other data cube visualization proposals. Notice that Hashedcubes matches Nanocubes visual representation, even though the latter does not experience leaf-size trade-offs.

visualization tools. Moreover, it enables the visual exploration of datasets and schemas that previously take a prohibitory amount of space or time.

As future work, we would like to expand pivot hierarchy concept to automatically find optimal pivoting ordering by calculating a metric that balances running time and memory usage. Since Hashedcubes building algorithms mainly require careful sorting operations that can be adopted to current Web technologies, we also want to explore an exclusively browser-side implementation. Hashedcubes uses a querying algorithm similar to a breadth-first search, with two working lists, one for expanding and another for temporary storage. We envision an alternative approach that use just one list, but that require significant enhancements to the data structure and are left for future work. Another promising research area is the handle of dynamic datasets or streaming data. Hashedcubes can benefit from existing approaches like *Packed-Memory Arrays* [8], a concept that aligns surprisingly well with Hashedcubes pivot notion and its worth to be further investigated. Hashedcubes is available as open source software at https://github.com/cicerolp/hashedcubes.

ACKNOWLEDGMENTS

We would like to thank AT&T Research for providing the set of queries, Capes for the financial support, as well as the anonymous reviewers.

REFERENCES

[1] V. Agafonkin. Leaflet - a Javascript library for mobile-friendly interactive maps, 2014. http://leafletjs.com/.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42. ACM, 2013.

[3] R. Agrawal, A. Kadadi, X. Dai, and F. Andres. Challenges and opportunities with big data visualization. In *Proc. of the 7th International Conference on Management of Computational and Collective intelligence in Digital EcoSystems*, MEDES '15, pages 169–173. ACM, 2015.

[4] American Statistical Association Data Expo. Airline on-time performance dataset, 2009.

[5] I. Assent, R. Krieger, F. Afschari, and T. Seidl. The ts-tree: Efficient time series search and retrieval. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, pages 252–263. ACM, 2008.

[6] G. Battle, R. Chang, and M. Ston. Dynamic prefetching of data tiles for interactive visualization. Technical Report MIT-CSAIL-TR-2015-031, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2015.

[7] L. Battle, M. Stonebraker, and R. Chang. Dynamic reduction of query result sets for interactive visualizaton. In *Big Data, 2013 IEEE International Conference on*, pages 1–8, Oct 2013.

[8] M. A. Bender and H. Hu. An adaptive packed-memory array. *ACM Trans. Database Syst.*, 32(4), Nov. 2007.

[9] M. Bostock. D3.js - data-driven documents, 2015. https://d3js.org/.

[10] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. isax 2.0: Indexing and mining one billion time series. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, pages 58–67. IEEE Computer Society, 2010.

[11] G. Cao, S. Wang, M. Hwang, A. Padmanabhan, Z. Zhang, and K. Soltani. A scalable framework for spatiotemporal analysis of location-based social media data. *Computers, Environment and Urban Systems*, 51:70 – 82, 2015.

[12] 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*, pages 1082–1090. ACM, 2011.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI04: Proceedings Of The 6th Conference On Symposium On Operating Systems Design And Implementation*. USENIX Association, 2004.

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

[15] P. Godfrey, J. Gryz, and P. Lasek. Interactive visualization of large data sets. Technical Report EECS-2015-03, Department of Electrical Engineering and Computer Science, York University, 2015.

[16] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131. ACM, 1983.

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

[18] D. Gupta and S. Siddiqui. Big data implementation and visualization. In *Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on*, pages 1–10, Aug 2014.

[19] M. M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, Oct. 2008.

[20] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *ACM SIGMOD Record*, 26(2):171–182, June 1997.

[21] X. Huang, Y. Zhao, C. Ma, J. Yang, X. Ye, and C. Zhang. Trajgraph: A graph-based visual analytics approach to studying urban network centralities using taxi trajectory data. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):160–169, Jan 2016.

[22] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 277–281. ACM, 2015.

[23] J.-F. Im, F. G. Villegas, and M. J. McGuffin. Visreduce: Fast and responsive incremental information visualization of large datasets. In *2013 IEEE International Conference on Big Data*, pages 25–32. IEEE, 2013.

[24] X. Jiang, C. Zheng, Y. Tian, and R. Liang. Large-scale taxi o/d visual analytics for understanding metropolitan human movement patterns. *J. Vis.*, 18(2):185–200, May 2015.

[25] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. *Proc. VLDB Endow.*, 7(10):797–808, June 2014.

[26] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. Vdda: Automatic visualization-driven data aggregation in relational databases. *The VLDB Journal*, 25(1):53–77, Feb. 2016.

[27] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 472–483, March 2014.

[28] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 547–554. ACM, 2012.

[29] K. Krishnamohan, P. Farmwald, and F. Ware. Prefetching into a cache to minimize main memory access time and cache size in a computer system, Mar. 12 1996. US Patent 5,499,355.

[30] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[31] S. Li, S. Dragicevic, F. A. Castro, M. Sester, S. Winter, A. Coltekin, C. Pettit, B. Jiang, J. Haworth, A. Stein, and T. Cheng. Geospatial big data handling theory and methods: A review and research challenges. {*ISPRS*} *Journal of Photogrammetry and Remote Sensing*, pages –, 2015.

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

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

[34] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. In *Proceedings of the 15th Eurographics Conference on Visualization*, pages 421–430. Eurographics Association, 2013.

[35] B. Mora. Naive ray-tracing: A divide-and-conquer approach. *ACM Trans. Graph.*, 30(5):117:1–117:12, Oct. 2011.

[36] K. Morton, M. Balazinska, D. Grossman, and J. Mackinlay. Support the data enthusiast: Challenges for next-generation data-analysis systems. *Proc. VLDB Endow.*, 7(6):453–456, Feb. 2014.

[37] S. Nepomnyachiy, B. Gelley, W. Jiang, and T. Minkus. What, where, and when: Keyword search with spatio-temporal ranges. In *Proceedings of the 8th Workshop on Geographic Information Retrieval*, GIR '14, pages 2:1–2:8. ACM, 2014.

[38] C. Pahins and C. Pozzer. Improving divide-and-conquer ray-tracing using a parallel approach. In *Proceedings of the 2014 27th SIBGRAPI Conference on Graphics, Patterns and Images*, pages 9–16. IEEE Computer Society, 2014.

[39] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., 2005.

[40] J. Shieh and E. Keogh. isax: Indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 623–631. ACM, 2008.

[41] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the petacube. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 464–475. ACM, 2002.

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

[43] H. Wickham. Asa 2009 data expo. *Journal of Computational and Graphical Statistics*, 20(2):281–283, 2011.

[44] H. Wickham. Bin-summarise-smooth: a framework for visualising large data. Technical Report , had.co.nz, 2013.

[45] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems: Vision paper. *Proc. VLDB Endow.*, 7(10):903–906, June 2014.