TopKube: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data

Fabio Miranda, Lauro Lins, James T. Klosowski, *Member, IEEE,* and Claudio T. Silva, *Fellow, IEEE*

Abstract—From economics to sports to entertainment and social media, ranking objects according to some notion of importance is a fundamental tool we humans use all the time to better understand our world. With the ever-increasing amount of user-generated content found online, "what's trending" is now a commonplace phrase that tries to capture the zeitgeist of the world by ranking the most popular microblogging hashtags in a given region and time. However, before we can understand what these rankings tell us about the world, we need to be able to more easily create and explore them, given the significant scale of today's data. In this paper, we describe the computational challenges in building a real-time visual exploratory tool for finding top-ranked objects; build on the recent work involving in-memory and rank-aware data cubes to propose TopKube: a data structure that answers top-k queries up to one order of magnitude faster than the previous state of the art; demonstrate the usefulness of our methods using a set of real-world, publicly available datasets; and provide a new set of benchmarks for other researchers to validate their methods and compare to our own.

Index Terms—Interactive Visualization, Data Cube, Top-K Queries, Rank Merging.

1 Introduction

Ranks and lists play a major role in human society. It is natural for us to rank everything, from movies to appliances to sports teams to countries' GDPs. It helps us understand a world that is increasingly more complex by only focusing on a subset of objects. There is probably no better way to describe a decade than by ranking its most popular songs or movies. One just needs to look at the *Billboard Hot 100* of any year to gain insight into how the majority of society used to think and behave. *High Fidelity* [1] describes a person obsessed with compiling top-five lists for every occasion; as summarized in the book, the act of ranking gives structure to life, by clarifying the past [2].

With the ever-increasing amount of user-generated content found online, ranks have never been so popular to our cultural landscape. *"What's trending"* has become a commonplace phrase used to capture the spirit of a time by looking at the most popular microblogging hashtags. The same way that the most popular songs can be used to describe the *zeitgeist* of a year or a decade, *what's trending* can be used to describe the spirit of a day or even an hour.

In addition to the deluge of new user-generated data, the ubiquity of GPS-enabled devices provides further insight into the people creating this content by providing, in many cases, their location when posting a message to Facebook, uploading a picture to Flickr, checking into Foursquare at their favorite restaurant, or posting a review about a new product they just bought. The location information provides a much more interesting, and more complicated, version of the ranking problem. Now, not only are we interested in what is trending over time, but also over space, which can range from the entire world all the way down to a city street. What might be trending in one neighborhood of a city, may be completely different from other neighborhoods in other cities, and these may be completely different from the overall global trend across the country. Thus, the ability to explore these ranks at different spatial and temporal resolutions is important for understanding our world and the people in it.

A system that can efficiently process large amounts of data and come up with answers in a few minutes or seconds can be applied to many important problems, including those described here. In recent years, though, with the explosion of data gathering capabilities, there is a growing demand for interactive tools with sub-second latencies. Problems for which no automatic procedure exists that can replace human investigation of multiple (visual) data patterns require exploratory tools that are driven by a low latency query solving engine. Making an analyst wait too long for the query answer might mean breaking her flow of thought, and be the difference between capturing an idea or not.

We have recently seen a growth in research revisiting previous techniques and proposing new variations for solving exactly the problem of low latency queries for the purposes of driving visual interactive interfaces. From the perspective of enabling fast scanning of the data at query time, works like MapD [3] and imMens [4] use the parallel processing power of GPUs to answer queries at rates compatible with interactive interfaces. From a complementary perspective, Nanocubes [5] describes how to pre-aggregate data in an efficient but memory intensive fashion to allow for light-weight processing of queries in real-time.

In this paper, we also follow this path of describing techniques for low-latency querying of large data for exploratory visualization purposes. We propose an extension to the efficient pre-aggregation scheme described in Nanocubes [5] to cover an important use case that was not discussed: interactively ranking top objects from a large data collection incident to an arbitrary multi-dimensional selection. For example, we would like to answer queries such as: "What are the top Flickr tags?" for manually selected spatial

• *F. Miranda and C. Silva are with New York University. Email: {fmiranda,csilva}@nyu.edu. L. Lins and J. Klosowski are with AT&T Labs. Email: {llins,jklosow}@research.att.com*

Manuscript received March 6, 2016; revised October 4, 2016.

regions such as Europe, California, Chicago, or Times Square. Furthermore, we want to be able to determine how the popularity of these tags evolves over time, as dictated by the end-user's interests, all at interactive rates.

We show that the state of the art is not able to compute such queries fast enough to allow for interactive exploration. TOPKUBE however, is a rank-aware data structure that is able to compute top-k queries with sufficiently low latency to allow for interactive exploration. Through a set of benchmarks which we make available publicly, we show that our proposal is up to an order of magnitude faster than the previous state of the art. More specifically, we can summarize our contributions as the following:

- A rank-aware data structure that allows for interactive visual exploration of top-k queries, with results up to one order of magnitude faster than previous work.
- A set of case studies that demonstrate the utility of our method using real-world, publicly available datasets, ranging in size up to hundreds of millions of records.
- A new set of publicly available benchmarks for others to validate their methods and compare to our own.

2 RELATED WORK

The challenge of visualizing large datasets has been extensively studied over the years. Most techniques usually propose some form of data reduction: they try to aggregate a large number of points into as few points as possible, and then visualize that smaller aggregation. The original dataset is reduced to a smaller, sometimes bounded, version. Such reductions try to convey most, if not all, of the properties of the original dataset while still being suitable for interactive visualization. Perceptually approximate techniques [6], [7], which utilize data reduction, maintain interactivity while returning visual approximations that approach the exact results. Sampling [8], filtering [9] and binned aggregation [10] are among the most popular reduction techniques. Even though sampling and filtering reduce the number of items, it comes at the price of missing certain aspects of the data, including outliers. As pointed out by Rousseeuw and Leroy [11], data outliers are an important aspect of any data analysis tool. Binned aggregation, however, does not have such limitations. The spatial domain is divided into bins, and each data point is placed into one of those bins. As such, binning does not remove outliers and also preserves the underlying density behavior of the data.

The *visual* exploration of large datasets, however, adds another layer of complexity to the visualization problem. Now, one needs to query the dataset based on a set of user inputs, and provide a visual response as quickly as possible, in order not to impact the outcome of the visual exploration. In Liu and Heer [12], the authors present general evidence for the importance of low latency visualizations, citing that even a half second delay in response time can significantly impact observation, generalization, and hypothesis rates. Systems such as imMens [4], Nanocubes [5], and DICE [13] leveraged a data cube to reduce the latency between user input and visualization response. Data cubes have been explored in the database community for a long time [14], but in the visualization community, they were first introduced in 2002 by Stolte et al. [15], [16]. All of these techniques, however, are limited to simple data types, such as *counts*. They were designed to answer queries such as: *"How many pictures were uploaded from Paris during New Year's Eve?"* or *"How many GitHub commits happened on the West Coast?"*. Our data structure goes beyond that. We aim to answer more detailed queries, such as *"What were the most popular image tags for all the pictures uploaded from Paris?"* or *"What are the GitHub projects with most commits in the West Coast?"*.

The notions of ranking and top-k queries were also first introduced by the database community. Chen et al. [17] presents a survey of the state of the art in spatial keyword querying. The schemes can be classified as spatial-first, text-first or a combination. Spatial-first structures create hierarchical spatial structures (e.g. grids [18], or R-trees [19]), and insert textual data into the leaf nodes. Text-first structures organize the data by considering the textual elements, and then linking a spatial data structure to it, keeping track of its occurrence in the space [18], [20]. Combined structures create data structures that handle both spatial and textual elements simultaneously [21]. The previous data structures, however, focus on building indexing schemes suitable for queries where the universe of keywords is restricted. In other words, given a set of keywords, rank them according to their popularity in a region. If there is no keyword restriction or the number of restricted keywords is too large, then such proposals become infeasible. Our proposal is much broader: we are able to compute the rank of most popular keywords even if there is no keyword restriction.

Rank-aware data cubes were also proposed in the database community. Xin et al. [22] defined the *ranking cube*, a rank-aware data cube for the computation of top-k queries. Wu et al. [23] introduced the ARCube, also a rank-aware data cube, but one that supports partial materialization. Our proposal differs from them in two major ways: we specialize our data structure to better suit spatiotemporal datasets, and we demonstrate how our structure provides low latency, real-time visual exploration of large datasets.

Another related database research area is top-k query processing: given a set of lists, where each element is a tuple with key and value, compute the top-k aggregated values. Several memory access scenarios led to the creation of a number of algorithms [24]. The NRA (no random access) algorithm [25] considers that all lists are sorted and that the only possible memory access method is through sorted access (i.e. read from the top of each list). The TA (threshold algorithm) [25] considers random access to calculate the top-k. More recently, Shmueli-Scheuer [26] presented a budget-aware query processing algorithm that assumed the number of memory reads is limited. We propose a different top-k query processing algorithm, suitable for our low latency scenario. We show that, due to the high sparsity of the merged ranks, past proposals are not suitable.

Similar to what we are trying to accomplish, Birdvis [27] displays the top words in a given region; however, the technique does not scale to more than a few hundred words. Wood et al. [28] also present a visual exploration of tags in maps, using a standard MySQL database; but they are limited to less than 2 millions words and they do not present any time measurements.

Although orthogonal to the core investigation done here, we lastly mention some of the visualization tech-

niques used strictly for display purposes of ranked objects. RankExplorer [29] proposes a modified theme river to better visualize ranking changes over time. Lineup [30] presents a visualization technique for multi-attribute rankings, based on stacked bars. A Table [31] proposes an enhanced soccer ranking table, with interactions that enable the exploration of the data along the time dimension. Our work enables these types of visualizations to be driven at interactive rates, rather than competes with them by offering a new visualization method.

3 Motivation

We begin with a simple example. Assume a data analyst is studying shots in National Basketball Association (NBA) games from a table like the one below (only three rows shown). Every row represents a shot in a game. The first row indicates that LeBron James from Cleveland took a shot in the 5th minute of a game from the court coordinates $x = 13$, and $y = 28$; the shot missed the basket and he scored 0 points with that shot. The second row represents a successful two point shot by Rajon Rondo from Boston, and the third row represents a successful three point shot by LeBron James.

team	player	time	pts	x	y
CLE	L. James	5	0	13	28
BOS	R. Rondo	5	2	38	26
CLE	L. James	7	3	42	35

TABLE 1

3.1 Generality vs. Speed

With such data and a SQL-like interface, an analyst could generate many insightful aggregations for this data. The query: ``SELECT player,count(*) AS shots FROM table GROUP BY player ORDER BY shots DESC LIMIT 50'' would generate a ranked list of the 50 players that took the most shots, while the following query would rank the top 50 players by points made: ``SELECT player,sum(pts) AS points FROM table GROUP BY player ORDER BY points DESC LIMIT 50''.

Although an SQL engine provides a lot of power and flexibility, these characteristics come with a cost: solving certain queries may require scanning all records stored in a potentially large database. Such an expensive computation may result in an analyst wasting precious time, or worse, breaking a flow of ideas and losing a potential insight. To solve these expensive queries more quickly, there are two alternatives: (1) expand the raw computational power of the query engine (e.g. using more CPU or GPU cores); or (2) anticipate important use cases and precompute information from the data so that query solutions can be composed more cheaply during data exploration. The idea of a *data cube* is essentially that of (2): make an explicit encoding of multiple aggregations a user might query later (e.g. store all possible group-by queries on a table using SUM). Note that we can think of solution (2) as being implemented by first running a big scanning query like in (1), but with a goal of storing enough information to allow for a fluid experience later.

In the Nanocubes paper [5], the authors follow this second approach, and observe that for spatiotemporal datasets with a few categorical dimensions, by carefully encoding the aggregations in a sparse, pointer-based data structure, they could represent, or *materialize*, entire data cubes efficiently. They report on multiple interesting use cases where these data cube materializations of relatively large datasets could fit into the main memory of commodity laptops, enabling aggregations (of the pre-defined measure of interest), at any scale and dimension (from city blocks to whole countries, from minutes to years), to be computed at interactive rates.

From a very high level, the Nanocubes approach is all about *binning* and *counting*. Each dimension in a Nanocube is associated with a pre-defined set of *bins*, and the technique consists essentially of pre-computing a mapping between every combination of bins (from the different dimensions) into some *measure* of all the records incident to that combination of bins. In the simple NBA example, the measure of interest could be the number of shots taken, but in other datasets this could be the number of lines of code, cell phone calls, hashtags, miles driven, or any other simple data value. Thus, the Nanocubes approach trades off flexibility (by requiring the measure of interest and binning scheme to be determined ahead of time) for interactivity (by providing rapid query responses during visual exploration of the data). It should also be noted that the original data records are not stored in the bins (just the counts) and therefore are not available as part of the Nanocube data structure.

The main drawback of in-memory data cubes as proposed by [5] is clear: even a minimal representation of a data cube tends to grow exponentially with the addition of new dimensions. In other words, there is no miracle solution to the curse of dimensionality. On the other hand, from a practical perspective, there seems to exist a sweet spot in terms of computing resource utilization where the application of in-memory data cubes can really be the driving engine of interactive exploratory experiences that would otherwise require prohibitively larger amounts of infrastructure.

With this context in mind, in this work we want to investigate the use of in-memory data cubes to drive an important use case for visualization that was not efficiently solved by Nanocubes. Namely, if we perform a multi-dimensional selection that contains millions of objects, how can we efficiently obtain a *list* of only the top-k most relevant objects with respect to our measure of interest. For example, consider the selections we made on the basketball court example in Figure 1. We want to compare the top-20 players that take shots on the left 3-point corner (orange) versus players that take shots from the right 3-point corner (blue). In this case, knowing that there are only a few hundred players in the NBA each year, it would not be computationally expensive to scan all players to figure out the top 20, but there are many other cases such as GitHub projects, Flickr images, and Microblog hashtags, where having to scan millions of objects can result in unacceptable latencies.

4 Multi-Dimensional Binning Model

In the following, we establish our own notations for well-known concepts in the database literature so that we can have a minimal, self-contained, and precise language to refer to when presenting the various data structures.

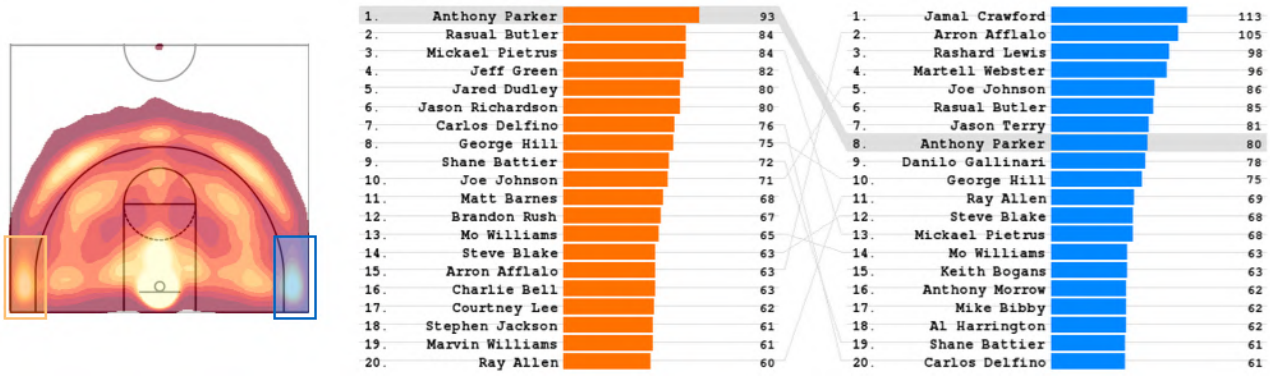


Fig. 1. Ranking NBA players by number of shots from the left 3-point corner (orange) and right 3-point corner (blue) for the 2009-2010 season. The left image is a heatmap of all shots: brighter colors indicate more shots were taken from that location. The hotspot clearly identifies the basket.

The core abstraction used by Nanocubes [5], and shared by our proposal of TopKube, is that of associating records with bins in multiple dimensions: a *multi-dimensional binning model*. For example, it is natural to associate the NBA shot records in Table 1 with dimensions `team`, `player`, `time`, `points`, and `location` (note we chose here to combine columns `x` and `y` into a single `location` dimension). Each player possibility can be associated its own bin in the `player` dimension. Team and points are handled similarly. For time, we could choose a one minute resolution and have each minute of the game be a bin in the `time` dimension; for location, we could have a 1ft. $\times$ 1ft. grid model of the court area and have each cell in this grid be a bin in the `location` dimension. Note that, in this modeling, each record is associated to one and only one bin in each dimension. More abstractly, in our formalism, we assume each dimension i has a *set of finest bins*, denoted by B'_i, and a record is always associated to a single finest bin in each dimension.

In addition to the set of finest bins B'_i associated with dimension i, we define the notion of *coarser* bins, or bins that "contain" multiple finer bins. For example, in the location dimension, we could group adjacent finest grid cells into 2x2 groups and make each of these groups a coarser bin cell in the `location` dimension. The interpretation of coarser bins is simply that if a record is associated with a finer bin b then it is also associated with a coarser bin that "contains" b. In the binning model we define here, we assume that the *set of all bins* B_i associated with dimension i forms a *hierarchy* $H_i = (B_i, \pi_i)$ where its leaves are a partition of finest bins B'_i. The containment function, $\pi_i : B_i \to B_i$ associates every bin b to another bin $\pi_i(b)$ which is either the smallest (i.e. finest) bin in H_i that contains b (in case $b \neq \pi_i(b)$) or it is the coarsest (or root) bin in H_i (in case $b = \pi_i(b)$).

An *n-dimensional binning schema* S is defined as an n-ordered list of hierarchies: $S = (H_1, \ldots, H_n)$. In order to extend the *finest sets of bins* for the `player` dimension, $B'_{\texttt{player}}$, into a valid bin hierarchy $H_{\texttt{player}} = (B_{\texttt{player}}, \pi_{\texttt{player}})$, we could include an additional coarse bin that serves as the root of this 1-level hierarchy by making all bins in $B'_{\texttt{player}}$ its direct children. This is indeed the natural way to model categorical dimensions with few classes. For dimensions where the number of finest bins is not small, it is best to use multi-level hierarchies so that data can later be accessed more efficiently in a level-of-detail fashion. For example, a natural way to model spatial dimensions is by using a quadtree bin-hierarchy; for a temporal dimension, in TopKube, we use a binary-tree bin-hierarchy. Given an n-dimensional binning schema S, we say that the Cartesian product $\mathcal{B} = B_1 \times \ldots \times B_n$ is the *product-bin set* of S, and that an element $\beta \in \mathcal{B}$ of this set is a *product-bin* of S.

To define a *multi-dimensional binning model*, it remains to formalize the notion of which records in a dataset are associated to which bins and product-bins. Let R be a set of records and S a binning schema. If we provide an *association function* $a_i : R \to B'_i$ for each dimension i that assigns a unique *finest bin* in B'_i for every record, we can naturally and uniquely define an association relation $A \subseteq R \times \mathcal{B}$ between records and product-bins. Here is how: (1) we say a general bin $b_i \in B_i$ is *associated with* record r if either $b_i = a_i(r)$ or b_i is an ancestor of $a_i(r)$ in H_i; (2) a product-bin $\beta = (b_1, \ldots, b_n)$ is associated with record r, denoted by $(r, \beta) \in A$ if b_i is associated with record r for $1 \leq i \leq n$.

A *multi-dimensional binning model* is thus a triple $M = (S, R, A)$, where S is a binning schema, R is dataset of records, and A is an association relation between records and product-bins $\mathcal{B}$ from schema S. Given a product-bin β we use $A(\beta)$ to denote its associated set of records in model M (i.e. $A(\beta) = \{r \in R : (r, \beta) \in A, \beta \in \mathcal{B}\}$). Analogously, we use $A(r)$ for a record r to denote its associated set of product bins (i.e. $A(r) = \{\beta \in \mathcal{B} : (r, \beta) \in A\}$).

4.1 Measure on a multi-dimensional binning model

The notion of product-bins in our model provides a way to refer to groups of records through their multi-dimensional characteristics. In our running NBA example, all shots of LeBron James would be specified by $A(\beta_{LJ})$, where the product-bin $\beta_{LJ} \in \mathcal{B}$ consists of the coarsest (root) bin in the bin-hierarchy of all dimensions, except on the player dimension where we would have the bin for LeBron James. If instead we want to refer to LeBron James' shots in the first minute of a game, we would replace the root bin in the `time` dimension of β_{LJ} with the bin for minute 1 of the game.

One natural approach to analyzing a set of records through their multi-dimensional characteristics is through some notion of "size". For example, instead of listing all NBA shots from the 3-pt left corner, we could simply be interested in how many shots happened in that region, or what the average or median distance from the basket was for

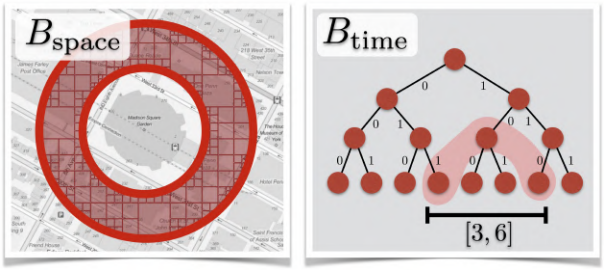


Fig. 2. Dimensions of *space* and *time* are represented as bin hierarchies. (left) B_{space} is a quad-tree hierarchy: here we show a 624 bin selection around Madison Square Garden, NY; (right) B_{time} is a binary hierarchy; in red we show 3 bins corresponding to the interval $[3, 6]$.

all of those shots. A *measure* for a multi-dimensional binning model M is simply a real function $\mu : \mathcal{B} \to \mathbf{R}$ that associates a number to any product-bin β of M, which captures some notion of "size" for the set of incident records $A(\beta)$. In the target applications we are interested in, we want to access measure values not for just one product-bin at a time, but for sets of product-bins that are semantically meaningful together. For example we might be interested in the spatial region on the left of Figure 2 that consists of multiple bins. In general, one cannot derive the measure value of the union of a set of product-bins by combining the measure values of the individual product-bins. The median distance of an NBA shot is such an example: we cannot derive the median of the union of two sets of values by knowing the median of each individual set. We avoid this problem here by restricting our universe to those of *additive* measures only. We start with a real function $\mu : R \to \mathbf{R}$ that associates a number to each record from model M and extend this function to the whole set of product-bins by using additivity $\mu(\beta) = \sum_{r \in A(\beta)} \mu(r)$. Additive measures can naturally count occurrences (e.g. how many records) by making $\mu(r) = 1$, or measure weight sums by making $\mu(r) = w_r$. In addition to scalars, we can also generalize additive measure to produce real vectors. For example, by making $\mu(r) = (1, w_r, w_r^2)$ additivity will yield a 3d vector on any product-bin and union of product-bins (just sum the vectors). In this 3d measure example, it is possible to post-process the vector entries to derive mean and variance of weights for any set of product-bins (mean: divide second entry by first entry). Correlations can also be derived by post-processing an additive measure [32]. In the remainder of this paper we assume simple additive scalar measures. We do not deal with post-processed ones.

We refer to the combination of a multi-dimensional binning model M with a measure μ to its product-bins as a *measure model* $M[\mu]$. The idea of precomputing and representing a *measure model* $M[\mu]$ so that we can quickly access $\mu(\beta)$ for any product-bin β is essentially the well-known notion of a *cube relational operator* (if all hierarchies in the model are all 1-level) or the more general *roll up cube relational operator* (if some hierarchies have 2 or more levels). Note that in practice, when precomputing such measure models, one does not expect to be able to retrieve the original records $A(\beta)$, but only its measure $\mu(\beta)$.

4.2 Nanocubes

In Nanocubes [5], the authors propose an efficient encoding of a measure model $M[\mu]$ with an additional special encoding for one *temporal* dimension. Nanocubes uses a pointer-based sparse data structure to represent the product-bins β that have at least one record associated to it, and tries to make every product-bin that yields the same set of records refer to the same memory location encoding its measure value. Conceptually, we can think of Nanocubes as an encoding to a mapping $\{\beta \mapsto \mu(\beta) : \beta \in \mathcal{B}, A(\beta) \neq \varnothing\}$. For the temporal dimension, the particular μ values are stored in Nanocubes as *summed area tables*:

$$\beta \mapsto ((t_1, v_1), (t_2, v_1 + v_2), \ldots, (t_p, v_1 + \ldots + v_p)), \quad (1)$$

where t_i's are all the *finest temporal bins* associated to the records in $A(\beta)$, they are sorted $t_i < t_{i+1}$, and v_i is the measure of $\mu(\beta, b_{\texttt{time}=t_i})$, i.e. product-bin with the added constraint in the time dimension. Note that by taking differences of values from two different indices of a summed area table one can quickly find the value of any query $(\beta, [t_a, t_b])$, where β is a product-bin (without the time dimension) and $[t_a, t_b]$ is the time interval of interest.

5 TopKube

A Nanocubes-like approach can efficiently retrieve a measure of interest for any pre-defined "bucket" (i.e. a product-bin plus a time interval). This capability can be handy for many applications, but is especially useful for interactive visualizations where each element presented on a screen (e.g. bar in a barchart, pixel in a heatmap) is associated with one of these "buckets" and encoded (e.g. bar length, pixel color) based on its value. However, suppose that, instead of simply accessing the measure associated with specific buckets, we are actually interested in identifying the top-k valued objects from a potentially large set of buckets. For example, "Who are the top-20 players that make the most shots from the right-hand 3-point corner of the basketball court?" (blue selection and ranking shown in Figure 1).

Since there is no ranking information encoded in a Nanocube, the only way to obtain such a top-20 rank is to find out, for each player associated with a shot in the selection, their total number of shots and report the top-20 players found. This computation takes time proportional to at least the number of players associated with the shots in the selection. While this computation in the case of NBA shots is not very expensive (only a few thousand players ever played in the NBA), there are interesting use cases, analogous to the *player-shot* case, where the number of "players" can be quite large. For instance, *project-commit* in GitHub (a cloud based project repository), *tag-photo* in Flickr (a cloud based photo repository), or *hashtag-post* in a microblog website. In these cases the number of projects, tags, and hashtags are counted in millions instead of in thousands. The need to scan millions of objects to solve a single top-k query can be a hard hit in the latency budget of a fluid interactive experience.

TopKube is a data structure similar to a Nanocube: it encodes a *measure* in a multi-dimensional binning model, and, with this encoding, it allows the quick access of the measure's value of any product-bin in the model. The main addition of a TopKube when compared to a Nanocube is that, in

order to speed up top-k queries on one of its dimensions (e.g. top players by number of shots, top projects by number of commits), a TOPKUBE also includes ranking information in the encoding of that dimension.

The special dimension in a TOPKUBE is one that could be modeled as yet another 1-level bin hierarchy, but that contains lots of bins (e.g. players in the NBA example, or projects in GitHub, or tags in Flickr) and that we are interested in quickly accessing the top valued bins from this dimension with respect to the additive measure of interest on any multi-dimensional selection. We refer to this special dimension of a TOPKUBE as its *key dimension*, and the bins in this dimension as *keys*. Note that efficiently retrieving ranks of top-k *keys* (and their respective values) for an arbitrary selection of product-bins is the main goal of our TOPKUBE data structure. All dimensions in a TOPKUBE, except for its *key dimension*, are represented in the same way as the (non-special) dimensions of a Nanocube: as nested bin-hierarchies. Nodes in the bin-hierarchy of a previous dimension point to a root bin of a bin-hierarchy in the next dimension until we get to the last special dimension (see Figure 2 of [5]). A path through the nested hierarchies down to the last and special dimension of a TOPKUBE corresponds to a product-bin β on all dimensions except the key dimension.

To represent the *key dimension* information associated with a product-bin β, TOPKUBE uses the following data:

$$\beta \mapsto \left\{q, v, \sigma, \sum v_i\right\}, \tag{2}$$

where $q = q_1 \dots q_p$, $v = v_1 \dots v_p$, and $\sigma = \sigma_1 \dots \sigma_p$ are arrays of equal length obeying the following semantics: q_i is the i-th smallest key that appears in β; v_i is the value of the measure of interest (e.g. occurrences) for key q_i in β; and σ_i represents index of the key with the i-th largest value in β. For example, the third highest values key in a specific β is given v_{σ_3} and corresponds to key q_{σ_3}. In addition to arrays q, v, σ, in order to quickly solve queries that contain no key constraints, we also store the measure of all records in β regardless of keys, i.e. $\mu(A(\beta))$. Since in all our applications we always assume linearity of our measures, this aggregate reduces to the sum of the values v in β.

In Figure 3, we show a concrete TOPKUBE corresponding to the model shown on the top left part of the display. This TOPKUBE consists of one spatial dimension (two level quad-tree hierarchy) and a key dimension. In this toy example, the keys of the key dimension are the letters A, B, and C and the measure is simply the number of occurrences of a letter in the corresponding product-bin. Note that since there is only one dimension outside of the key dimension in this example, a product-bin β corresponds exactly to one spatial bin. The TOPKUBE data structure with the keys, counts, rank and total count are shown as tables in the bottom part of the figure. Note, for example, that the top valued key in the whole model is given by $q_{\sigma_1} = \texttt{C}$ and $v_{\sigma_1} = 6$ in the right-most table which corresponds to the coarsest spatial bin.

With this encoding for the key dimension information of a product-bin, to find out if a given key exists in a product-bin, we can perform a binary search in the q array (logarithmic time in the length of the array), and to access the i-th top ranked key we perform two random accesses: first we retrieve σ_i and then q_{σ_i} or v_{σ_i} (both constant time).

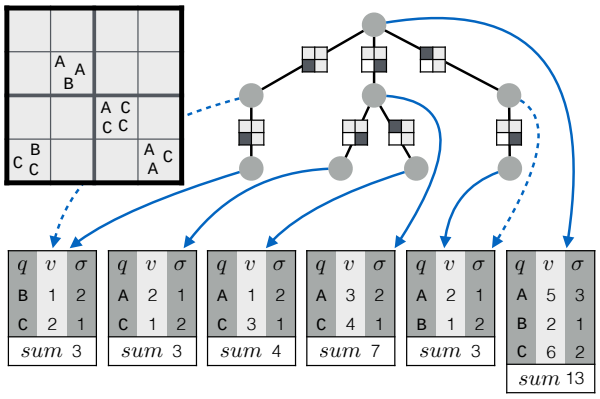


Fig. 3. Concrete example of a TOPKUBE with one spatial dimension and the special key-dimension for counting and ranking the event types: A, B, or C. The additional ranking information $(q, v, sigma)$ from Equation 2 is shown in the tables associated with each product-bin.

As in a Nanocube, the size of a TOPKUBE is proportional to its number of product-bins β plus the size of the encodings of the special dimension information associated with each of its product-bins. In the case of a Nanocube, this extra size per product-bin is the size of the summed area data from Equation 1, while in the case of TOPKUBE, it is given by the size of the rank aware data-structure of Equation 2. Note that if a Nanocube and a TOPKUBE have the same set of product-bins β and the number of time stamps and keys encoded in their respective special dimensions are comparable, the extra size cost of a TOPKUBE compared to the similar Nanocube will be the rank arrays σ. This extra size cost of a TOPKUBE represents a good trade-off if queries for interactive top-k keys are important for a given application. Another important remark with respect to the sizes of Nanocubes and TOPKUBES is that in order to represent a Nanocube special temporal dimension into a TOPKUBE dimension, we have to convert it into a conventional TOPKUBE dimension (e.g. a binary tree where the leaves are timestamps: right side of Figure 2). This adds a multiplicative logarithmic term to the size of that dimension: while $O(n)$ in a Nanocube, it becomes $O(n \log n)$ in a TOPKUBE. The advantage here is that now multiple temporal dimensions can be supported.

5.1 Top-K from Ranked Lists

The easiest top-k query for a TOPKUBE happens when a single product-bin β in involved. Suppose a user wants the top ranked keys in a multi-dimensional selection without any constraints. This query boils down to the single coarsest product-bin β in the cube (formed by root bins in all dimensions). In this case, obtaining the top-k keys is the same as generating from β the list $(q_{\sigma_1}, v_{\sigma_1}), \dots, (q_{\sigma_k}, v_{\sigma_k})$, and, clearly, it can be done in $O(k)$ steps. In general, though, this task is not that easy. The number of product-bins involved in the answer of a multi-dimensional selection is not one. For common spatial brushes, time intervals, categorical selections, the typical number of product-bins involved in a query ranges from tens up to a few thousand. For example, in Figure 2, we show a 624 bin selection in space and 3 bins in time which potentially means a 1,872 product-bin selection. In this case, the pre-stored ranks, or σ, we have for each

```
1: function SWEEP(L, k)
2:     h ← [ ]
3:     for i = 1 to LENGTH(L) do
4:         if LENGTH(L[i]) > 0 then
5:             PUSHHEAP(h, (L[i], 1), LESSTHANID)
6:     r ← [ ]
7:     key ← null
8:     sum ← 0
9:     while LENGTH(h) > 0 do
10:        (ℓ, i) ← POPHEAP(h, LESSTHANID)
11:        if key ≠ ℓ[i].key then
12:            INSERTK(r,k,key,sum)
13:            key ← ℓ[i].key
14:            sum ← ℓ[i].value
15:        else
16:            sum ← sum + ℓ[i].value
17:        if i < LENGTH(ℓ) then
18:            PUSHHEAP(h, (ℓ, i + 1), LESSTHANID)
19:    INSERTK(r,k,key,sum)
20:    return r

21: procedure INSERTK(r,k,key,sum)
22:    if key ≠ null then
23:        if LENGTH(r) < k then
24:            PUSHHEAP(r, (key, sum), LESSTHANSUM)
25:        else if r[1].value < sum then
26:            POPHEAP(r, LESSTHANSUM)
27:            PUSHHEAP(r, (key, sum), LESSTHANSUM)

28: function LESSTHANID((ℓ1, i1), (ℓ2, i2))
29:    return ℓ1[i1].key < ℓ2[i2].key

30: function LESSTHANSUM((key1, sum1), (key2, sum2))
31:    return sum1 < sum2
```

Fig. 4. Pseudo-code for the SWEEP ALGORITHM.

```
1: function TA(L, k)
2:     queues ← [ ]
3:     max ← 0
4:     for i = 1 to LENGTH(L) do
5:         if LENGTH(L[i]) > 0 then
6:             PUSHBACK(queues, (L[i], 1))
7:             max ← max + VALUEBYRANK(L[i], 1)
8:     r ← [ ]
9:     processed ← ∅
10:    i ← 1
11:    while i ≤LENGTH(queues) do
12:        (ℓ, rank) ← queues[i]
13:        (key, value) ← VALUEBYRANK(ℓ,rank)
14:        if rank <LENGTH(ℓ)
15:            queues[i] ← (ℓ, rank + 1)
16:            max ← max+VALUEBYRANK(ℓ, rank + 1)
17:            i ← i + 1
18:        else
19:            SWAPBACK(queues,i)
20:            POPBACK(queues)
21:        max ← max − value
22:        if key ∉ processed then
23:            for j = 1 to LENGTH(queues) do
24:                (ℓj, _) ← queues[j]
25:                if ℓ ≠ ℓj
26:                    value ← value+VALUEBYKEY(ℓj,key)
27:            INSERTK(r,k,key,value)
28:            if LENGTH(r)=k and max ≤ r[1].value then
29:                break
30:            processed ← processed ∪ {key}
31:        if i > LENGTH(queues) then
32:            i ← 1
33:    return r
```

Fig. 5. Pseudo-code for the THRESHOLD ALGORITHM. VALUEBYRANK uses σ to retrieve the p-th largest value of $\mathcal{L}[i]$ in constant time. VALUEBYKEY runs a binary search to access the value of a given key.

product-bin should help speeding up the top-k query, but is not as trivial as collecting top-k keys and values in $O(k)$ steps. Due to the lack of a consistent name in the literature, we refer to this problem as *Top-k from Ranked Lists* or TKR.

5.2 Sweep Algorithm

Let us step back a bit. Suppose we do not store the ranking information, σ, in each product-bin. In other words, if we go back to a rank-unaware data structure, how can we solve the top-k keys problem? One way, which we refer to as the NAIVE ALGORITHM, is to traverse the key and value arrays (q and v in Eq. 2) of all the product-bins in the selection, and keep updating a dictionary structure of key-value pairs. We would increment the value of a key q_i already in the dictionary with the current value v_i found for that key in the current product-bin. Once we finish traversing all product-bins, we would sort the keys by their values and report the top-k ones. The NAIVE ALGORITHM is correct, but inefficient. It uses memory proportional to all the keys present in all lists of all product-bins in the selection, and this number might be much larger than k.

A more efficient way to find the top-k keys in the union of multiple product-bins $\beta_1 \dots \beta_m$ is shown in the pseudo-code listed in Figure 4. Assume in the pseudo-code descriptions that $\mathcal{L}$ is a list of m key-value-rank data structures corresponding to Eq. 2 of the m input product-bins. The idea is to create, from $\mathcal{L}$, a heap/priority queue where the product-bin with a current smallest key is on the top of the heap (Lines 3-5). If we keep popping the next smallest key (Line 10) and its value from all the lists, we will sweep all key-value pairs in key increasing order, and every time we get a larger key (Line 11), we can be sure that the total measure of the previous key was complete. Using this approach, we can maintain a result buffer of size k (Line 23) instead of a dictionary with all keys in the all lists. We will refer to this approach as the SWEEP ALGORITHM. Note that this algorithm scans all keys of the product-bins β in the selection, as does the NAIVE ALGORITHM, but it does not need a potentially large buffer to solve the top-k problem. If we assume N is the sum of the number of keys in each input product-bin, it is easy to see that the worst case complexity of the SWEEP ALGORITHM is $O(m \log m + N \log k + N \log m)$.

Although the top-k problem was not discussed in the original Nanocubes paper [5], the SWEEP ALGORITHM can also be used to solve top-k queries there. Note also that SWEEP ALGORITHM is an natural way to solve unique count queries in both TOPKUBE and Nanocubes (how many unique keys are present in multi-dimensional selection).

5.3 Threshold Algorithm

The idea for adding the ranking information, σ, into a TOPKUBE is that it can potentially reduce the number of steps needed to find the top-k keys compared to the number of steps SWEEP ALGORITHM does. Instead of scanning all entries in all product-bins in our selection in key order, we would like to use the ranking information to scan first those keys with a higher chance of being in the top-k keys (i.e. larger partial values). The hope is that, by using such a strategy, a small partial scan and some bookkeeping would be enough to identify the top-k keys without a full scan.

```
1: function HYBRID(L, k, θ)
2:     SORT(L, LESSTHANLENGTH)
3:     m ←LENGTH(L)
4:     n ←LENGTH(L[m])
5:     entries ← n
6:     i_split ← m
7:     for i = m − 1 downto 1 do
8:         entries ← entries+ LENGTH(L[i])
9:         θ_i ← entries/((m − i + 1) ∗ n)
10:        if θ_i < θ then
11:            break
12:        else
13:            i_split ← i
14:    if i_split = 1 then
15:        return TA(L, k)
16:    else if i_split = m then
17:        return SWEEP(L, k)
18:    else
19:        aux ← SWEEP([L[1], ..., L[i_split]],∞)
20:        aux ← MAKERANK(aux)
21:        return TA([aux, L[i_split + 1], ..., L[m]],k)
```

Fig. 6. Pseudo-code for the HYBRID ALGORITHM.

In fact, this outline of an algorithm is well-known in the computer science community (see pseudo-code in Figure 5): the famous THRESHOLD ALGORITHM (TA) was described in [25]. Furthermore, TA was proven to be optimal in a strong sense: no other algorithm can access less data than it does and still obtain the correct answer. It essentially rotates through all ranked input lists (loop on Line 11) popping the highest value key in each of the lists (Line 13). For each popped key q_i, it goes through all the other lists (Loop on Line 23) binary searching for other key bins containing key q_i, once the aggregated value of q_i is found, it inserts q_i and its final value into a running top-k result set (Line 27). Once the algorithm figures out that no other key can have a higher value than the current top-k keys, the computation is done (Line 29). The worst case complexity of the THRESHOLD ALGORITHM is given by $O(m+Nm+N\log k)$. Note that the Nm term dominates this complexity and makes this worse than the one for SWEEP ALGORITHM.

5.4 Hybrid Algorithm

Although THRESHOLD ALGORITHM has the theoretical guarantees one would want, in practice we have observed that the instances of the TKR problem that show up in our use cases do not have the same characteristics as assumed in the explanations of TA that we reviewed. In those explanations, there was an implicit assumption that all m input lists in $\mathcal{L}$ contained the same set of keys. This is a natural assumption given the implicit application usually associated with TA: the m lists corresponded to m attribute-columns of a table with (mostly) non-zero entries. However, the instances of the TKR problem we observed in our use cases were sparse: one key q_i is present in only a small fraction of the m lists in the query selection. This sparseness introduces a wasteful step in the THRESHOLD ALGORITHM: the loop on Line 23. Most of the binary searches in that loop will fail to find the key that we are searching for. A typical case we see in our instances of the TKR problem is that on average each key is present in less than 3% of the m lists in our query selections.

While the THRESHOLD ALGORITHM can have wasted cycles trying to access entries for keys that do not exist, the SWEEP ALGORITHM wastes no such cycles: all entries it accesses are present in the input lists. In order to get the best results in our experiments, we combined the strengths of TA (early termination using rank information) and the SWEEP ALGORITHM (no wasted accesses on sparse instances) into a combined algorithm: the HYBRID ALGORITHM (pseudo-code shown in Figure 6).

The idea of the HYBRID ALGORITHM is simple: (1) raise the density of the input problem by running the SWEEP ALGORITHM on the smallest (easiest) input lists, and (2) run the THRESHOLD ALGORITHM on this denser equivalent problem. To make things more concrete, think about the 624 spatial bins in Figure 2. We typically expect the smaller squares in that spatial selection to have less data than larger squares. The idea would be to merge all the lists of the smaller squares to make the problem instance dense for the THRESHOLD ALGORITHM. We formalize this process below.

The input parameter θ in the HYBRID ALGORITHM controls the *density level* of the input problem in order to be considered ready for the THRESHOLD ALGORITHM. In other words, if, based on θ, our original input problem is too sparse for the THRESHOLD ALGORITHM, we would like to merge the smaller i_{split} lists using the SWEEP ALGORITHM (Line 19 of the HYBRID ALGORITHM), and then run an equivalent input denser problem through the THRESHOLD ALGORITHM (Line 21 of the HYBRID ALGORITHM). We define the *density* of a TKR instance to be N (i.e. sum of length of the q arrays in all product-bins of the selection) number of entries in all input lists, divided by the actual number of distinct keys (if we do the union of all m sets of keys) multiplied by m (the length of $\mathcal{L}$). Obviously, to compute this density one needs to find the number of keys after merging all the keys in arrays q for all m lists, which is an inefficient process that requires scanning all entries in all lists. To keep things computationally simple, and still correlated with the density notion, we define the *density level* θ as an upper bound for the actual density where we replace number of keys in the union of all m arrays by the length of the largest array q from the m product-bins (Lines 7-13).

6 EXPERIMENTAL RESULTS

Our system was implemented using a distributed client-server architecture. The TOPKUBE server program was implemented in C++ and provides a query API through HTTP. This enables flexibility: it can serve various client types ranging form desktop to mobile applications. All rendering in this work was done on client programs. We implemented a portable browser based client for using HTML5, D3, and WebGL as well as an OpenGL based C++ client for more native performance. The timing experiments relative to back-end performance in this paper ran on a 64 core AMD Opteron with 2.3 GHZ CPU and 512 GB of RAM.

The datasets used in our case studies are freely available and allow the extraction of geotagged keywords: articles on Wikipedia, tags on Flickr, projects on GitHub, and hashtags on Microblogs. The number of records (in millions) for these four datasets are respectively: 112M, 84M, 58M, and 40M. The number of keywords (in millions) are respectively: 3.0M, 1.6M, 1.5M, and 4.7M; thus although Wikipedia has the most records, Microblogs has the most unique keywords. The

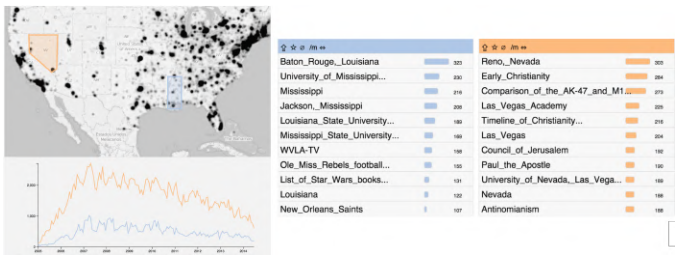


Fig. 7. Comparing the top edited articles in Nevada and Mississippi.

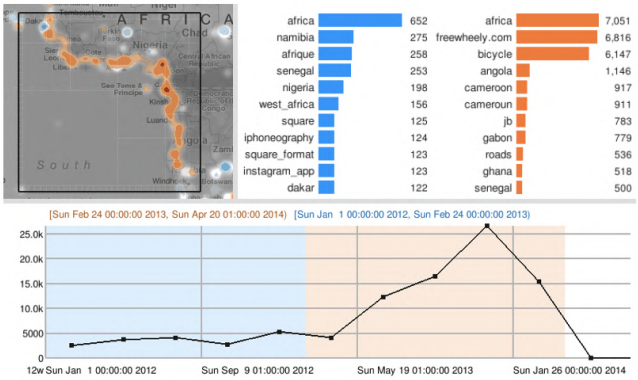


Fig. 8. Geolocated Flickr tags in Africa: the unusual activity on the west coast are from photos taken during a bike trip.

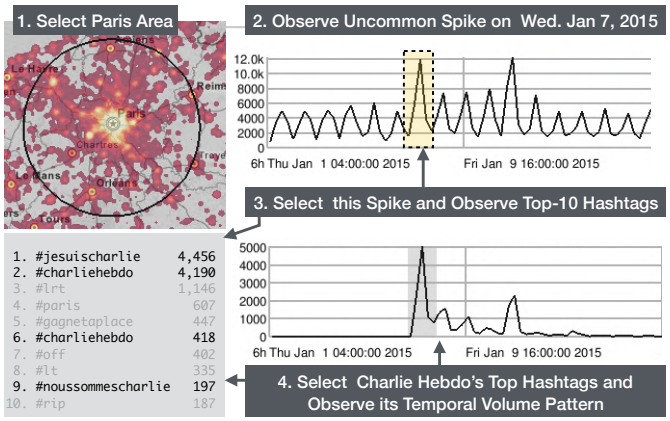


Fig. 9. Microblog exploration using TopKube: a temporal perspective of the top hashtags related to the the Charlie Hebdo terrorism act in Paris.

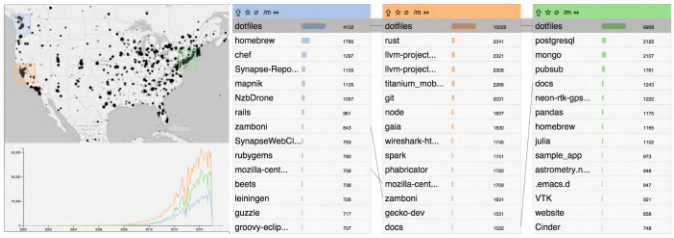


Fig. 10. GitHub projects with most commits in three large urban centers.

keywords were then used as *keys* in the construction of our TopKube data structure.

6.1 Use Cases

Wikipedia: The Wikipedia English dump datasets [33] contains edit history for every article since its creation in 2005. Anonymous edits contain the IP information of the user, which we used to trace their location. The final dataset, with geographical information, contains more than 112 million edits of over three million articles. Figure 7 presents a visualization of the dataset using TopKube. It is interesting to see that even though Nevada is not considered a state with a high percentage of religious people, religious articles are among the highest ranked. On the other hand, Mississippi, considered one of the most religious states in the U.S., does not have a single article related to religion among the top-20.

Flickr: The Yahoo! Flickr Creative Commons dataset [34] contains 100 million public Flickr photos and videos, with each record containing a set of user tags and geographical information. The dataset contains 84 million geolocated tags (1.57 million unique ones). Figure 8 shows how exploration can be used to gain insight of unusual patterns in the data along the West Coast of Africa. By highlighting the region, we can see that there were an unusual spike of activity during a few days in January. We create two different brushes in the timeseries: a blue one covering the low activity days, and an orange one covering the high activity days. We can see that the high activity spike is mostly due to photos tagged with *freewheely.com* and *bicycle*, which were taken by a Flickr user during his bike trip.

Microblogging: This dataset is comprised of publicly available geotagged microblog entries. From each post, we extracted the latitude, longitude, and hashtags from the blog. We can use TopKube to explore the most popular hashtags in order to understand how trending topics vary over time and in a given region. Figure 9 presents a sequence of exploration steps within January 2015 records. First we select a geographical area around Paris and find out an unusual Wednesday peak (Jan. 7) in the volume of hashtags. By selecting this peak we quickly find evidence of the event that caused the volume spike by inspecting the top-10 hashtags in the current selection (i.e. Paris and Jan 7). The event in question was the terrorism attack at the Charlie Hebdo headquarters. To understand how the hashtags created for this event at the day of the attack faded in time, we further constrain our selection to just the hashtags related to the terrorism attack and see that those fade almost completely (relative to event day) after one week of the attack.

GitHub: The GitHub dataset was first made available by Gousios [35] and contains all events from the GitHub public event time line. We were able to obtain information on more than 58 million commits for roughly 1.5 million projects. Each commit was geolocated based on the location of the user responsible for the action. Figure 10 presents a visualization with the top-k projects of three large urban centers. The only common project among all three regions is *dotfiles*, a project for sharing customized environment files on Unix-based operating systems. It is also interesting to notice how *llvm* and related projects (such as *clang*), are very popular in California, but not elsewhere. This shows a highly diversified open source community across the United States.

6.2 Performance

To determine which of the previously described algorithms works best when solving top-k queries, we conducted an initial evaluation using the Microblogs dataset, which is the

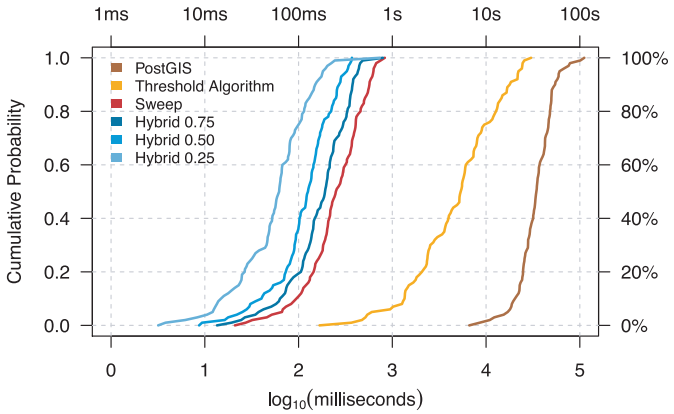


Fig. 11. Empirical cumulative distributions of the time to retrieve the top-32 valued keys for 100 spatiotemporal queries on the microblog dataset. Speedup potential of Hybrid versus Sweep, Threshold, and PostGIS.

most challenging because it has the most keys (4.7M). The first experiment consisted of collecting 100 spatiotemporal selections ranging from large geospatial areas (continents) to smaller regions (cities) combined with time interval selections ranging from multiple weeks to a few hours. Next, we retrieved the top-32 valued keys in each of the 100 selections with the different methods we describe in Section 5. In addition to SWEEP, THRESHOLD, and HYBRID, we also included PostGIS in this experiment. PostGIS is the most popular open source GIS package that can solve the problem that we were targeting in this work. It is the de facto spatial database in our opinion, which is why we chose to compare our techniques to it. We configured PostGIS according to its official documentation for a dataset containing key, latitude, longitude, and timestamp.

In Figure 11 we present the results of our first experiment in the form of cumulative distributions: what percentage of the 100 spatiotemporal queries we could retrieve the top-32 keys in less than t time units. All results were exactly the same for all the methods tested including PostGIS. We are able to see that the HYBRID ALGORITHM with varying θ thresholds had query times consistently smaller than both TA and SWEEP. This fact confirmed our hypothesis that we can accelerate top-k queries by adding rank information to the index. Although this fact seems obvious, this study shows that a natural use of rank information as done by TA does not yield a speedup. Only a combination of the strengths of TA and SWEEP illustrated by the HYBRID approach gave the speedup we expected. It is worth noting, however, that there was a steep increase in query times for HYBRID on the most difficult problems (as cumulative probability approached 1), which suggests that a better balance between SWEEP and TA was possible. In Section 7 we perform a more thorough experiment to understand the behavior of our top-k methods.

7 TOPKUBE-BENCHMARK

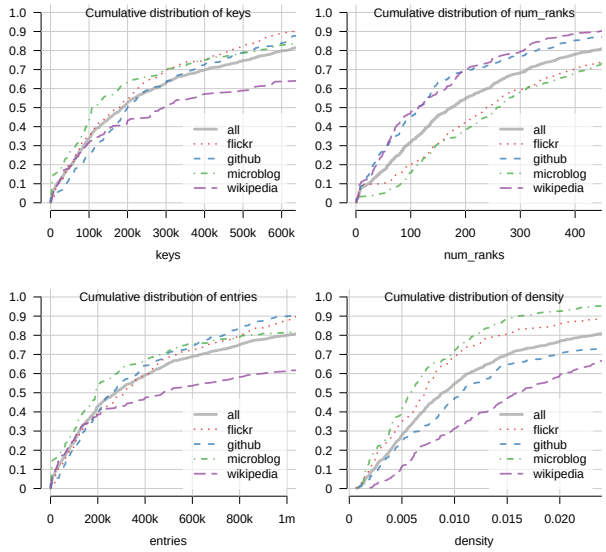
As illustrated in the previous examples, the main use case that drove the development of TOPKUBE was to provide an interactive visualization front-end to quickly access top-k "terms" for arbitrary spatiotemporal selections. Although we observe significant speedups using the HYBRID ALGORITHM (e.g. $\theta = 0.25$ in Figure 11) compared to other techniques, we believe in further improvements. To assess how different top-k algorithms (the ones shown here and future ones) perform in rank merging problems on datasets similar to the ones we collected for this work, we created the TOPKUBE-BENCHMARK and made it publicly available: `github.com/laurolins/topkube_benchmark`.

7.1 Benchmark Characteristics

The TOPKUBE-BENCHMARK consists of one thousand TKR problems. Each problem consists of a list of *ranks*, where each rank is defined by a list of key-value pairs and the associated ordering information, σ, as shown in Equation 2. The goal is to, given a value k, find the top-k keys and their aggregated value from a consolidated rank of the multiple input ranks (note that this problem does not require explicitly finding the total consolidated rank). Each of the four datasets (i.e. Flickr, GitHub, Microblog, and Wikipedia) contributed equally with two hundred and fifty problems for the the TOPKUBE-BENCHMARK. These problems were collected during interactive exploratory sessions using these four datasets In Figure 12, we present the distribution of four characteristics of the problems in the benchmark: (1) number of keys; (2) number of ranks; (3) number of entries; and (4) density. The number of keys of a problem is simply the union of the keys present in each rank. The number of ranks is the number of lists (of key values) from the selection. The number of entries is the sum of the sizes of the ranks (i.e. the total number of keys in all ranks). Note that number of entries should be larger than the number of keys since the same key is usually present in more than one rank. Finally, the density is simply the number of entries divided by the product of number of ranks and number of keys. If a problem has density one, each key is present in all ranks.

If we follow the overall thick solid gray line in the keys plot (Figure 12, top left), we notice that fewer than 40% of the problems involved fewer than 100k keys, which means that most problems (more than 60%) involved 100k keys or more. If we check the table entry in row *keys/all* and column *90%* from the table in that figure, we see that more than 10% of the problems involved 1.1 million keys or more. So, given that these problem instances were collected from natural visual interactions with the data, it is clear that large TKR problems can show up at exploration time: a challenging problem for interactivity. In terms of the number of ranks, we see that more than 50% of the problems have 170 or more ranks to be processed (row *num_ranks/all*, column *50%*), and in 10% of the cases we had 860 ranks or more (lots of non-empty product-bins being hit by the multi-dimensional selection). In terms of number of entries, we see that 20% of the problems had more than 1 million entries (row *entries/all*, column *80%*). Perhaps the most important observation of the problems in the TOPKUBE-BENCHMARK comes from their density: the problems are really sparse (worst-case scenario for TA). If we consider 100 ranks in a problem and a density of 0.051 (90% of the problems have density 0.051 or less: see row *density/all*, column *90%*), on average we will have one key present in only 5.1 of the 100 ranks. These real-world, interactive explorations clearly demonstrate the sparsity of our inputs to the TKR problem, and that the binary searches on Line 23 in TA are largely wasted effort.

From the characteristics of the four datasets, we know that spatially the Microblog and Flickr datasets involve more



var.	dataset	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
keys	all	7.0E+00	1.8E+04	5.5E+04	8.7E+04	1.3E+05	1.9E+05	2.7E+05	4.0E+05	6.1E+05	1.1E+06	4.7E+06
	flickr	1.4E+03	2.2E+04	5.4E+04	7.9E+04	1.2E+05	1.8E+05	2.3E+05	3.0E+05	4.7E+05	6.3E+05	1.6E+06
	github	7.0E+00	4.2E+04	7.9E+04	1.2E+05	1.7E+05	2.0E+05	2.6E+05	3.7E+05	5.2E+05	7.1E+05	1.5E+06
	microblog	4.2E+02	4.1E+03	3.4E+04	6.6E+04	9.3E+04	1.1E+05	1.8E+05	3.0E+05	5.4E+05	1.1E+06	4.7E+06
	wikipedia	1.5E+03	2.1E+04	5.7E+04	9.4E+04	1.6E+05	3.0E+05	5.2E+05	9.4E+05	1.2E+06	1.7E+06	3.0E+06
num_ranks	all	1.0E+00	2.4E+01	6.0E+01	9.2E+01	1.3E+02	1.7E+02	2.4E+02	3.1E+02	4.3E+02	8.6E+02	4.3E+03
	flickr	1.0E+00	5.3E+01	1.0E+02	1.4E+02	1.9E+02	2.4E+02	3.1E+02	4.0E+02	6.7E+02	1.1E+03	4.3E+03
	github	1.0E+00	1.5E+01	3.2E+01	5.8E+01	8.5E+01	1.2E+02	1.4E+02	2.1E+02	3.3E+02	5.4E+02	2.6E+03
	microblog	1.0E+00	8.4E+01	1.2E+02	1.5E+02	2.1E+02	2.6E+02	3.1E+02	4.3E+02	6.4E+02	1.1E+03	2.5E+03
	wikipedia	1.0E+00	8.9E+00	3.8E+01	5.7E+01	7.4E+01	1.1E+02	1.6E+02	2.0E+02	3.1E+02	4.4E+02	3.0E+03
entries	all	9.0E+00	2.5E+04	7.4E+04	1.2E+05	1.9E+05	2.7E+05	4.2E+05	6.4E+05	1.0E+06	2.1E+06	2.0E+07
	flickr	1.9E+03	3.3E+04	8.8E+04	1.3E+05	2.0E+05	3.1E+05	4.2E+05	5.4E+05	8.0E+05	1.0E+06	2.0E+06
	github	9.0E+00	5.0E+04	9.9E+04	1.5E+05	2.0E+05	2.7E+05	3.7E+05	5.0E+05	7.3E+05	1.0E+06	2.3E+06
	microblog	5.4E+02	5.5E+03	5.0E+04	1.0E+05	1.4E+05	1.9E+05	2.8E+05	4.8E+05	8.4E+05	1.7E+06	7.0E+06
	wikipedia	1.6E+03	2.4E+04	7.0E+04	1.2E+05	2.2E+05	4.8E+05	9.0E+05	2.2E+06	3.2E+06	4.8E+06	2.0E+07
density	all	6.9E-04	2.3E-03	3.9E-03	5.4E-03	7.0E-03	8.9E-03	1.1E-02	1.5E-02	2.3E-02	5.1E-02	1.0E+00
	flickr	6.9E-04	2.0E-03	2.9E-03	4.4E-03	6.1E-03	7.2E-03	8.2E-03	1.0E-02	1.4E-02	2.8E-02	1.0E+00
	github	9.1E-04	2.8E-03	4.3E-03	6.8E-03	8.7E-03	1.1E-02	1.4E-02	2.0E-02	3.4E-02	8.0E-02	1.0E+00
	microblog	7.7E-04	1.6E-03	2.6E-03	3.8E-03	5.0E-03	5.9E-03	7.1E-03	9.7E-03	1.2E-02	1.6E-02	1.0E+00
	wikipedia	2.0E-03	4.9E-03	6.5E-03	9.8E-03	1.3E-02	1.6E-02	2.1E-02	2.8E-02	3.7E-02	2.2E-01	1.0E+00

Fig. 12. Characteristics of the TKR problems in the TOPKUBE-BENCHMARK. Left: we plot of the cumulative distributions for the number of keys, ranks, entries, and density up to the 0.8 quantile (or 80th percentile). Right: we list the actual values of the percentiles for these distributions.

spatial bins than the Wikipedia and GitHub datasets. The reason for this is simply that both Wikipedia and GitHub datasets were obtained by geocoding the IP address of the device associated with an article edit or project commit which induces a more constrained set of locations when compared to GPS locates from devices used for posts in Flickr and a Microblog service. This fact explains the distribution shown in the number of ranks plot in Figure 12: more product-bins are involved in the spatiotemporal selections for Flickr and Microblogs than Wikipedia and GitHub. Given the similar sparse nature of the TKR problems that we see on these four datasets (see x-axis of the density plot in Figure 12), we focus the rest of our analysis here on the entire set of one thousand problems without splitting them by source.

θ \ %	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
TA	1.32E−03	5.73E−03	1.70E−02	3.94E−02	7.75E−02	1.48E−01	3.02E−01	7.21E−01	2.43E+00	1.68E+02
0.05	1.25E−03	4.66E−03	1.17E−02	2.59E−02	5.04E−02	9.57E−02	1.83E−01	4.14E−01	1.22E+00	6.71E+01
0.10	1.08E−03	3.80E−03	8.34E−03	1.64E−02	3.03E−02	5.48E−02	1.04E−01	2.02E−01	5.48E−01	2.34E+01
0.15	1.00E−03	3.38E−03	7.36E−03	1.41E−02	2.53E−02	4.42E−02	7.65E−02	1.43E−01	3.51E−01	1.06E+01
0.20	9.79E−04	3.40E−03	7.31E−03	1.41E−02	2.37E−02	4.18E−02	7.20E−02	1.29E−01	2.97E−01	6.43E+00
0.25	9.80E−04	3.48E−03	7.63E−03	1.43E−02	2.44E−02	4.15E−02	7.22E−02	1.29E−01	3.00E−01	4.76E+00
0.30	9.85E−04	3.69E−03	8.21E−03	1.54E−02	2.63E−02	4.41E−02	7.36E−02	1.33E−01	3.19E−01	4.15E+00
0.35	1.10E−03	3.92E−03	9.19E−03	1.67E−02	2.89E−02	4.81E−02	7.76E−02	1.47E−01	3.53E−01	4.07E+00
0.40	1.23E−03	4.41E−03	9.95E−03	1.88E−02	3.21E−02	5.38E−02	8.51E−02	1.65E−01	3.94E−01	4.24E+00
0.45	1.45E−03	4.99E−03	1.14E−02	2.13E−02	3.58E−02	5.86E−02	9.41E−02	1.80E−01	4.17E−01	4.51E+00
0.50	1.78E−03	5.65E−03	1.27E−02	2.38E−02	3.97E−02	6.54E−02	1.03E−01	1.96E−01	4.54E−01	4.83E+00
0.55	2.23E−03	6.94E−03	1.45E−02	2.61E−02	4.47E−02	7.03E−02	1.14E−01	2.08E−01	4.82E−01	5.09E+00
0.60	2.47E−03	7.89E−03	1.67E−02	2.83E−02	4.97E−02	7.90E−02	1.29E−01	2.26E−01	5.29E−01	5.40E+00
0.65	2.99E−03	8.94E−03	1.90E−02	3.23E−02	5.59E−02	8.68E−02	1.39E−01	2.42E−01	5.79E−01	5.72E+00
0.70	3.16E−03	1.00E−02	2.03E−02	3.53E−02	6.17E−02	9.58E−02	1.51E−01	2.67E−01	6.09E−01	6.08E+00
0.75	3.43E−03	1.07E−02	2.27E−02	4.01E−02	6.91E−02	1.06E−01	1.67E−01	2.94E−01	6.47E−01	6.21E+00
0.80	3.63E−03	1.20E−02	2.67E−02	4.55E−02	7.71E−02	1.16E−01	1.80E−01	3.10E−01	6.75E−01	6.48E+00
0.85	3.84E−03	1.41E−02	2.94E−02	5.11E−02	8.48E−02	1.25E−01	1.98E−01	3.22E−01	7.01E−01	6.47E+00
0.90	4.01E−03	1.45E−02	3.06E−02	5.38E−02	8.91E−02	1.36E−01	2.06E−01	3.38E−01	7.35E−01	6.64E+00
0.95	3.98E−03	1.51E−02	3.17E−02	5.53E−02	9.13E−02	1.41E−01	2.18E−01	3.48E−01	7.52E−01	6.64E+00
Sweep	7.66E−03	2.17E−02	3.73E−02	5.84E−02	8.58E−02	1.28E−01	1.91E−01	3.03E−01	6.21E−01	6.63E+00

0.5 ms. <	1 ms. <	5 ms. <	10 ms. <	50 ms. <	100 ms. <	500 ms. <	1 s. <	5 s. <	10 s.

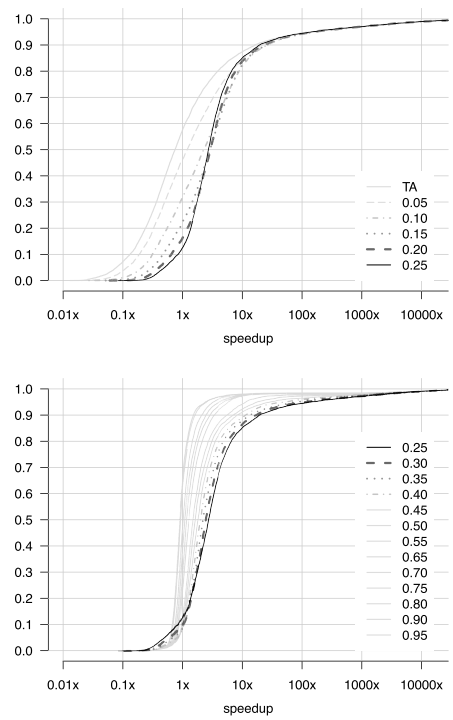
Fig. 13. Latency distribution (percentiles) of twenty one different strategies (1 x TA, 19 x Hybrid, 1 x Sweep) when solving the one thousand TOPKUBE-BENCHMARK problems for seven different values of $k = 5, 10, 20, 40, 80, 160, 320$. Darker blue shades indicate smaller latencies; darker red shades indicate larger latencies.

7.2 Benchmark Performance

To assess the performance of the SWEEP ALGORITHM, the THRESHOLD ALGORITHM, and the HYBRID ALGORITHM, we ran each of the algorithms on the one thousand problems of the TOPKUBE-BENCHMARK for $k = 5, 10, 20, 40, 80, 160, 320$, for a total of seven thousand runs for each algorithm. We ran the HYBRID ALGORITHM with the threshold θ varying from 0.05 to 0.95 by increments of 0.05. So, for each problem and each k, we ran the SWEEP ALGORITHM and the THRESHOLD ALGORITHM each once, and the HYBRID ALGORITHM nineteen times (one for each θ): a total of 21 different algorithmic recipes to find the top-k terms. The (percentiles of the) distributions of the latency (i.e. time to solve) for each of the seven thousand TKR instances by each of the 21 strategies are shown in Figure 13.

We assume that less than 0.1 seconds latency is the appropriate level for fluid interactivity: blue color tones in Figure 13 correspond to latencies that are less than 0.1 seconds, while red tones represent high latencies ($\geq$ 0.1s.). This table contains clear evidence that the HYBRID strategy can improve on the latency of the two extreme strategies: sweep (which consolidates a full rank before generating top-k) and the threshold strategy which chases a proof of the top-k terms directly from all ranks of the input problem. Note that for values of θ between 0.15 and 0.45, the shades of blue have the widest reach: 70% of the problems had less than 100 ms latency, while for the sweep strategy 40% of the problems had 128 ms latency or more (column 60%).

To more deeply explore the speedup of the HYBRID ALGORITHM over the SWEEP ALGORITHM on the TOPKUBE-BENCHMARK problems, we divide the query times (i.e. latency) for each problem instance i: $\text{speedup}_i^\theta = \text{sweep\_time}_i / \text{hybrid\_time}_i^\theta$. If it takes two times more for the sweep strategy to solve an instance compared to the hybrid strategy, we say there was $2\times$ speedup. On the other hand, if the sweep strategy takes half the time, we say there was $1/2\times$ speedup or, equivalently, a $2\times$ slow down.



θ \ %	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
TA	1.31E−01	2.38E−01	3.64E−01	5.16E−01	7.46E−01	1.11E+00	1.84E+00	3.90E+00	1.63E+01	1.52E+05
0.05	1.88E−01	3.27E−01	4.99E−01	7.58E−01	1.19E+00	1.92E+00	3.26E+00	6.60E+00	2.14E+01	1.43E+05
0.10	3.11E−01	5.52E−01	9.14E−01	1.46E+00	2.24E+00	3.47E+00	5.32E+00	8.81E+00	2.66E+01	1.52E+05
0.15	4.74E−01	8.88E−01	1.45E+00	2.11E+00	2.88E+00	3.95E+00	5.56E+00	8.62E+00	2.52E+01	1.37E+05
0.20	6.48E−01	1.21E+00	1.73E+00	2.22E+00	2.94E+00	3.74E+00	4.97E+00	7.66E+00	2.29E+01	1.37E+05
0.25	8.42E−01	1.37E+00	1.76E+00	2.21E+00	2.71E+00	3.34E+00	4.34E+00	6.59E+00	1.93E+01	1.43E+05
0.30	1.02E+00	1.40E+00	1.72E+00	2.08E+00	2.45E+00	2.99E+00	3.84E+00	5.73E+00	1.69E+01	1.52E+05
0.35	1.06E+00	1.36E+00	1.63E+00	1.89E+00	2.19E+00	2.65E+00	3.36E+00	4.83E+00	1.38E+01	1.43E+05
0.40	1.05E+00	1.30E+00	1.51E+00	1.72E+00	1.99E+00	2.34E+00	2.91E+00	4.17E+00	1.15E+01	1.37E+05
0.45	1.02E+00	1.22E+00	1.40E+00	1.58E+00	1.81E+00	2.08E+00	2.56E+00	3.53E+00	8.35E+00	1.37E+05
0.50	9.98E−01	1.15E+00	1.30E+00	1.44E+00	1.63E+00	1.87E+00	2.29E+00	3.08E+00	7.11E+00	1.37E+05
0.55	9.59E−01	1.09E+00	1.22E+00	1.34E+00	1.50E+00	1.72E+00	2.03E+00	2.71E+00	5.61E+00	1.52E+05
0.60	9.22E−01	1.04E+00	1.14E+00	1.25E+00	1.38E+00	1.57E+00	1.85E+00	2.34E+00	4.05E+00	1.43E+05
0.65	8.73E−01	9.76E−01	1.07E+00	1.16E+00	1.28E+00	1.42E+00	1.66E+00	2.14E+00	3.40E+00	1.35E+05
0.70	8.37E−01	9.29E−01	1.01E+00	1.09E+00	1.17E+00	1.30E+00	1.49E+00	1.83E+00	2.88E+00	1.35E+05
0.75	7.92E−01	8.79E−01	9.48E−01	1.01E+00	1.09E+00	1.20E+00	1.37E+00	1.68E+00	2.46E+00	1.04E+05
0.80	7.50E−01	8.30E−01	8.94E−01	9.55E−01	1.03E+00	1.12E+00	1.24E+00	1.45E+00	2.09E+00	9.34E+04
0.85	7.16E−01	7.94E−01	8.51E−01	9.05E−01	9.66E−01	1.04E+00	1.15E+00	1.33E+00	1.73E+00	1.35E+05
0.90	6.95E−01	7.62E−01	8.17E−01	8.69E−01	9.21E−01	9.86E−01	1.08E+00	1.24E+00	1.62E+00	1.29E+05
0.95	6.69E−01	7.39E−01	7.94E−01	8.42E−01	8.91E−01	9.51E−01	1.04E+00	1.19E+00	1.52E+00	1.43E+05

1/100x <	1/10x <	1/5x <	1/2x <	1/1.5x <	1/1.1x <	1.1x <	1.5x <	2x <	5x <	10x <	100x

Fig. 14. Speedup distribution of the Threshold and Hybrid algorithms ($0 < \theta < 1$) over Sweep algorithm for all problems in the TOPKUBE-BENCHMARK.

We consider SWEEP as the baseline approach, and we want to understand how THRESHOLD and HYBRID compare to it. Figure 14 shows the cumulative distribution of the speed-up (or slow-down if less than 1.0) for different runs of HYBRID with threshold θ varying by 0.05 from 0 to 0.95. Note that THRESHOLD is simply HYBRID with $\theta = 0$. Slow downs of 10% or more are colored in shades of red, and speed ups of 10% or more are colored in shades of blue. For $\theta = 0.25$, 80% of the problems had a speed up of 37% or more (column 20%); 70% of the instances had a speed up of 76% or more (column 30%); and 10% of the instances had a speedup of an order of magnitude (at least $19.3\times$).

It is also clear that our choice of θ impacts how well the HYBRID ALGORITHM performs. For $\theta = 0.95$, 50% of the latencies were 10% worse than the sweep strategy. In general, $\theta = 0.25$ performs very well overall and tends to be our default selection; however, $\theta = 0.20$ is arguably just as efficient and even beats $\theta = 0.25$ between 40% - 90%.

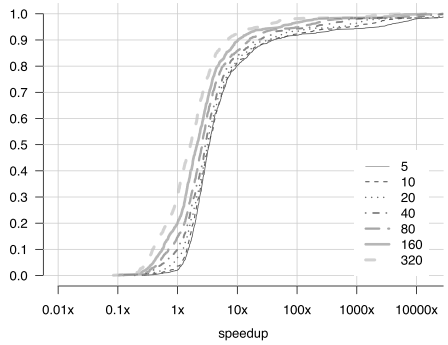
As can be seen in Figure 12, we explicitly included some extreme problem instances into TOPKUBE-BENCHMARK: problems with a single rank and very few entries/keys, or conversely thousands of ranks and millions of entries/keys. These problems show up in the 0 and 100 percentile columns of that table, as well as the 100 percentile column of the colored distribution tables shown. We do not place undue emphasis on those columns: a speed up of one hundred thousand times ($\theta = 0.25$ 100%) is not representative of the typical cases, while the speedups up to 90% are more typical.

7.3 Speedup Relative to k

In Figure 15, we present the speedup over SWEEP by HYBRID with $\theta = 0.25$, for the different values of k. The blue and red shading follows the same speedup encoding as in Figure 14. As expected, a higher value of k demands more computation to find the top-k terms in the consolidated rank. For $k = 5$, 90% of the instances were solved at least 52% faster than the sweep approach (row 5, column 10%); for $k <= 20$, 80% had a $2\times$ speedup; and for $k = 320$, 60% of the instances had a 24% speed up. We argue that a value of $k \leq 100$ is appropriate for exploration of top terms at interactive rates. Going deeper (i.e. larger k's) on an investigation could use more computational resources and a full sweep based consolidated rank could be used there.

7.4 Speedup Relative to Keys

We expect that as the TKR problems become larger (i.e. more keys), the speedups of our approach over the Sweep approach will increase. In the top plot of Figure 16, we show the distribution of the speedups of HYBRID with $\theta = 0.25$ on four equally sized groups of problems obtained from the TOPKUBE-BENCHMARK. Again we ran each of these problems with $k = 5, 10, 20, 40, 80, 160, 320$. The first group has the first quarter of problems with the fewest keys, the second group has the next quarter of problems with next fewest keys, and so on up to the fourth quarter of problems with the most keys. From the plot, we observe the expected pattern: as problems become harder, the distribution is shifted to the right of the cumulative distribution plot, indicating larger speedup values. For example, on the intersection of the vertical line at $1\times$ (same speed), we already have more than 30% of the problems in group one (the easiest group), while this number is around 5% for the problems in group 4 (the hardest group). This is encouraging and supports using TOPKUBE with HYBRID on even larger problems.



k \ %	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
5	1.52E+00	1.97E+00	2.34E+00	2.83E+00	3.40E+00	4.11E+00	5.67E+00	9.81E+00	4.07E+01	1.43E+05
10	1.44E+00	1.83E+00	2.27E+00	2.71E+00	3.34E+00	4.22E+00	5.57E+00	9.02E+00	3.74E+01	1.35E+05
20	1.24E+00	1.63E+00	2.09E+00	2.62E+00	3.16E+00	3.90E+00	5.15E+00	8.37E+00	3.18E+01	8.84E+04
40	1.01E+00	1.48E+00	1.90E+00	2.45E+00	2.93E+00	3.53E+00	4.63E+00	6.86E+00	2.06E+01	4.60E+04
80	7.30E−01	1.31E+00	1.68E+00	2.13E+00	2.58E+00	3.08E+00	3.90E+00	5.81E+00	1.61E+01	2.74E+04
160	5.54E−01	9.96E−01	1.36E+00	1.65E+00	2.13E+00	2.69E+00	3.39E+00	4.70E+00	1.04E+01	1.43E+04
320	3.86E−01	6.80E−01	9.73E−01	1.24E+00	1.64E+00	2.11E+00	2.65E+00	3.61E+00	6.72E+00	5.57E+03

1/100x <	1/10x <	1/5x <	1/2x <	1/1.5x <	1/1.1x <	1.1x <	1.5x <	2x <	5x <	10x <	100x

Fig. 15. Distribution of the speedup of the Hybrid Algorithm ($\theta = 0.25$) over the SWEEP ALGORITHM broken down by k.

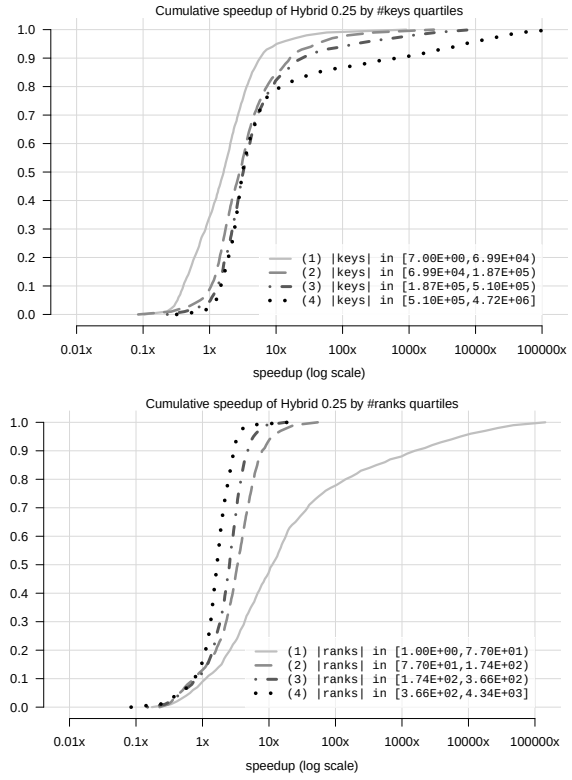


Fig. 16. Cumulative distribution of the speedup of HYBRID with $\theta = 0.25$ when partitioning the benchmark problems into four (equally sized) groups based on which quartile in the distribution of the number of keys (top) or ranks (bottom) each problem falls into. On top, the groups with greater number of keys (harder problems) observe greater speedups. On the bottom, groups with fewer ranks observer greater speedups.

7.5 Speedup Relative to Ranks

We partitioned the problems in TOPKUBE-BENCHMARK in four groups based on which quartile of the distribution of the number of ranks they fall into. In the bottom plot of Figure 16, we see an exact inversion of the pattern observed when we broke down the problems by the number of keys. The speedup is greater when fewer ranks are involved in a TKR problem. This is explained by the fact that more ranks yield larger values for i_{split} in the HYBRID ALGORITHM, and thus a longer sweep phase (Figure 6, Line 19).

8 POTENTIAL IMPROVEMENTS

With the availability of the TOPKUBE-BENCHMARK problems and the C++ reference implementation for the algorithms presented in this paper, we would like to motivate new studies to improve on our results. One issue with HYBRID is its dependency on the parameter θ. Although $\theta = 0.25$ works very well, we see in Figure 14 that is not the fastest in every case. This suggests the need for an adaptive way to find i_{split} in HYBRID that is not based simply on a fixed θ.

The main focus of this work has always been computing top-k queries interactively; thus TOPKUBE construction times and memory utilization were never optimized, but both can be greatly improved. The construction times (in hours) for the four datasets using a single CPU thread were respectively: 5.3h, 3.9h, 3.4h, and 1.7h. This yields insertion rates ranging from 4.7 to 6.5 thousand records per second. Preliminary experiments have shown that by using a multi-threaded build, these insertion rates can increase close to linearly with the number of threads. The memory (in gigabytes) used by TOPKUBE for the datasets was respectively: 114GB, 20GB, 14GB, and 53GB. These numbers are significant, but again can be greatly reduced with an optimized implementation. We note that by utilizing path compression on sparse hierarchies (i.e. deep hierarchies with few branches) we can reduce the reported memory usage by more than an order of magnitude.

9 CONCLUSION

As user-generated online data continues to grow at incredible rates, ranking objects and information has never played such an important role in understanding our culture and the world. Although previous techniques have been able to create such rankings, they are inefficient and unable to be used effectively during an interactive exploration of the ranked data. We have introduced TOPKUBE, an enhanced in-memory data cube that is able to generate ranked lists up to an order of magnitude faster than previous techniques. A careful evaluation of our techniques with public datasets has demonstrated its value. We have also made our benchmarks available for others to make direct comparisons to our work.

10 ACKNOWLEDGMENTS

This work was supported in part by the Moore-Sloan Data Science Environment at NYU, NYU Tandon School of Engineering, NYU Center for Urban Science and Progress, AT&T, IBM Faculty Award, NSF awards CNS-1229185, CCF-1533564 and CNS-1544753.

REFERENCES

[1] N. Hornby, *High fidelity*. Penguin UK, 2000.

[2] B. J. Faulk, "Love and lists in nick hornby's high fidelity," *Cultural Critique*, vol. 66, no. 1, pp. 153–176, 2007.

[3] T. Mostak, "An overview of mapd (massively parallel database)," in *White paper*. Massachusetts Institute of Technology, 2013.

[4] Z. Liu, B. Jiang, and J. Heer, "immens: Real-time visual querying of big data," *Computer Graphics Forum*, vol. 32, no. 3, pp. 421–430, 2013.

[5] L. Lins, J. T. Klosowski, and C. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *IEEE Trans. Vis. Comput. Graphics*, vol. 19, no. 12, pp. 2456–2465, Dec 2013.

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

[7] 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*, Oct 2013, pp. 25–32.

[8] A. Dix and G. Ellis, "By chance - enhancing interaction with large data sets through statistical sampling," in *Proceedings of the Working Conference on Advanced Visual Interfaces*. ACM, 2002, pp. 167–176.

[9] B. Shneiderman, "Dynamic queries for visual information seeking," *IEEE Software*, vol. 11, no. 6, pp. 70–77, Nov 1994.

[10] D. B. Carr, R. J. Littlefield, W. Nicholson, and J. Littlefield, "Scatterplot matrix techniques for large n," *Journal of the American Statistical Association*, vol. 82, no. 398, pp. 424–436, 1987.

[11] P. J. Rousseeuw and A. M. Leroy, *Robust regression and outlier detection*. John Wiley & Sons, 2005, vol. 589.

[12] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2122–2131, Dec 2014.

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

[14] 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*, vol. 1, no. 1, pp. 29–53, 1997.

[15] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *IEEE Trans. Vis. Comput. Graphics*, vol. 8, no. 1, pp. 52–65, Jan 2002.

[16] ——, "Multiscale visualization using data cubes," *IEEE Trans. Vis. Comput. Graphics*, vol. 9, no. 2, pp. 176–187, Apr 2003.

[17] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: an experimental evaluation," in *Proceedings of the 39th International Conference on Very Large Data Bases*. VLDB Endowment, 2013, pp. 217–228.

[18] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson, "Spatio-textual indexing for geographical search on the web," in *Advances in Spatial and Temporal Databases*. Springer, 2005, pp. 218–235.

[19] A. Cary, O. Wolfson, and N. Rishe, "Efficient and scalable method for processing top-k spatial boolean queries," in *Scientific and Statistical Database Management*. Springer, 2010, pp. 87–95.

[20] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-k spatial keyword queries," in *Advances in Spatial and Temporal Databases*. Springer, 2011, pp. 205–222.

[21] D. Zhang, K.-L. Tan, and A. K. Tung, "Scalable top-k spatial keyword search," in *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013, pp. 359–370.

[22] D. Xin, J. Han, H. Cheng, and X. Li, "Answering top-k queries with multi-dimensional selections: The ranking cube approach," in *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB Endowment, 2006, pp. 463–474.

[23] T. Wu, D. Xin, and J. Han, "Arcube: supporting ranking aggregate queries in partially materialized data cubes," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 79–92.

[24] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, p. 11, 2008.

[25] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 614–656, 2003.

[26] M. Shmueli-Scheuer, C. Li, Y. Mass, H. Roitman, R. Schenkel, and G. Weikum, "Best-effort top-k query processing under budgetary constraints," in *IEEE International Conference on Data Engineering 2009*. IEEE, 2009, pp. 928–939.

[27] N. Ferreira, L. Lins, D. Fink, S. Kelling, C. Wood, J. Freire, and C. Silva, "Birdvis: Visualizing and understanding bird populations," *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 12, pp. 2374–2383, Dec 2011.

[28] J. Wood, J. Dykes, A. Slingsby, and K. Clarke, "Interactive visual exploration of a large spatio-temporal dataset: reflections on a geovisualization mashup." *IEEE Trans. Vis. Comput. Graphics*, vol. 13, no. 6, pp. 1176–1183, Nov 2007.

[29] C. Shi, W. Cui, S. Liu, P. Xu, W. Chen, and H. Qu, "Rankexplorer: Visualization of ranking changes in large time series data," *IEEE Trans. Vis. Comput. Graphics*, vol. 18, no. 12, pp. 2669–2678, Dec 2012.

[30] S. Gratzl, A. Lex, N. Gehlenborg, H. Pfister, and M. Streit, "Lineup: Visual analysis of multi-attribute rankings," *IEEE Trans. Vis. Comput. Graphics*, vol. 19, no. 12, pp. 2277–2286, 2013.

[31] C. Perin, R. Vuillemot, and J.-D. Fekete, "A table!: Improving temporal navigation in soccer ranking tables," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 887–896.

[32] Z. Wang, N. Ferreira, Y. Wei, A. S. Bhaskar, and C. Scheidegger, "Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 1, pp. 681–690, Jan 2017.

[33] "Wikimedia downloads," https://dumps.wikimedia.org/, accessed: 2015-03-31.

[34] "Yahoo! webscope dataset yfcc-100m," http://research.yahoo.com/Academic_Relations, accessed: 2015-03-31.

[35] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236.

Fabio Miranda Fabio Miranda is a PhD Candidate in the Computer Science and Engineering Department at NYU Tandon School of Engineering. He received a MSc in Computer Science from PUC-Rio, Brazil. During this period, Fabio worked as a researcher and software engineer developing visualization tools for the oil industry. Fabio's current research focuses on large scale data analysis, data structures and urban data visualization.

Lauro Lins Lauro Lins is a researcher at the Information Visualization Department at AT&T Labs. He received a BSc/MSc in Computer Science and a PhD in Computational Mathematics from Universidade Federal de Pernambuco (Brazil). Lauro has also worked as a Post-Doc at the SCI Institute (Utah) and as an Assistant Research Professor at NYU-Poly. Lauro's research focuses on algorithms, data structures, and systems for enabling intuitive and scalable data visualization.

James T. Klosowski Jim Klosowski is the Director of the Information Visualization Research Department at AT&T Labs. Prior to joining AT&T in 2009, Jim worked on interactive computer graphics and scalable visualization systems at the IBM T.J. Watson Research Center. Jim's current research focuses on all aspects of information visualization and analysis, but in particular working with large geospatial, temporal, and network datasets.

Claudio T. Silva Claudio Silva is a Professor of Computer Science and Engineering and Data Science at NYU. Silva's research lies in the intersection of visualization, data analysis, and geometric computing, and recently has focused on urban and sports data. Silva has received a number of awards: IEEE Fellow, IEEE Visualization Technical Achievement Award, and elected Chair of the IEEE Technical Committee on Visualization and Computer Graphics.