Packed-Memory Quadtree: a cache-oblivious data structure for visual exploration of streaming spatiotemporal big data

Julio Toss, Cícero Augusto de Lara Pahins, Bruno Raffin, João Luiz Dihl Comba

Packed-Memory Quadtree: a cache-oblivious data structure for visual exploration of streaming spatiotemporal big data

Julio Toss[1,2], Cícero A. L. Pahins[1], Bruno Raffin[2], and João L. D. Comba[1]

[1]Instituto de Informática - UFRGS, Porto Alegre, Brazil
[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble

September 18, 2018

Abstract

The visual analysis of large multidimensional spatiotemporal datasets poses challenging questions regarding storage requirements and query performance. Several data structures have recently been proposed to address these problems that rely on indexes that pre-compute different aggregations from a known-a-priori dataset. Consider now the problem of handling *streaming* datasets, in which data arrive as one or more continuous data streams. Such datasets introduce challenges to the data structure, which now has to support dynamic updates (insertions/deletions) and rebalancing operations to perform self-reorganizations. In this work, we present the Packed-Memory Quadtree (PMQ), a novel data structure designed to support visual exploration of streaming spatiotemporal datasets. PMQ is *cache-oblivious* to perform well under different cache configurations. We store streaming data in an internal index that keeps a spatiotemporal ordering over the data following a quadtree representation, with support for real-time insertions and deletions. We validate our data structure under different dynamic scenarios and compare to competing strategies. We demonstrate how PMQ could be used to answer different types of visual spatiotemporal range queries of streaming datasets.

1 Introduction

Advanced visualization tools are essential for big data analysis. Most approaches focus on large static datasets, but there is a growing interest in analyzing and visualizing data streams upon generation. Twitter is a typical example. The stream of tweets is continuous, and users want to be aware of the latest trends. This need is expected to grow with the Internet of things (IoT) and massive deployment of sensors that generate large and heterogeneous data streams. Over the past years, several in-memory big-data management systems have appeared in academia and industry. In-memory databases systems avoid the overheads related to traditional I/O disk-based systems and have made possible to perform interactive data-analysis over large amounts of data. A vast literature of systems and research strategies deals with different aspects, such as the limited storage size and a multi-level memory-hierarchy of caches [42]. Maintaining the right data layout that favors locality of accesses is a determinant factor for the performance of in-memory processing systems.

Stream processing engines like Spark [41] or Flink [1] support the concept of *window*, which collects the latest events without a specific data organization. It is possible to trigger the analysis upon the occurrence of a given criterion (time, volume, specific event occurrence). After a window is updated, the system shifts the processing to the next batch of events. There is a need to go

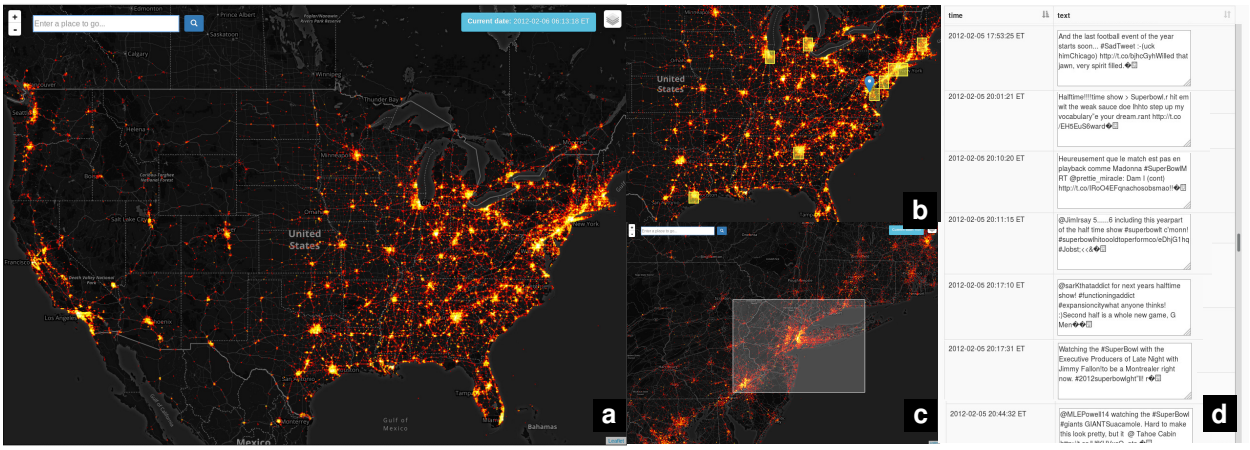


Figure 1: A Twitter stream is consumed in real-time, indexed and stored in the Packed-Memory Quadtree. **(a)** : live heat-map displays tweets in the current time window. **(b)** : alerts indicate regions with high activity of Twitter posts at the moment. **(c)** : the interface allows to drill-down into any region and query the current data. **(d)** : the actual data can be retrieved from the Packed-Memory Quadtree to analyze the tweets in the region of interest.

one step further to keep a live window continuously updated while having a fine grain data replacement policy to control the memory footprint. The challenge is the design of dynamic data structures to absorb high rate data streams, stash away the oldest data to stay in the allowed memory budget while enabling fast queries executions to update visual representations. A possible solution is the extension of database structures like R-trees [20] used in SpatiaLite [37] or PostGis [34], or to develop dedicated frameworks like Kite [26] based on a pyramid structure [27, 28].

In this paper, we propose a novel self-organized cache-oblivious data structure, called Packed-Memory Quadtree (PMQ), for in-memory storage and indexing of fixed length records tagged with a spatiotemporal index. We store the data in an array with a controlled density of gaps (*i.e.*, empty slots) that benefits from the properties of the *Packed Memory Arrays* [3]. The empty slots guarantee that insertions can be performed with a low amortized number of data movements ($O(\log^2(N))$) while enabling efficient spatiotemporal queries. During insertions, we rebalance parts of the array when required to respect density constraints, and the oldest data is stashed away when reaching the memory budget. To spatially subdivide the data, we sort the records according to their Morton index [17], thus ensuring spatial locality in the array while defining an implicit, recursive quadtree, which leads to efficient spatiotemporal queries. We validate PMQ for consuming a stream of tweets to answer visual and range queries. Figure 1 shows the user interface prototype built to support the data analysis process. PMQ significantly outperforms the widely adopted spatial indexing data structure R-tree, typically used by relational databases, as well as the conjunction of Geohash and B^+-tree, typically used by NoSQL databases [15]. In summary, we contribute (1) a self-organized cache-oblivious data structure for storing and indexing large streaming spatiotemporal datasets; (2) algorithms to support *real-time* visual and range queries over streaming data; (3) performance comparison against tried and trusted indexing data structures used by relational and non-relational databases.

2 Related Work

In building an efficient system to enable interactive exploration of data streams, we must deal with challenges common to areas like in-memory big-data, stream processing, geospatial processing, and information visualization.

Data Structures. Data structures need to dynamically

process streams of geospatial data while enabling the fast execution of spatiotemporal queries, such as the *top-k* query that ranks and returns only the k most relevant data matching predefined spatiotemporal criteria. One approach is to store data continuously in a dense array following the order given by a space-filling curve, which leads to desirable data locality. Inserting an element takes on average $O(n)$ data movements, i.e., the number of elements to move to make room for the newly inserted element. The cost of memory allocations can be reduced using an amortized scheme that doubles the size of the array every time it gets full. However, elements are often inserted in batches in an already sorted array. In that case, one approach is to use adaptive sorting algorithms to take advantage of already sorted sequences [13, 11, 31]. Timsort [33] is an example of an adaptive sorting algorithm with efficient implementations. We show experiments that compare our data structure to Timsort. Another possibility is to rely on trees of linked arrays. The B-tree [2] and its variations [10] are probably the most common data structure for databases. The UB-Tree is a B-tree for multidimensional data using space-filling curves [35]. These structures are seldom used for in-memory storage with a high insertion rate. They are competitive when data access time is large enough compared to management overheads, often the case for on-disk storage. Such data structures are *cache-aware*, *i.e.*, to ensure cache efficiency they require a calibration according to the cache parameters of the target architecture.

Sparse arrays are an alternative that lies in between dense arrays and trees of linked arrays. Data is stored in an array larger than the actual number of elements to store, using the extra room to make insertions and deletions more efficient. Itai et al. [21] were probably the first to propose such data structure. Bender et al. [3, 5] refined it, leading to the Packed Memory Array (PMA). The main idea is that by maintaining a controlled spread of gaps, insertions of new elements can be performed moving much fewer than $O(N)$ elements. The insertion of an element in the PMA only requires $O(\log^2(N))$ amortized element moves. This cost goes down to $O(\log(N))$ for random insertion patterns. Bender and Hu [5] also proposed a more complex PMA, called adaptive PMA, that keeps this $O(\log(N))$ for specific insertion patterns like bulk insertions. PMA is a *cache-oblivious* data structure [16], *i.e.* it is cache efficient without explicitly knowing the cache parameters. Such data structures are interesting today since the memory hierarchy is getting deeper and more complex with different block sizes. Cache-oblivious data structures are seamlessly efficient in this context. Bender et al. [3, 4] also proposed to store a B-tree on a PMA using a van Emde Boas layout, leading to a cache-oblivious B-tree. However, it leads to a complex data structure without a known practical implementation. Still, PMA has few known applications. Mali et al. [29] used PMA for dynamics graphs. Durand et al. [12] relied on PMA to search for neighbors in particle-based numerical simulations. They indexed particles in PMA based on the Morton index computed from their 3D coordinates. They proposed an efficient scheme for batch insertion of elements, while Bender relied on single element insertions. In this paper, we propose to extend PMA for in-memory storage of streamed geospatial data.

Stream Processing and Datacubes Structures. Stream processing engines, like GeoInsight for MS SQL StreamInsight [22], are tailored for single-pass processing of the incoming data without the need to keep in memory a large window of events that require an advanced data structure. The emergence of geospatial databases led to the development of a specialized tree, called R-Tree [20], that associates a bounding box to each tree node. Several data processing and management tools have been extended to store geospatial data relying on R-trees or variations like the SpatiaLite [37] extension for SQLite or PostGis [34] for PostgreSQL. Our experiments include comparisons with both. Though such spatial libraries brought flexibility for applications in the context of traditional spatial databases, their algorithms are not adapted to consume a continuous data stream. Magdy et al. [27, 28] proposed an in-memory data structure to query and update real-time streams of tweets. Initially called Mercury, then Venus and eventually Kite [26] for the latest implementation (Kite is also benchmarked in our experiments). They rely on a pyramid structure that decomposes the space into H levels. Periodically the pyramid is traversed to remove the oldest tweets to keep the memory footprint below a given budget. This idea to rely on bounding volume hierarchies is also popular in computer graphics for indexing 3D objects and accelerating collision detection [40]. One difficulty in these data structures is to ensure fast insertions while keeping the tree balanced. The data structure

may also become too fragmented in memory leading to an increase of cache misses. The partitioning criteria are based on heuristics. There is often no theoretical performance guarantees.

Finally, we point out that several data structures were proposed recently for the visual analysis of big data. A common theme is the idea of pre-computing aggregations in *datacubes* proposed by Gray et al. [19]. Representative work include imMens [25], Nanocubes [24], Hashedcubes [32] and Gaussian Cubes [39], all designed for processing static data. Streamcube [14] combines an explicit spatio-temporal quadtree with datacubes for on-pass hashtag clustering. The PMQ proposes a dynamic data structure supporting the main visual queries described in these works.

3 Packed-Memory Quadtree

In this section, we explain the PMQ internal organization, update methods, and query types to support stream data analysis. In our description, we used as motivation dataset a stream of tweets, each with spatial location and associated satellite data. Our PMQ builds upon a PMA data structure but departs from the original one on different aspects:

- Data are indexed and sorted according to their Morton index to enforce data locality for efficient spatial queries;
- Data insertions are performed by batches in a top-down scheme to factor rebalance operations, while the PMA inserts elements one by one using a bottom-up scheme;
- The PMQ has a limited memory budget. Once we reach the maximum size and density, we stash the oldest data through a customized process;
- Support for answering geospatial visual queries to allow interactive analysis of streaming datasets.

3.1 The PMQ Data Structure

We present here the PMQ data structure that strongly relies on the Packed-Memory Array [21, 5]. A PMQ is an array with extra space to maintain a given density of (empty) gaps between the (valid) elements. An array of size N (counting the gaps) is divided into $O(N/\log(N))$ consecutive *segments* of size $O(\log(N))$. For convenience, the number of segments is chosen to be a power of 2. PMQ is stored in memory and keeps, for each element, an indexing *key* and associated *value*. Segments are paired hierarchically by *windows* following a tree structure. A level 0 window corresponds to a single segment, while the h level window encompasses the full array. The *density* of a window is the ratio between the number of (valid) elements in the window and its size. As we will see, the PMQ goal is to control the window densities to ensure insertion or removal of elements can be performed at low cost.

The minimum and maximum density bounds for a window at level l are respectively ρ_l and τ_l. We define density bounds such that:

$$\rho_0 < \cdots < \rho_h < \tau_h < \cdots < \tau_0. \quad (1)$$

Thus, the larger a window, the more constraining its density bounds. The minimum and maximum densities of windows at intermediate levels are linearly interpolated between the $[\rho_0, \rho_h]$ and $[\tau_h, \tau_0]$ thresholds as defined below:

$$\tau_l = \tau_h + (\tau_0 - \tau_h)\frac{(h-l)}{h}, \quad (2)$$

$$\rho_l = \rho_h - (\rho_0 - \rho_h)\frac{(h-l)}{h}, \quad (3)$$

and $2\rho_h < \tau_h$. The upper (resp. lower) density threshold decreases (resp. increases) by $O(1/\log(N))$. This $O(1/\log(N))$ interval is fundamental to guarantee that an insertion or deletion requires $O(\log^2(N))$ amortized data movements. A *valid* PMQ is the one that satisfies density values for all windows. To compute a window density in constant time without having to scan its content, we associate an auxiliary *accounting array* to store the number of valid elements per window. This array requires an extra $O(2\frac{N}{\log(N)})$ of memory space. Figure 2 illustrates a PMQ for 9 elements. The density thresholds are: $\rho_0 = 0.08$, $\rho_2 = 0.3$, $\tau_2 = 0.7$, $\tau_0 = 0.92$, and the values for ρ_1 and τ_1 are defined according to Equation 2 and Equation 3.

3.2 Data Indexing

Space-filling curves define a map of multidimensional points to one dimension, which allows to order the data

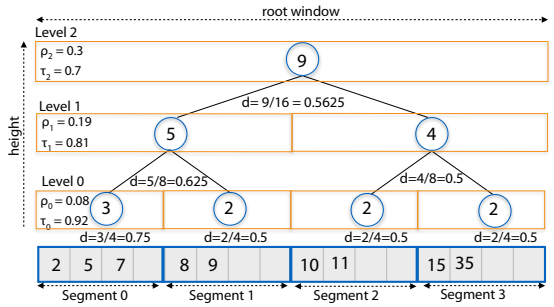


Figure 2: PMQ with 4 segments: ρ_l and τ_l are the minimum and maximum densities allowed at each level l of PMQ, d the actual window density. The numbers in circles count the valid element per window and are stored in the PMQ accounting array.

in a 1D array. The PMQ relies on the Morton space-filling curve to store the elements sorted according to their Morton index. The Morton curve enables to linearly index data with 2D coordinates through a low cost bit-level operation, while preserving well the data spatial locality. Data that are close in 2D tend to have close Morton index (also called Z-index or geohash) and thus are stored nearby in the PMQ. Elements with the same Morton index are sorted according to their timestamp (e.g., tweet timestamp).

The Morton curve actually defines a recursive Z-shaped space partitioning that follows a quadtree subdivision. The ordering generated by the Morton curve is equivalent to the ordering produced by a depth-first traversal in a quadtree [7]. The number of bits used for the Z-index defines this quadtree max depth. For the PMQ, this number of bits is static. For each new incoming element, its Z-index is computed from its (x,y) position coordinates by interleaving the bits of x and y, defined according to Equation 4 and Equation 5 (note that both equations output integer values). Truncating by 2 bits a Z-index provides the index of the parent cell in the quadtree. Figure 3 illustrates a PMQ for 9 elements in a 2D domain. Elements are sorted based on their Z-index that recursively defines an implicit quadtree subdivision (never stored).

3.3 Dynamic Updates

The PMQ is designed to store a stream of data inserted by batches. The insertion starts at the topmost window of the PMQ (the full array) by checking if a violation of the density bounds occurs after inserting a batch of incoming elements. Consider a valid PMQ filled with K ordered elements. Suppose we want to insert a batch of I new elements stored in an *insertion array*. The goal is to insert these new elements while keeping the PMQ valid. The insertion algorithm follows a top-down approach.

If the density of the full PMQ array goes beyond τ_h counting the I new elements, we first scan the array to count the number of elements with a timestamp older than a given threshold λ. If removing these elements the PMQ meets its density bounds, we remove them, rebalance evenly the remaining elements while inserting the I new elements (sorted first). Otherwise, we perform the same operation but first doubling the PMQ array size. The constraint $2\rho_h < \tau_h$ guarantees that the density of this double size PMQ is above ρ_h.

We now describe the batch insertion. Consider the case where inserting elements does not cause the full array density to go over τ_h. Let p be the key of the first element of the right top window. We re-order the insertion array such that elements smaller than p are on the left of the insertion array, while the others are on the right. The left elements will go in the top left window of PMQ, the others in the top right window. We test for both top windows their new densities against the corresponding thresholds, counting the elements to insert. If at least one top window does not respect the density thresholds, we *rebalance* the elements of the full array while including the new ones, *i.e.*, we evenly redistribute all elements. After the rebalance we have the guarantee that all windows down to segments satisfy their density bounds since densities are less constraining as the window size decreases. Otherwise, density thresholds are respected and the algorithm proceeds recursively. In the best case, *rebalances* only span individual segments. We update the accounting array after each batch insertion.

Note that when performing a rebalance we keep the elements sorted based on their Morton index and insertion timestamp. Since the sorting during rebalancing is stable, the only requirement is that we order the elements in the batch array by arrival time (which is the natural

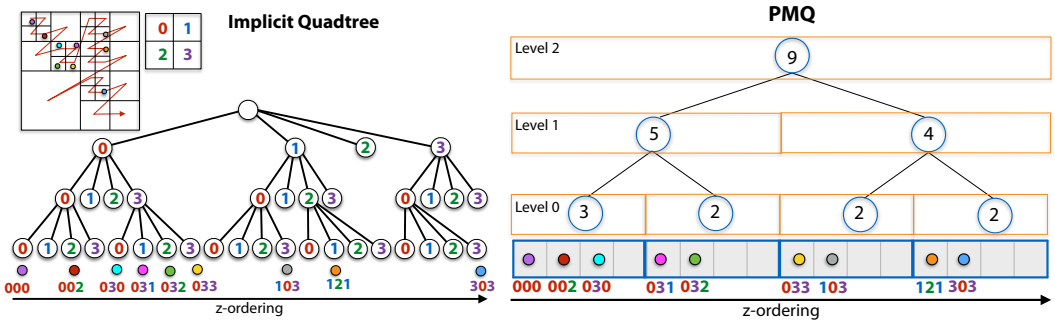


Figure 3: **PMQ** storage of 9 z-indexed elements from a 2D domain (right). Z-indices correspond to a quadtree actually never built as z-index are directly computed by interleaving their x-y bits (Right).

order in a real-time stream). Rebalancing is automatically triggered when needed. No heuristic is needed to decide when to split or delete a node as in [27, 28]. Memory allocations are only needed when doubling the array. We control memory consumption by setting the threshold timestamp λ based on the arrival rate of the data stream. In practice, PMQ self-stabilizes: it doubles its size until reaching a *steady* state where insertions and deletions balance themselves. The accounting array is updated with each window rebalance.

Notice that none of these operation use the lower densities. But they are kept in the PMQ description and supported in our implementation for completeness. They can be useful for scenarios not evaluated here. They enable to trigger window rebalances when removing elements.

The PMQ is a cache-oblivious data structure as it does not depend on cache parameters. The worst-case amortized cost is $O(\log^2(N))$ per insertion. The proof is given in A.

3.4 Query Types

We present three types of queries that we support in the current implementation of PMQ: heatmap, range, and top-k queries. Other types of queries can be incorporated if needed.

Heatmap Queries. The visual interface of our system uses a world heatmap continuously updated based on the content stored in the PMQ (see Figure 4). An important observation is that Z-cells do not align with the tiles of the heatmap. Also, the interface allows zooming into specific regions of the world, thus needing to map the tiles of the heatmap grid to Z-cells. We compute the *zoom* level ζ in the quadtree of Z-cells corresponding to each heatmap tile. If $\zeta = 0$, we need to aggregate the full PMQ data into a single tile, corresponding to the full PMQ data. Heatmap construction for a single tile consists of counting the number of data samples for each pixel drawn inside the tile. For instance, a tile of $256x256$ pixels corresponding to a quadtree node at level ζ is computed by counting for each pixel the number of elements stored in the corresponding descendant Z-cells at depth $\zeta + 8$. As we only need to *count* the elements per tile (element values are not necessary), we accelerate counting using the accounting array.

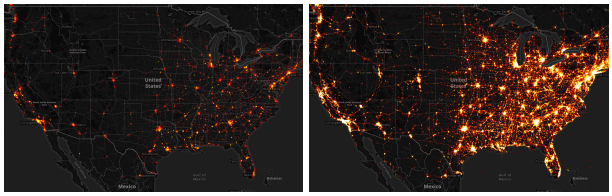


Figure 4: The heatmap is updated dynamically as the stream of tweets is received. With an average insertion rate of 1000 $tweets/sec$ we show the heatmap when PMQ contains $100K$ (left) or $10M$ (right) elements.

Range Queries. A range query is a spatial query that requests all elements stored in a rectangular region (Figure 5). We define a range query by the corners of a bounding box in the map. Given a range query, we have to ac-

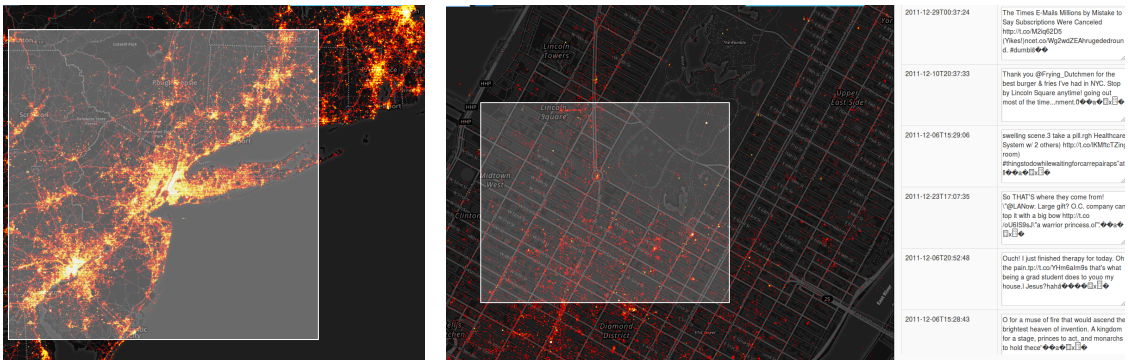


Figure 5: Heatmap zoom and range queries are used to explore the latest stream of tweets around the New York city area. The in-memory storage of PMQ provides fast access to the actual tweets' content allowing real-time user interaction even on large range queries (R = radius of the selected area).

cess the PMQ to retrieve all records within the rectangular region. We return the result to the application for any post-processing of this information. In our interface, we currently display a subset of the results (e.g., a fixed number of tweets). Unlike heatmaps, which queries the PMQ using a fixed resolution grid, the range query can define an arbitrary region. Therefore, we need to find the coarsest Z-cells that contain the bounding box of the range query. Since we do not store the quadtree explicitly, we follow the Z-ordering recursively to find the Z-cells that fully enclose the bounding box. We refine each Z-cell to locate the leaf Z-cells intersecting or included in the bounding box. We refer to the book by Samet [36] for the range query algorithm for quadtrees. Using the Z-cell indices, we locate through binary searches in the PMQ the ranges that contain the needed elements.

Top-k queries. The top-k query combines the temporal ordering with the spatial dimension to find the most relevant data according to a given spatiotemporal interval. This query is processed like the range query but filters the candidate values in a temporary priority queue of size k. Given a 2D point p, the top-k query finds the elements with k lowest values according to a score function. The search space of the top-k queries can be reduced using both spatial and time thresholds. The parameter R defines a radius where records are going to be ranked by distance to p. Similarly, the parameter T limits the oldest timestamp to consider in the scoring function. Both scores are then normalized and combined into a final score to balance the importance of the spatial and temporal dimensions. Elements in the same Z-cell (*i.e.*, with the same Morton code) are ordered based on their timestamp. The top-k search uses the same refinement algorithm as for range queries to find the records included inside the bounding box of radius R centered at p. We insert the returned elements into a priority queue of max-size k ordered by the spatiotemporal score to keep only the k elements with the highest score.

4 Implementation

We implemented PMQ in C++. Each element has a 64-bit *key* representing the spatial index plus a *value* for storing additional information. We rely on 50-bit length Morton code for the spatial index, defining a quadtree with a fixed depth of 25 levels of refinement. The Morton index is computed from the (x,y) coordinated obtained through the EPSG:3857 projection of the longitude (lon) and latitude (lat) associated with each element, defined as:

$$x = \left(\frac{lon + 180}{360} . 2^z \right), \tag{4}$$

$$y = \left(1 - \frac{\ln\left(\tan\left(lat . \frac{\pi}{180}\right) + \frac{1}{\cos\left(lat . \frac{\pi}{180}\right)} \right)}{\pi} \right) . 2^{z-1}, \tag{5}$$

where both $(x,y \in \mathbb{Z} \mid 0 \leq x,y < 4^z)$ and Z is the maximum quadtree depth.

The choice of record content to consider depends on the application needs. If only the metadata is required, we used a 16-byte struct to store latitude (32-bit float), longitude (32-bit float) and insertion timestamp (64-bit unsigned integer).

Our implementation uses PMQ segments of a fixed size of eight elements as we found no significant performance benefit in increasing the segment size with the array size. When rebalancing a window, we always pack data at the left of each segment to favor low-level optimizations like prefetching. Besides giving the window densities, the accounting array is also used to get the position of the last valid element of each segment, to avoid scanning the empty slots.

We target streaming scenarios where the PMQ is used to store the most recent incoming data sorted in memory, keeping as much data as possible for a given memory budget. Data are removed only when the PMQ is full, keeping the PMQ density high enough to avoid reducing its size (waste of memory and time as leading to a PMQ oscillating through cyclic size halving and doubling). Thus rebalances are only triggered when the window upper density bounds are reached. In our experiment we use $\tau_0 = 0.92$ and $\tau_h = 0.7$ that give the best performance tradeoff (see Section 6.3).

5 Examples of Visualization Analysis using PMQ

We present an example of the interactive exploration of tweets enabled by PMQ and its user interface. The dataset consists of geolocated tweets collected with the Twitter API between November 2011 and June 2012 over the United States. The dataset has a total of 210.6 million tweets. We simulate an incoming stream of tweets by grouping them into batches of fixed size and iteratively inserting them into a given data structure.

5.1 Drill-down exploration of a Twitter stream

We provide a visual interface that allows a drill-down exploration of a tweet stream and support different queries. PMQ supports the storage in memory of the latest tweets, thus filling the gap between stream processing engines working only on a small window of the input stream, and classical solutions on persistent storage. The heatmap enables to display the concentration of tweets posted over the last hours. PMQ can index one second of tweets and keep all the elements sorted in less than 1.5 ms on average, with a maximum at about 1s when a batch triggers a top rebalance with element removal (Table 3). It is more than capable of keeping up with the stream rate and support visual queries. The interface allows the user to zoom into the heatmap or to perform range queries. We display the tweets inside selected areas in a separate area next to the map. Figure 5 shows the combined use of heatmaps and range queries at different zoom levels over New York. The user can interactively zoom until finding the desired information.

5.2 Allert detection of regions with high tweet rates

Systems like TwitInfo [30] provide an interface for visualizing real-time Twitter data. Based on the user-given keyword search, the system fetches the matching twitter stream and generates an aggregated higher-level visualization, which is kept up-to-date with the incoming tweets. Such exploratory framework enables a better understanding of on-going events. However, since the stream is filtered with a fixed input keyword, it is not suitable when monitoring unexpected events. One example of an unexpected event is the monitoring of the volume of tweets in a given region. We implemented a simple pre-processing of batches to trigger alerts in regions with high tweet activity. Once an alert is triggered, the user can further investigate it by interactively exploring the last received tweets. During exploration, one can also perform top-k queries to retrieve the top-most relevant tweets at a given point. For instance, on February 5th of 2012 at 14:22 UTC, the system indicates a high tweet activity over Indianapolis. Zooming into the alert zone and using range queries, we observe that many people are at the Lucas Oil Stadium commenting about the Super Bowl game. We set a top-k query at the stadium to follow the most relevant tweets nearby. The filtered feed displayed on the right panel of the interface shows tweets with information like the teams playing (New York Giants Vs. New England Patriots), or about the Madonna's show during half-time

(Figure 6).

6 Performance Evaluation

We created a series of benchmarks to evaluate the performance of PMQ. When possible, we showed comparisons against competing solutions from the industry and other open-source libraries. The B-Tree compared in our experiments is the `stx::btree`, an efficient open source implementation of in-memory B+Tree [8]. The R-Tree implementation is from the `Boost` C++ template library [9]. We conducted the experiments to allow reproducing results and exploring different parameter configurations. We created a GitHub repository to store supplemental material and additional benchmarks[1]. The benchmarks were run on a dedicated Linux machine with an Intel i7-4790 CPU @ 3.60GHz with 32GB of main memory.

6.1 Evaluating Storage Solutions for Spatial Data

We conducted a set of experiments to evaluate PMQ against two different storage solutions for spatial data. The first type of storage solution is traditional open source relational databases (RDBMS) with geospatial library extensions, such as: (1) in-memory SQLite + SpatiaLite and (2) PostgreSQL + PostGIS. SQLite uses in-memory storage, while PostgreSQL uses a disk. Typically, RDBMS store entries in a table and build an additional spatial index separately. This optional index supports efficient spatial queries that contain geometric predicates. The second storage solution are dense vectors using a pointer-based quadtree index on a dense C++ `std::vector`. Spatial ordering in the container uses two sorting algorithms: (1) the C++ `std::sort()` implementation from `GNU GCC libstdc++` and (2) the `C++ TimSort` [18] adaptive sorting algorithm.

This set of benchmarks gives an insight into the scalability of insertion and query operations of the solutions above. The different data structures are initially empty and increase their size as elements arrive in batches. We measured the insertion time of each batch, including the time for updating the index (in case of RDBMS) and physically storing the data. We also measured the time for accessing data from the storage. After each batch insertion, we queried all elements indexed by the data structure (Figure 7). As can be seen, even with a small number of elements, the database solutions have poor scalability. While PostGIS uses disk storage, it spends most of the time optimizing the index and physically reordering elements on disk. SpatiaLite, on the other hand, seems to have a less efficient indexing strategy than PostGIS. It spends less time on indexing and insertion operations, but pays a significant cost to access the data, even if stored in memory. None of the database solutions are suited to the real-time latency requirements of update and read operations. The spikes on PMQ benchmarks correspond to doubling the array size when the structure reaches the maximum density. The sparse storage of PMQ allows reducing the time on insertion when compared to the dense vectors. In the next experiments, we remove the database solutions from the comparisons since they are an order of magnitude slower than the vector-based storage approaches, and compare against low-level structures such as B-trees and R-trees.

6.2 Evaluating Insertions

We evaluate the scalability of the data structures by comparing their trade-offs between insertions and scanning operations. These two operations represent a performance compromise of two conflicting workloads. While tree-based data structures, like B-Tree and R-Tree, show good insertion performance (Figure 8(a)), they fail in maintaining in-memory data locality, which has a significant impact on scanning operations (Figure 8(b)). The solution using dense sorted vectors reveals the importance of data locality when scanning. We derive the lower-bounds of scanning performance when we achieve the best locality. However, its update costs for frequent insertions make it impracticable for large amounts of data. PMQ shows a good compromise between these two operations. On insertions, it performs similarly to the R-Tree, it is 2X times slower than the B-Tree and scales logarithmically with the size of the data structure. At the same time, PMQ pays only a small constant overhead relative to best possible scanning data-structure, the dense vector. Compared to the tree-based data-structures, with 10 M elements, the scan operations on PMQ are 3X times faster than B-Tree and 5X faster than R-Tree.

[1] `https://github.com/pmq-authors/pmq-extras`

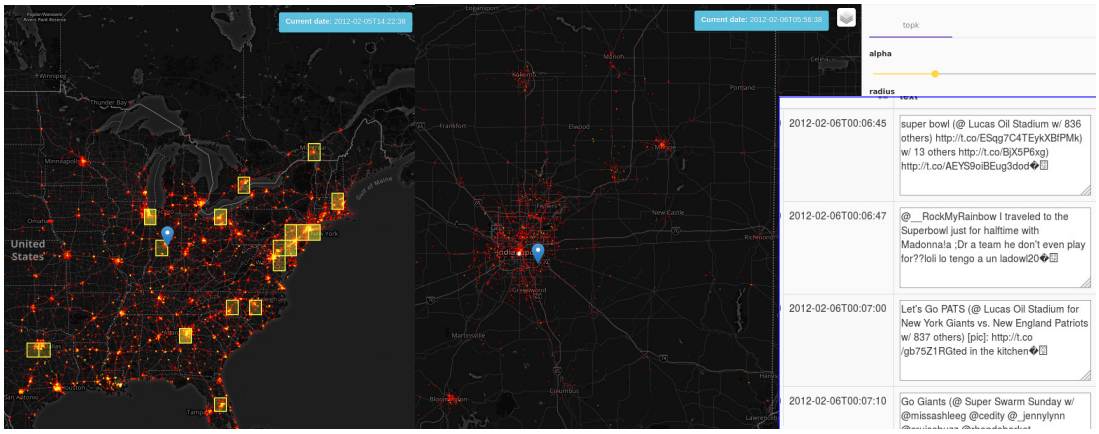


Figure 6: Allert detection: triggers configured by the user show alerts (yellow squares) on several cities with a high rate of tweet arrival during Super Bowl 2012. We select the Indianapolis region (where the game occurs) and filter tweets using a top-k query to retrieve the most relevant tweets in the area.

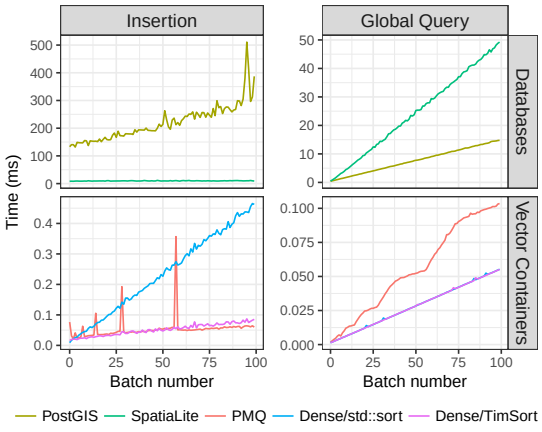
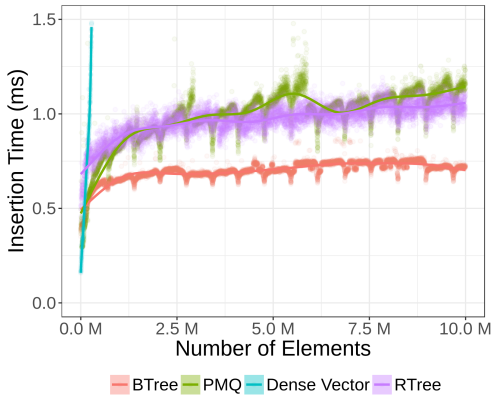


Figure 7: Performance comparison of spatial data storage solutions. **Top row:** standard geospatial databases can not handle real-time insertions. **Bottom row:** in-memory containers based on dense or sparse (PMQ) vectors.

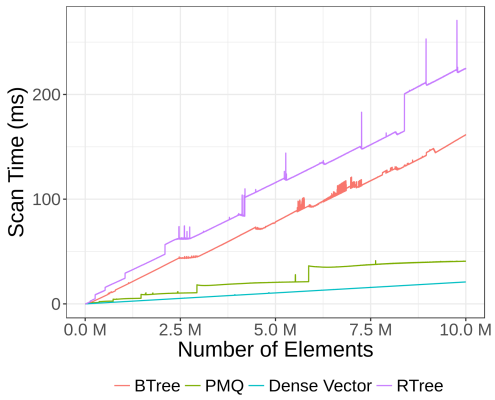
6.3 Evaluating Bulk Deletions

In the case of streaming data, the memory available limits the storage of information. We used a stashing procedure to evict old data while receiving new incoming records. Data structures are in a *steady regime* if they cannot grow after inserting a given number of records. At this point, a bulk removal is triggered to remove a number of the oldest elements in the data structure, given by the threshold parameter λ. How often removals are triggered depends on the rate of the incoming stream and the threshold λ. We keep the incoming rate constant by inserting a batch of 1000 elements at each simulation step. Because the bulk removal is slower than regular insertions, the choice of λ has an impact in two performance indicators: the average execution time of each operation (Figure 9(a)) and the bulk removal execution time (Figure 9(b)). To evaluate the indicators and choose the best λ, we insert in each structure a dataset of nearly 46 million elements and set the maximum number of elements to be stored to half of this size.

Figure 9 shows running times for λ varying from 0.1% to 50% of the maximum capacity. For both the B-Tree and the R-Tree, the removal size has to be chosen carefully to balance the time spend on each removal operation and the total running time. If the removal size is large (over 3.13%), each operation deletes many elements

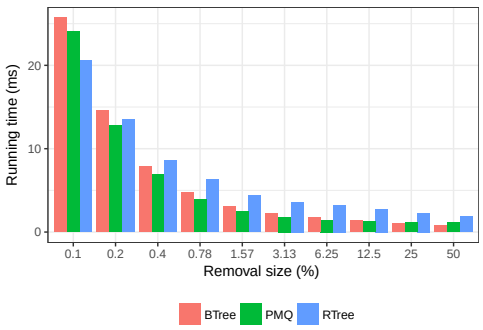


(a) Insertion of $10M$ elements by batches of 1000 elements.

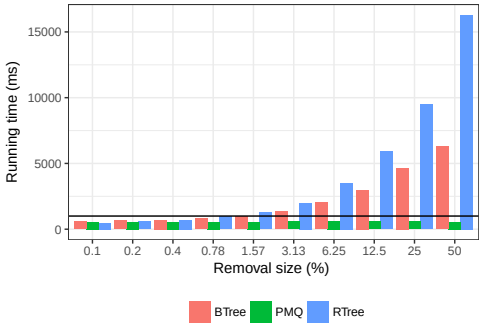


(b) Time for a full scan of the dataset after each batch insertions.

Figure 8: Scalability insertion and scan operation.



(a) Total average running time of the experiment.



(b) Average time of each bulk removal operation.

Figure 9: **Steady data regime:** deletions are performed periodically. For each test, we insert a dataset of 46 million elements. The maximum number of elements allowed is half of the dataset size (around 23 million elements), and removals are configured with different percentages of the maximum.

at once causing expensive removal operations that take over 1 second (black horizontal line in Figure 9(b)). If the removal size is small (under 0.78%), each removal is fast since only a small percentage of elements are evicted. However, it increases the frequency of removals impairing the total running time (Figure 9(a)). In opposite, the PMQ triggers element removal automatically when the top density threshold is reached. As a consequence, the execution time of removals is much less sensitive to λ. As Figure 9(b) shows, for any removal size, the running time is under one second. As expected, the total running time for all structures is best with larger and less frequent removals (Figure 9(a)). Therefore the choice of the parameter λ should favor larger removal sizes.

In Table 1, we summarize the results from Figure 9(a) and Figure 9(b). We show, for each structure, the best tradeoff between removal and average running time. The B-Tree and R-Tree require a small removal percentage (λ), while PMQ removes 50% of if elements and it is $2\times$ faster for both removal and average running time. In Table 2, we take the λ value that gives the best performance tradeoff for the B-Tree and set it to the other data structures. Once again, PMQ performs best and is the only one to make removal operations in less than a second.

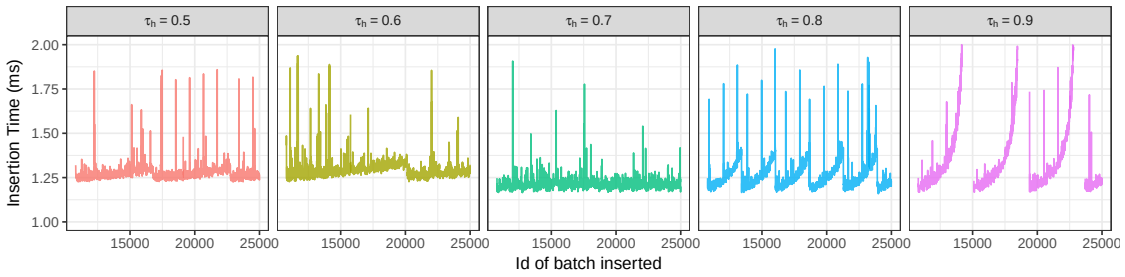


Figure 10: PMQ performance at steady regime for different τ_h thresholds.

In Figure 10 we compare several values of τ_h threshold (with fixed $\tau_0 = 0.92$) for the PMQ at steady regime: when the PMQ density reaches τ_h a bulk removal is triggered keeping at least 10.8 M elements. The value $\tau_h = 0.7$ gives the best average insertion time. The PMQs with $\tau_h = 0.5$ and 0.6 require twice more storage memory compared to the ones with 0.7, 0.8 and 0.9, with a high average insertion time. High τ_h values (0.8 and 0.9) leave the PMQ fill, leading to costly rebalances of large windows. The value $\tau_h = 0.7$, chosen for all our other experiments, gives the best average insertion time with low memory footprint. A high value $\tau_0 = 0.92$ gives the best results, allowing some local high-density spots.

6.4 Evaluating the Rebalancing Procedure

PMQ supports a rebalancing procedure that is only activated when necessary. In Table 3 we simulate a tweet insertion rate of 1000 tweets per second. We present the average insertion time in PMQ after reaching a steady state (*i.e.* after the first top-level rebalance that started removing tweets). Notice that during this steady state, elements deletion neither leads to halving nor doubling the PMQ size. Between the top two level rebalances, the number of elements in the container varies from $Elts\_min$ to $Elts\_max$. The maximum value in the table corresponds to a single insertion that triggers a top-level rebalance. Although these periodic rebalances can take up to one second, this latency is hidden from the user as the mean insertion time (and the 99th percentile) is much smaller than the insertion rate.

We also evaluated how PMQ scales with varying insertion rates. We used a time window of 6 hours and increase the insertion rate up to 8k tweets/s. Figure 11 shows the average insertion time and standard deviation. PMQ takes less than 8 *ms* to digest 6000 tweets per second, the current average number of tweets posted per second worldwide [38].

Algo.	Min. Elts	RM (ms)	Avg. Run Time (ms)	RM Interval	λ
B-Tree	22.7 M	1331	2.19	735	3.13%
PMQ	11.7 M	**550**	**1.09**	11744	50%
R-Tree	23.1 M	1287	4.43	368	1.57%

Table 1: Parameter λ set for the best relation of removal *RM* time and average *Avg* runnning time for each algorithm.

Algo.	Min. Elts	RM (ms)	Avg. Run Time (ms)	RM Interval	λ
B-Tree	22.7 M	1331	2.19	735	3.13%
PMQ	22.7 M	**601**	**1.80**	735	3.13%
R-Tree	22.7 M	1984	3.60	735	3.13%

Table 2: Comparison using same λ optimized for the B-Tree.

λ	Elts_min	Elts_max	Mean	99%	Max
$3h$	$10.8 * 10^6$	$11.74 * 10^6$	1.209	1.066	265.613
$6h$	$21.6 * 10^6$	$23.48 * 10^6$	1.310	1.134	554.971
$9h$	$32.4 * 10^6$	$46.97 * 10^6$	1.278	1.587	1007.040
$12h$	$43.2 * 10^6$	$46.97 * 10^6$	1.423	1.321	1045.950

Table 3: Insertion time of batches of 1K elements in a PMQ with different time-windows λ. The number of elements in the container varies from Elts_min to Elts_max. The Mean, 99% and Max times are in ms.

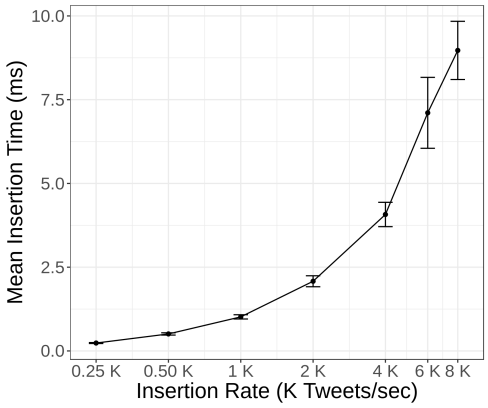


Figure 11: PMQ average insertion time with a window of $6h$ and varying rates. Current Twitter insertion rate (6K tweets/s) can be processed under 7.5 ms.

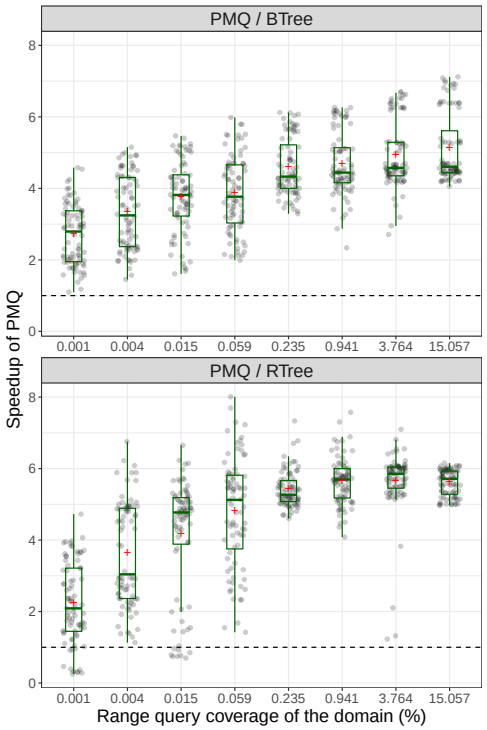


Figure 12: Range queries: PMQ speedup over the B-Tree and R-Tree. Each boxplot represents the speedup of throughput for each query instance. The average speedup is denoted by red crosses. PMQ is faster than the B-Tree in all cases. Compared to the R-Tree, PMQ has a speedup on 97% of queries tested. The cases where PMQ performs worst corresponds to queries returning a small number of elements compared to the dataset size.

6.5 Evaluating Range Queries

We evaluated the range query performance of the different data structures. We defined synthetic queries at varying sizes and different positions to simulate searches over the world map. Queries are defined using latitude and longitude coordinates over a rectangular map of the world using the Mercator projection. Each query is specified by its center latitude and longitude coordinates (lat, lon), where $lat \in \Re_{-90,+90}$ and $lon \in \Re_{-180,+180}$, and by its width $W = \{w \in \Re : w = \frac{90}{2^i} \wedge 0 \leq i < 8\}$. For each w in W, we randomly pick ten tweet coordinates from the dataset to generate a unique query. We discarded queries not fully contained on the world map. As a result, the query dataset we built has 80 queries. This set of queries was run over 8 different datasets at varying sizes from 1M to 128M elements, for a total of 640 query results. The size of the dataset is given by a parameter S defined as $\{1M \times 2^i : 0 \leq i < 8\}$. We performed each query 10 times and computed the average running time. Since we fixed the number of elements in memory, we computed the throughput of each query as the number of records returned by the query divided by the running time (in ms).

We compared the throughput by showing the speedup of PMQ over B-Tree and R-Tree (Figure 12). The boxplots in Figure 12 show the speedup grouped by w. The labels on the x-axis show the range query coverage relative to the total area of the domain. PMQ and B-Tree use the same querying mechanism based on the recursive space partitioning of a quadtree. In this case, PMQ is always better than the B-Tree, with speedups that can achieve up to 7× on the largest range queries. The speedup over the B-Tree is proportional to the number of elements returned by the query and is mainly due to the memory locality of PMQ. When the number of elements scanned to answer a query increases, the B-Tree has to access nodes scattered in memory locations, thus causing a poor usage of the cache memories. The average speedup increases with the range query size.

Since the R-Tree is a pointer-based data structure, its

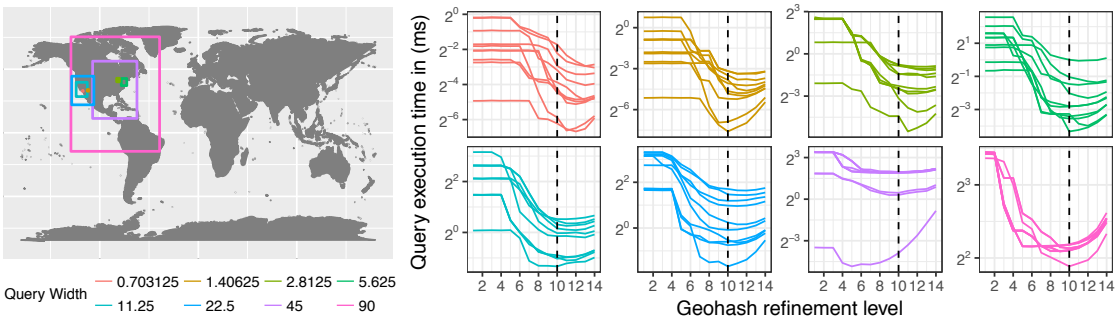


Figure 13: Examples of Range Queries. We define 8 different query widths. The B-Tree and PMQ use 10 levels of quadtree refinement for range queries. We choose this parameter experimentally to provide the best overall results.

internal nodes have bounding boxes containing the space occupied by its children, and only leaf nodes have the actual elements. Because of its internal index, the R-Tree is efficient for point queries, with good performance when a small amount of elements is queried. In a streaming dataset, elements are inserted individually in the data structure, as they arrive from the stream and without any specific ordering in memory. When the size of the range queries increases, the cost of scanning more elements hinders the throughput. The query algorithm uses a max depth parameter to limit the refinements done in the linear quadtree. The max depth of 10 used in this experiments was found to give the best results (for B-Tree and PMQ) as shown in Figure 13. The refined quadrants that do not fall entirely inside the queried region are scanned linearly to test the elements contained in the queried region. As a consequence, small range queries in regions with a high density of elements suffers from discontinuities in the Z-curve ordering. The throughput has a negative impact when the number of valid elements returned is low compared to the number of records scanned. Despite this, in our experiments the PMQ query algorithm outperforms the R-Tree (which does not rely on a Z-curve) by 5.5 times in 97% of the largest range queries.

6.6 Evaluating Top-k Queries

We compared the performance of top-k queries implemented on top of our PMQ against the `Kite` framework [26]. We generated 10K top-K queries from the check-in locations of the Brightkite social network [23]. Queries

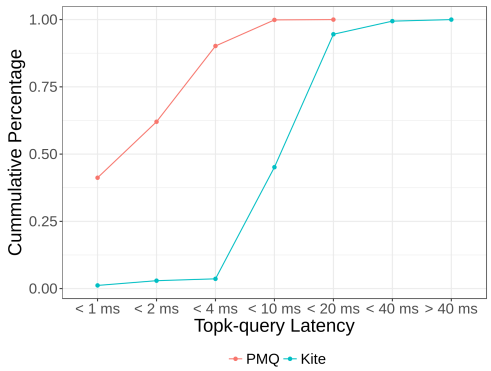


Figure 14: **Top-k Queries:** cumulative percentages of query latency for $K = 100$, $R = 30$ km and $T = 10000$ seconds. We compared the search performance of PMQ against the *Kite* framework.

correspond to users, in a given location, trying to find the most relevant tweets nearby.

At the moment of execution of the top-k queries, there were 10M tweets stored in PMQ. For each query, we measured the latency of accessing the storage array and computing the top-k elements. The top-k ranking function used the default parameters values $K = 100$, $R = 30km$ and $\lambda = 10000$ seconds. Temporal and spatial scores in PMQ were balanced with values 0.2 and 0.8 respectively. `Kite` does not offer a balancing parameter for the temporal and spatial dimensions, ranking elements with radius r merely according to the temporal dimension. Figure 14

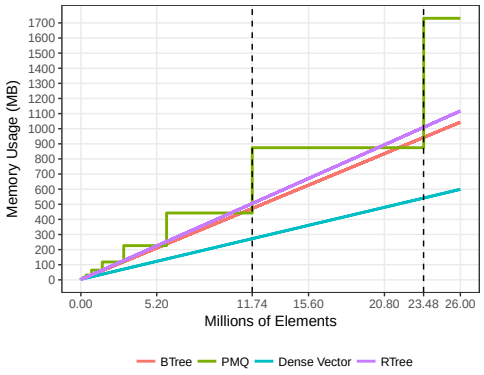


Figure 15: PMQ memory usage depends on density thresholds $2\rho_h < \tau_h$. `Boost C++` R-Tree has a bigger index overhead than `stx::btree`.

shows queries for different latency values. PMQ answers 90% of the queries in less than 4 ms while *Kite* can only process 3% of it. Kite uses a regular grid as a spatial index. Since the grid does not change to adapt to the complexity of the data, Kite does not perform well under scenarios of streaming datasets.

The data being sorted first based on their Z-index and next their timestamp, a pure time based query with no spatial constraint would need to scan all the elements of PMQ. We have seen that the PMQ shows a good scan performance (Fig. 8(b)). But if a majority of requests of this type are expected it may be advantageous to index data based on their timestamp first.

6.7 Evaluating Memory Usage

We measured the amount of resident set size used by each data structure individually (e.g., the physical memory used by the process code and data). The `Linux Kernel` maintains a pseudo-file system directory for each running process. By parsing `/proc/self/statm` we have access to the current resident set size in pages. We multiply this value by the page size from `sysconf()` to obtain the amount of used memory in bytes. We set the record size to `16 bytes`, the minimum space required to store spatial and timestamp metadata. Each batch inserts `1000 records` per iteration. For every iteration, we measured the current resident set size used by the data structures.

	Number of Elements stored	
Algorithm	11.7 M Elts	23.4 M Elts
Dense Vector	275.32 MB	545.97 MB
PMQ	882.97 MB	882.97 MB
B-Tree	477.00 MB	951.63 MB
R-Tree	510.42 MB	1019.15 MB

Table 4: Memory usage summary

Memory usage in a dense vector is directly proportional to the number of records, as shown in Figure 15, and serves as a baseline since this alternative does not have any storage overhead. PMQ memory usage depends on the max density parameter, τ_h (see Section 3.1). In our experiments τ_h is configured to 0.7, *i.e.*, the used memory slots correspond at maximum to 70% of slots allocated. As the number of elements in the data structures increases, PMQ doubles its size when the maximum density is reached (note the staircase-shaped curve of PMQ in Figure 15). In the experiments of Section 6.3, the maximum number of elements presented in memory was 23.488.000, which corresponds to the memory consumption show in Table 4.

6.8 Discussion of the Evaluation Results

The design of a data structure that is at the same time efficient for insertion/removal operations and large range queries requires careful analysis of trade-offs. Experiments have shown that PMQ offers a good tradeoff between both types of workloads. At a steady regime PMQ can perform efficient bulk removals, which are usually expensive in tree-based data structure because they require a full scan of the data.

The execution of queries in PMQ is substantially different from R-Trees because there are no index pointers to locate records. Instead the PMQ keeps data sorted based on Z-indices, and it suffices to use a fast range searching algorithm. We used the same Z-order in PMQ and B-Tree. However, as data is inserted and removed dynamically, a tree structure becomes fragmented in memory. PMQ avoids this issue by keeping the locality of its records along the Z-curve. Our experiments always verified that PMQ outperforms the B-Tree.

Some insertions in the PMQ can lead to higher execution times when a full rebalance is required: when the PMQ size needs to be doubled (Figure 7), or, with a lesser impact, at steady regime during bulk data removals (Fig-

ure 10). To mitigate the impact on query response time, one could rely on multithreading to overlap as much as possible rebalances with queries, adapting the approach proposed in [6] for the PMA.

7 Conclusion and Future Work

We introduced PMQ, a new data structure to keep sorted a stream of data that can fit in a controlled memory budget. PMQ reorganizes itself when needed with a low amortized number of data movements per insertion ($O(\log^2(N))$). Amongst the data structure compared, PMQ, B-Tree, and R-Tree, PMQ proved to have the best performance trade-off between insertion and searching times. Experiments showed that PMQ enables querying a continuously updated window with the latest arrived tweets in real-time. PMQ can maintain a significant amount of data in memory, filling the gap between stream processing engines working only on small windows of received stream, and other classical persistent storage solutions.

One direction for improvement would be to combine in-memory and persistent storage in a multi-level PMQ. The lazy stashing protocol might not adapt to some needs, as old data may stay a *long time* (up to the next top rebalancing) before being removed. We plan to develop a more reactive protocol for such situations. The current implementation imposes that every operation must acquire a thread lock before accessing or modifying PMQ. All requests are thus performed sequentially, which limits the number of transactions that PMQ can support.

8 Acknowledgment

The authors wish to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work was supported in part by *CNPq 308851/2015-3* and *CNPq 140313/2017-6*.

References

[1] Apache flink. `https://flink.apache.org`, 2017.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[3] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[4] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees, June9–11 2007.

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

[6] M. A. M. Bender, E. D. E. E. D. E. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. *SIAM Journal on Computing*, 35(2):341–358, jan 2005.

[7] M. Bern, D. Eppstein, and S.-H. Teng. *Parallel construction of quadtrees and quality triangulations*, pages 188–199. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[8] T. Bingmann. STX B+ Tree C++ Template Classes v0.9. `https://github.com/bingmann/stx-btree`, 2013.

[9] Boost. Geometry Index. `http://www.boost.org/`, 2017.

[10] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries, 2003.

[11] C. R. Cook and D. J. Kim. Best sorting algorithm for nearly sorted lists. *Commun. ACM*, 23(11):620–624, Nov. 1980.

[12] M. Durand, B. Raffin, and F. Faure. A Packed Memory Array to Keep Moving Particles Sorted, Dec. 2012.

[13] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992.

[14] W. Feng, C. Zhang, W. Zhang, J. Han, J. Wang, C. Aggarwal, and J. Huang. Streamcube: Hierarchical spatio-temporal hashtag clustering for event

exploration over the twitter stream. In *2015 IEEE 31st International Conference on Data Engineering, ICDE 2015*, pages 1561–1572, 2015.

[15] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon. Spatio-temporal indexing in non-relational distributed databases, Oct 2013.

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms, 1999.

[17] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, Dec. 1982.

[18] F. Goro. Timsort. `https://github.com/gfx/cpp-TimSort`, 2016.

[19] 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.

[20] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[21] A. Itai, A. G. Konheim, and M. Rodeh. A Sparse Table Implementation of Priority Queues, 1981.

[22] S. Kazemitabar, U. Demiryurek, M. Ali, A. Akdogan, and C. Shahabi. Geospatial stream query processing using Microsoft SQL Server StreamInsight. *Proc. of the VLDB Endowment*, 3(1-2):1537–1540, 2010.

[23] J. Leskovec and A. Krevl. Snap datasets: Stanford large network dataset collection, Mars 2017.

[24] 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.

[25] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data, 2013.

[26] A. Magdy and M. Mokbel. Kite. `http://kite.cs.umn.edu`, Mars 2017.

[27] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. *Proc. - International Conference on Data Engineering*, pages 172–183, 2014.

[28] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He. Venus: Scalable Real-Time Spatial Queries on Microblogs with Adaptive Load Shedding. *IEEE Transactions on Knowledge and Data Engineering*, 28(2), 2016.

[29] G. Mali, P. Michail, A. Paraskevopoulos, and C. Zaroliagis. *A New Dynamic Graph Structure for Large-Scale Transportation Networks*, pages 312–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[30] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller. Twitinfo, 2011.

[31] R. J. McGlinn. A parallel version of cook and kim's algorithm for presorted lists. *Softw. Pract. Exper.*, 19(10):917–930, Sept. 1989.

[32] C. Pahins, S. Stephens, C. Scheidegger, and J. Comba. Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–1, 2016.

[33] T. Peters. Timsort. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`, 2002.

[34] PostGIS. Spatial and geographic objects for postgresql. `http://postgis.net`, 2018.

[35] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the ub-tree into a database system kernel., 2000.

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

[37] SpatiaLite. Spatial and geographic objects for sqlite. `https://www.gaia-gis.it/fossil/libspatialite/index`, 2018.

[38] I. L. Stats. Twitter usage statistics. `http://www.internetlivestats.com/twitter-statistics`, Mars 2017.

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

[40] S.-E. Yoon and D. Manocha. Cache-efficient layouts of bounding volume hierarchies, 2006.

[41] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets, 2010.

[42] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

A Proof of the PMQ Amortized Cost

Let first identify an important property on windows densities after rebalance. A j-level window w_j is rebalanced when overfull ($d(w_j) > \tau_j$)). The rebalance occurs at the smallest underfull upper window w_l with $l > j$, i.e. the smaller one checking $d(w_l) < \tau_l$)). In worst case this is a top level rebalance requiring to double the PMQ size. After rebalance the density of w_j checks:

$$d(w_j) < \tau_l < \tau_j, \tag{6}$$

by Equation 1 of page 4. So w_j gets a density $d(w_j) < \tau_{j+1}$.

Now let consider a window w_j and let see how many insertions are necessary in this window so that it triggers a rebalance, i.e. it requires to rebalance the parent window w_{j+1}. We assume w_j just get rebalanced, thus $d(w_j) < \tau_{j+1}$ by Equation 6. The next rebalance triggered by w_j occurs once $d(w_j) > \tau_j$, i.e. after the insertion of

$$(\tau_j - \tau_{j+1})2^j K$$

elements where $K = O(\log(N))$ is the segment size.

Such rebalance requires to move $2^{j+1}K$ elements. If the rebalance occurs at the root window ($j = h$), the PMQ first makes a full scan of the PMQ to identify the data to be stashed. These data are next removed during the rebalance that is either performed on $w_h = O(N)$ if the new density is bellow τ_h or on a twice larger window after doubling the PMQ size. Thus a root rebalance cost is bounded by $2^{h+2}K$. We also need to count the cost of updating the accounting array. Each rebalance triggered by w_l leads to update $2^{j+2} - 1 + h - (j+1)$ elements of the accounting array.

Putting all these costs together, we have a cost associated to a rebalance triggered by w_j bounded by:

$$2^{j+2} + h - j - 2 + 2^{j+2}K < 2^{j+2}K + 2^{j+2} + \log(N).$$

This leads to the amortized cost per insertion of:

$$\begin{aligned}\frac{2^{j+2}K + 2^{j+2} + \log(N)}{(\tau_j - \tau_{j+1})2^j K} &< \frac{4K + 4 + \log(N)}{(\tau_j - \tau_{j+1})K}, \\ &= O(\log(N))\end{aligned}$$

by Equation 2 of page 4.

When an element is inserted into the PMQ, it actually contributes to the density of all enclosing windows from the segment up to the root, i.e. of $h = O(\log(N))$ windows. The amortized rebalance cost per insertion into the PMQ is thus $O(\log^2(N))$.

Each element needs to be inserted in the right place in the PMQ. If inserted one by one a binary sort is used with cost $O(\log(N))$. If inserted by batches, the insertion array is sorted with a cost per element that is also bounded by $O(\log(N))$. Added to the amortized rebalance cost, we get an unchanged total amortized cost per insertion of $O(\log^2(N))$.