K^2-Treaps: Range Top-k Queries in Compact Space *

Nieves R. Brisaboa[1], Guillermo de Bernardo[1], Roberto Konow[2,3], Gonzalo Navarro[2]

[1] Databases Lab., Univ. of A. Coruña, Spain, {brisaboa,gdebernardo}@udc.es
[2] Dept. of Computer Science, Univ. of Chile {rkonow, gnavarro}@dcc.uchile.cl
[3] Escuela de Informática y Telecomunicaciones, Univ. Diego Portales, Chile

Abstract. Efficient processing of top-k queries on multidimensional grids is a common requirement in information retrieval and data mining, for example in OLAP cubes. We introduce a data structure, the K^2-treap, that represents grids in compact form and supports efficient prioritized range queries. We compare the K^2-treap with state-of-the-art solutions on synthetic and real-world datasets, showing that it uses 30% of the space of competing solutions while solving queries up to 10 times faster.

1 Introduction

Top-k queries on multidimensional weighted point sets ask for the k heaviest points in a range. This type of query arises most prominently in data mining and OLAP processing (e.g., find the sellers with most sales in a time period) and in GIS applications (e.g., find the cheapest hotels in a city area), but also in less obvious document retrieval applications [16]. In the example of sales, one coordinate is the seller id, which are arranged hierarchically to allow queries for sellers, stores, areas, cities, states, etc., and the other is time (in periods of hours, days, weeks, etc.). Weights are the amounts of sales made by a seller during a time slice. Thus the query asks for the k heaviest points in some range $Q = [x_1, x_2] \times [y_1, y_2]$ of the grid.

Data mining and information systems such as those mentioned above usually handle huge amounts of data and may have to serve millions of queries per second. Representing this steadily increasing amount of data space-efficiently can make the difference between maintaining the data in main memory or having to resort to external memory, which is orders of magnitude slower.

We introduce a new compact data structure that performs fast range top-k queries on multidimensional grids and is smaller than state-of-the-art compact data structures. Our new representation, called K^2-treap, is inspired by two previous data structures: the K^2-tree [5] and the treap [18]. The K^2-tree is a

* Funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, by a Conicyt scholarship, by MICINN (PGE and FEDER) TIN2009-14560-C03-02 and TIN2010-21246-C02-01, by CDTI, MEC and AGI EXP 00064563/ITC-20133062, and by Xunta de Galicia (with FEDER) GRC2013/053.

compressed and self-indexed structure initially designed to represent Web graphs and later used in other domains as an efficient and compact representation of binary relations. The treap is a binary tree that satisfies the invariants of a binary search tree and a heap at the same time, which is useful for prioritized searches. Our results show that the K^2-treap answers queries up to 10 times faster, while using just 30% of the space, of state-of-the-art alternatives.

2 Basic Concepts

Rank and select on bitmaps. Let $B[1,n]$ be a sequence of bits, or bitmap. We define operations $rank_b(B,i)$ as the number of occurrences of $b \in \{0,1\}$ in $B[1,i]$, and $select_b(B,j)$ as the position in S of the jth occurrence of b. B can be represented using $n + o(n)$ bits [15], so that both operations are solved in constant time.

Wavelet trees and discrete grids. An $n \times m$ grid with n points, exactly one per column (i.e., x values are unique), can be represented using a *wavelet tree* [10, 12]. This is a perfect balanced binary tree of height $\lceil \lg m \rceil$ where each node corresponds to a contiguous range of values $y \in [1,m]$ and represents the points falling in that y-range, sorted by increasing x-coordinate. The root represents $[1,m]$ and the two children of each node split its y-range in half, until the leaves represent a single y-coordinate. Each internal node stores a bitmap, which tells whether each point corresponds to its left or right child. Using *rank* and *select* queries on the bitmaps, the wavelet tree uses $n \lg m + o(n \log m)$ bits, and can count the number of points in a range in $O(\log m)$ time, because the query is decomposed into bitmap ranges on at most 2 nodes per wavelet tree level. Any point can be tracked up (to find its x-coordinate) or down (to find its y-coordinate) in $O(\log m)$ time as well.

K^2-trees. The K^2-tree [5] is a data structure to compactly represent sparse binary matrices (which can also be regarded as point grids). The K^2 tree subdivides the matrix into K^2 submatrices of equal size. The submatrices are considered left-to-right and top-to-bottom, and each is represented with a bit, set to 1 if the submatrix contains at least one non-zero cell. Each node whose bit is 1 is recursively decomposed, subdividing its submatrix into K^2 children, and so on. The subdivision ends when a fully-zero submatrix is found or when we reach the individual cells. The K^2-tree can answer range queries with multi-branch top-down traversal of the tree, following only the branches that overlap the query range. While it has no good worst-case time guarantees, in practice times are competitive. The worst-case space, if t points are in an $n \times n$ matrix, is $K^2\, t \log_{K^2} \frac{n^2}{t}(1 + o(1))$ bits. This can be reduced to $t \lg \frac{n^2}{t}(1 + o(1))$ if the bitmaps are compressed. This is similar to the wavelet tree space, but in practice K^2-trees use much less space when the points are clustered.

The K^2-tree is stored in two bitmaps: T stores the bits of all the levels except the last one, in a level-order traversal, and L stores the bits of the last level (corresponding to individual cells). Given a node at position p in T, its

children are be located from position $rank_1(T, p) \cdot K^2$ in $T : L$. This property enables K^2-tree traversals using just T and L.

Treaps and Priority Search Trees. A *treap* [18] is a binary search tree with nodes having two attributes: *key* and *priority*. The treap maintains the binary search tree invariants for the keys and the heap invariants for the priorities, that is the key of a node is larger than those in its left subtree and smaller than those in its right subtree, whereas its priority is not smaller than those in its subtree. The treap does not guarantee logarithmic height, except on expectation if priorities are independent of keys [13]. The *priority search tree* [14] is somewhat similar, but it is balanced. In this case, a node is not the one with highest priority in its subtree, but that element is stored in addition to the element at the node. The element stored separately is also removed from the subtree. Priority search trees can be used to solve 3-sided range queries on n-point grids, returning t points in time $O(t + \log n)$.

3 Related Work

Navarro et al. [17] introduced compact data structures for various queries on two-dimensional weighted points, including range top-k queries. They enhance the bitmaps of each node as follows: Let $x_1, \ldots, x_r$ be the points represented at a node, and $w(x)$ be the weight of point x. Then a range maximum query (RMQ) data structure built on $w(x_1), \ldots, w(x_r)$ is stored together with the bitmap. Such a structure uses $2r + o(r)$ bits and finds the position of the maximum weight in any range $[w(x_i), \ldots, w(x_j)]$ in constant time [8] and without accessing the weights themselves. Therefore, the total space becomes $3n \lg m + o(n \log m)$ bits.

To solve top-k queries on a grid range $Q = [x_1, x_2] \times [y_1, y_2]$, we first traverse the wavelet tree to identify the $O(\log m)$ bitmap intervals where the points in Q lie (a counting query would, at this point, just add up all the bitmap interval lengths). The heaviest point in Q in each bitmap interval is obtained with an RMQ, but we need to obtain the actual priorities in order to find the heaviest among the $O(\log m)$ candidates. The priorities are stored sorted by x- or y-coordinate, so we obtain each one in $O(\log m)$ time by tracking the point with maximum weight in each interval. Thus a top-1 query is solved in $O(\log^2 m)$ time. For a top-k query we must maintain a priority queue of the candidate intervals, and each time the next heaviest element is found, we remove it from its interval and reinsert in the queue the two resulting subintervals. The total query time is $O((k + \log m) \log(km))$.

It is possible to reduce the time to $O((k+\log m) \log^\epsilon m)$ time and $O(\frac{1}{\epsilon} n \log m)$ bits, for any constant $\epsilon > 0$ [16], but the space usage is much higher, even if linear.

4 The K^2-treap

In one dimension, an RMQ structure using $2n+o(n)$ bits [8] is sufficient to answer range top-k queries in $O(k \log k)$ or $O(k \log \log n)$ time, using the algorithm just

described on a single interval. However, a similar RMQ structure for two or more dimensions needs $\Omega(mn \log m)$ bits [9] (on dense grids), and therefore it is better to directly look for representations of the data points that can also answer range top-k queries. The idea is to combine a K^2-tree with a treap data structure. If keys are $[1, n]$, treaps can be stored in $2n + o(n)$ bits plus the priorities [11], whereas priority search trees cannot. In two and more dimensions, however, this advantage vanishes. Therefore, our data structure combines the K^2-tree with a priority search tree, which is more convenient for its balancing guarantees.

4.1 Data Structure

Consider a matrix $M[n \times n]$ where each cell can either be empty or contain a weight in the range $[0, d-1]$. We consider a quadtree-like recursive partition of M into K^2 submatrices, the same performed in the K^2-tree with binary matrices. We build a conceptual K^2-ary tree similar to the K^2-tree, as follows: the root of the tree will store the coordinates of the cell with the maximum weight of the matrix, and the corresponding weight. Then the cell just added to the tree is marked as *empty*, deleting it from the matrix. If many cells share the maximum weight, we pick anyone of them. Then, the matrix is conceptually decomposed into K^2 equal-sized submatrices, and we add K^2 child nodes to the root of the tree, each representing one of the submatrices. We repeat the assignment process recursively for each child, assigning to each of them the coordinates and value of the heaviest cell in the corresponding submatrix and removing the chosen points. The procedure continues recursively for each branch until we find a completely empty submatrix (either because the matrix did not contain any weights in the region or because the cells with weights have been "emptied" during the construction process) or we reach the cells of the original matrix.

Fig. 1 shows an example of K^2-treap construction, for $K = 2$. At the top of the image we show the state of the matrix at each level of decomposition. $M0$ represents the original matrix, where the maximum value is highlighted. The coordinates and value of this cell are stored in the root of the tree. In the next level of decomposition (matrix $M1$) we find the maximum values in each quadrant (notice that the cell assigned to the root has already been removed from the matrix) and assign them to the children of the root node. The process continues recursively, subdividing each matrix into K^2 submatrices. The cells chosen as local maxima are highlighted in the matrices corresponding to each level of decomposition, except in the last level where all the cells are local maxima. Empty submatrices are marked in the tree with the symbol "–".

The data structure is represented in three parts: The location of local maxima, the weights of the local maxima, and the tree topology.

Local maximum coordinates: The conceptual K^2-treap is traversed levelwise, reading the sequence of cell coordinates from left to right in each level. The sequence of coordinates at each level ℓ is stored in a different sequence $coord[\ell]$. The coordinates at each level ℓ of the tree are transformed into an offset in the corresponding submatrix, transforming each c_i into $c_i \bmod (n/K^\ell)$

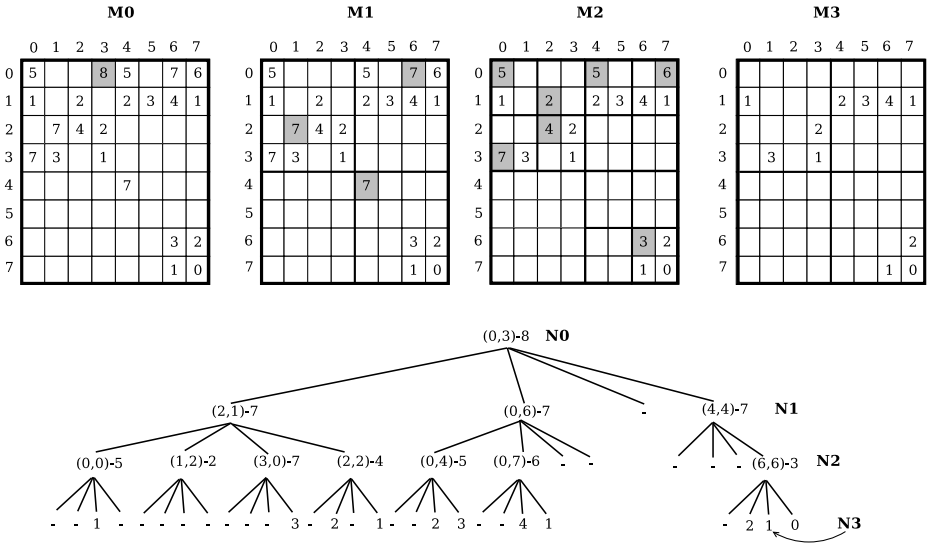

Fig. 1. Example of K^2-treap construction from a matrix

using $\lceil \lg(n) - \ell \lg K \rceil$ bits. For example, in Fig. 2 (top) the coordinates of node $N1$ have been transformed from the global value $(4, 4)$ to a local offset $(0, 0)$. In the bottom of Fig. 2 we highlight the coordinates of nodes $N0$, $N1$ and $N2$ in the corresponding *coord* arrays. In the last level all nodes represent single cells, so there is no *coord* array in this level. With this representation, the worst-case space for storing t points is $\sum_{\ell=0}^{\log_{K^2}(t)} 2K^{2\ell} \lg \frac{n}{K^\ell} = t \lg \frac{n^2}{t}(1 + O(1/K^2))$, that is, the same as if we stored the points using the K^2-tree.

Local maximum values: The maximum value in each node is encoded differentially with respect to the maximum of its parent node. The result of the differential encoding is a new sequence of non-negative values, smaller than the original. Now the K^2-treap is traversed level-wise and the complete sequence of values is stored in a single sequence named *values*. To exploit the small values while allowing efficient direct access to the array, we represent *values* with Direct Access Codes [4]. Following the example in Fig. 2, the value of node $N1$ has been transformed from 7 to $8 - 7 = 1$. In the bottom of the figure the complete sequence *values* is depicted. We also store a small array $first[0, \lg_K n]$ that stores the offset in *values* where each level starts.

Tree structure: We separate the structure of the tree from the values stored in the nodes. The tree structure of the K^2-treap is stored in a K^2-tree. Fig. 2 shows the K^2-tree representation of the example tree, where only cells with value are labeled with a 1. We will consider a K^2-tree stored in a single bitmap T with *rank* support, that contains the sequence of bits from all the levels of the tree. Our representation differs from a classic K^2-tree (that uses two bitmaps T and L and only adds rank support to T) because we will need to perform rank operations also in the last level of the tree. The other difference is that

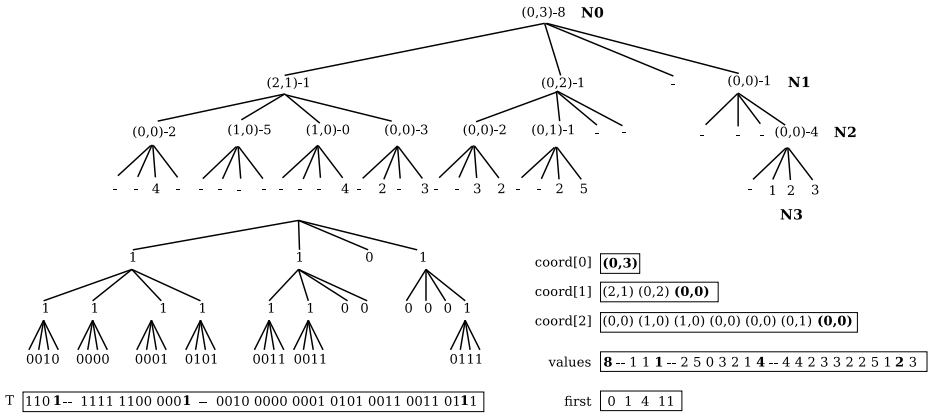

Fig. 2. Storage of the conceptual tree in our data structures

points stored separately are removed from the grid. Thus, in a worst-case space analysis, it turns out that the space used to represent those explicit coordinates is subtracted from the space the K^2-tree would use, therefore storing those explicit coordinates is free in the worst case.

4.2 Query algorithms

Basic navigation. To access a cell $C = (x, y)$ in the K^2-treap we start by accessing the K^2-tree root. The coordinates and weight of the element stored at the root node are $(x_0, y_0) = coord[0][0]$ and $w_0 = values[0]$. If $(x_0, y_0) = C$, we return w_0 immediately. Otherwise, we find the quadrant where the cell would be located and navigate to that node in the K^2-tree. Let p be the position of the node in T. If $T[p] = 0$ we know that the complete submatrix is empty and return immediately. Otherwise, we need to find the coordinates and weight of the new node. Since only nodes set to 1 in T have coordinates and weights, we compute $r = rank_1(T, p)$. The value of the current node will be at $values[r]$, and its coordinates at $coord[\ell][r - first[\ell]]$, where ℓ is the current level. We rebuild the absolute value and coordinates, w_1 as $w_0 - values[r]$ and (x_1, y_1) adding the current submatrix offset to $coord[\ell][r - first[\ell]]$. If $(x_1, y_1) = C$ we return w_1, otherwise we find again the appropriate quadrant in the current submatrix where C would be located, and so on. The formula to find the children is identical to that of the K^2-tree. The process is repeated recursively until we find a 0 bit in the target submatrix, we find a 1 in the last level of the K^2-tree, or we find the coordinates of the cell in an explicit point.

Top-k queries. The process to answer top-k queries starts at the root of the tree. Given a range $Q = [x_1, x_2] \times [y_1, y_2]$, the process initializes an empty max-priority queue and inserts the root of the K^2-tree. The priority queue stores, in general, K^2-tree nodes sorted by their associated maximum weight. Now, we iteratively extract the first priority queue element (the first time this is the root).

If the coordinates of its maximum element fall inside Q, we output it as the next answer. In either case, we insert all the children of the extracted node whose submatrix intersects with Q, and iterate. The process finishes when k results have been found or when the priority queue becomes empty (in which case there are less than k elements in Q).

Other supported queries. The K^2-treap can also answer basic range queries (i.e., report all the points that fall in Q). This is similar to the procedure on a K^2-tree, where the submatrices that intersect Q are explored in a depth-first manner. The only difference is that we must also check whether the explicit points associated to the nodes fall within Q, and in that case report those as well. Finally, we can also answer *interval queries*, which ask for all the points in Q whose weight is in a range $[w_1, w_2]$. To do this, we traverse the tree as in a top-k range query, but we only output weights whose value is in $[w_1, w_2]$. Moreover, we discard submatrices whose maximum weight is below w_1.

5 Experiments and Results

To test the efficiency of our proposal we use several synthetic datasets, as well as some real datasets where top-k queries are of interest. Our synthetic datasets are square matrices where only some of the cells have a value set. We build different matrices varying the following parameters: the *size* $s \times s$ of the matrix (s = 1024, 2048, 4096, 8192), the number of different weights d in the matrix (16, 128, 1024) and the *percentage* p of cells that have a point (10, 30, 50, 70, 100%). The distribution of the weights in all the datasets is uniform, and the spatial distribution of the cells with points is random. For example, the synthetic dataset with $(s = 2048, d = 128, p = 30)$ has size 2048×2048, 30% of its cells have a value and their values are follow a uniform distribution in $[0, 127]$.

We also test our representation using real datasets. We extracted two different views from a real OLAP database storing information about sales achieved per store/seller each hour over several months: *salesDay* stores the number of sales per seller per day, and *salesHour* the number of sales per hour. Huge historical logs are accumulated over time, and are subject to data mining processes for decision making. In this case, finding the places (at various granularities) with most sales in a time period is clearly relevant. Table 1 shows a summary with basic information about the real datasets. For simplicity, in these datasets we ignore the cost of mapping between real timestamps and seller ids to rows/columns in the table, and assume that the queries are given in terms of rows and columns.

We compare the space requirements of the K^2-treap against the solution based on wavelet trees enhanced with RMQ structures [17] introduced in Section 3 (*wtrmq*). Since our matrices can contain none or multiple values per column, we transform our datasets to store them using wavelet trees. The wavelet tree will store a grid with as many columns as values we have in our matrix, in column-major order. A bitmap is used to map the real columns with virtual ones: we append a 0 per new point and a 1 when the column changes. Hence,

Dataset	#Sellers (rows)	Time instants (columns)	Number of diff. values	K^2-treap (bits/cell)	$mk2tree$ (bits/cell)	$wtrmq$ (bits/cell)
$SalesDay$	1314	471	297	2.48	3.75	9.08
$SalesHour$	1314	6028	158	1.06	0.99	3.90

Table 1. Real datasets used, and space required to represent them

range queries in the $wtrmq$ require a mapping from real columns to virtual ones (2 $select_1$ operations per query), and the virtual column of each result must be mapped back to the actual value (a $rank_1$ operation per result).

We also compare our proposal with a representation based on constructing multiple K^2-trees, one per different value in the dataset. In this representation ($mk2tree$), top-k queries are answered by querying consecutively the K^2-tree representations for the higher values. Each K^2-tree representation in this proposal is enhanced with multiple optimizations over the simple bitmap approach we use, like the compression of the lower levels of the tree using DACs (see [5] for a detailed explanation of this and other enhancements of the K^2-tree).

All bitmaps that are employed use a bitmap representation that supports $rank$ and $select$ using 5% of extra space. The wtmrq was implemented using a pointer-less version of the wavelet tree [7] with a RMQ implementation that requires 2.38 bits per value. For all experiments we use $K = 2$ for the K^2-treap and $mk2tree$.

We ran all our experiments on a dedicated server with 4 Intel(R) Xeon(R) E5520 CPU cores at 2.27GHz 8MB cache and 72GB of RAM memory. The machine runs Ubuntu GNU/Linux version 9.10 with kernel 2.6.31-19-server (64 bits) and gcc 4.4.1. All data structures were implemented in C/C++, compiled with full optimizations.

5.1 Space Comparison

We start by comparing the compression achieved by the representations. As shown in Table 1, the K^2-treap overcomes the $wtrmq$ in the real datasets studied by a factor over 3.5. The $mk2tree$ representation is competitive with the K^2-treap and even obtains slightly less space in the dataset $salesHour$, taking advantage of the relatively small number of different values in the matrix.

The K^2-treap also obtains the best space results in most of the synthetic datasets studied. Only in the datasets with a very small number of different values ($d = 16$) the $mk2tree$ uses less space than the K^2-treap. Notice that, since the distribution of values and cells is uniform, the synthetic datasets are close to a worst-case scenario for the K^2-treap and $mk2tree$. To provide additional insight on the compression capabilities, Fig. 3 provides a summary of the space results for some of the synthetic datasets used. The left plot shows the evolution of compression with the size of the matrix. The K^2-treap is almost unaffected by the matrix size, as its space is around $t \lg \frac{s^2}{t} = s^2 \frac{p}{100} \lg \frac{100}{p}$ bits, that is, constant

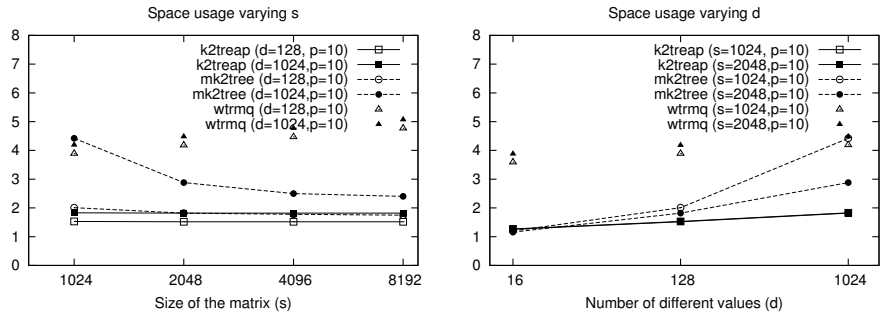

Fig. 3. Evolution of the space usage with s and d in the synthetic datasets, in bits/cell (in the right plot, the two results for the K^2-treap are on top of each other)

per cell as s grows. On the other hand, the *wtrmq* uses $t \lg s = s^2 \frac{p}{100} \lg s$ bits, that is, its space per cell grows logarithmically with s. Finally, the *mk2tree* obtains poor results in the smaller datasets but is more competitive on larger ones (some enhancements in the K^2-tree representations behave worse in smaller matrices). Nevertheless, notice that the improvements in the *mk2tree* compression stall once the matrix reaches a certain size.

The right plot of Fig. 3 shows the space results when varying the number of different weights d. The K^2-treap and the *wtrmq* are affected only logarithmically by d. The *mk2tree*, instead, is sharply affected, since it must build a different K^2-tree for each different value: if d is very small the *mk2tree* representation obtains the best space results also in the synthetic datasets, but for large d its compression degrades significantly.

As the percentage of cells set p increases, the compression in terms of bits/cell (i.e., total bits divided by s^2) will be worse. However, if we measure the compression in bits per point (i.e., total bits divided by t), then the space of the *wtrmq* is independent of p ($\lg s$ bits), whereas the K^2-treap and *mk2tree* use less space as p increases ($\lg \frac{100}{p}$). That is, the space usage of the *wtrmq* increases linearly with p, while that of the K^2-treap and *mk2tree* increases sublinearly. Over all the synthetic datasets, the K^2-treap uses from 1.3 to 13 bits/cell, the *mk2tree* from 1.2 to 19, and the *wtrmq* from 4 to 50 bits/cell.

5.2 Query Times

In this section we analyze the efficiency of top-k queries, comparing our structure with the *mk2tree* and the *wtrmq*. For each dataset, we build multiple sets of top-k queries for different values of k and different spatial ranges (we ensure that the spatial range is at least of size k). All query sets are generated for fixed k and w (side of the spatial window). Each query set contains 1000 queries where the spatial window is placed at a random position within the matrix.

Fig. 4 shows the time required to perform top-k queries in some of our synthetic datasets, for different values of k and w. The K^2-treap obtains better

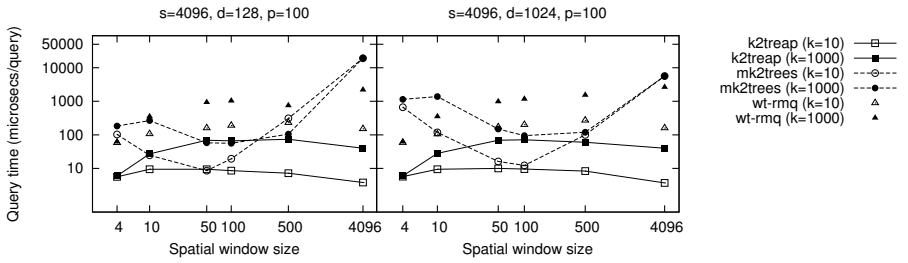

Fig. 4. Times of top-k queries in synthetic datasets

query times than the *wtrmq* in all the queries, and both evolve similarly with the size of the query window. On the other hand, the *mk2tree* representation obtains poor results when the spatial window is small or large, but it is competitive with the K^2-treap for medium-sized ranges. This is due to the procedure to query the multiple K^2-tree representations: for small windows, we may need to query many K^2-trees until we find k results; for very large windows, the K^2-treap starts returning results in the upper levels of the conceptual tree, while the *mk2tree* approach must reach the leaves; for some intermediate values of the spatial window, the K^2-treap still needs to perform several steps to start returning results, and the *mk2tree* representation may find the required results in a single K^2-tree. Notice that the K^2-treap is more efficient when no range limitations are given (that is, when $w = s$), since it can return after exactly K iterations. Fig. 4 only shows the results for two of the datasets, but similar comparison results have been obtained in all the synthetic datasets studied, with the K^2-treap outperforming the alternative approaches in most of the cases, except in some queries with medium-sized query windows, when the *mk2tree* can obtain slightly better query times.

Next we perform a set of queries that would be interesting in our real datasets. We start with the same $w \times w$ queries as before, which filter a range of rows (sellers) and columns (days/hours). Fig. 5 shows the results of these range queries. As we can see, the K^2-treap outperforms both, the *mk2tree* and *wtrmq*, in all cases. Similarly to the previous results, the *mk2tree* approach also obtains poor query times for small ranges but is better in larger ranges.

We run two more specific sets of queries that may be interesting in many datasets, and particularly in our examples: "column-oriented" and "row-oriented" range queries, that only restrict one of the dimensions of the matrix. Row-oriented queries ask for a single row (or a small range of rows) but do not restrict the columns, and column-oriented ask for single columns. We build sets of 10,000 top-k queries for random rows/columns with different values of k. Fig. 6 (left) shows that in column-oriented queries the *wtrmq* is faster than the K^2-treap for small values of k, but our proposal is still faster as k grows. The reason for this difference is that in "square" range queries, the K^2-treap only visits a small set of submatrices that overlap the region; in row-oriented or column-oriented

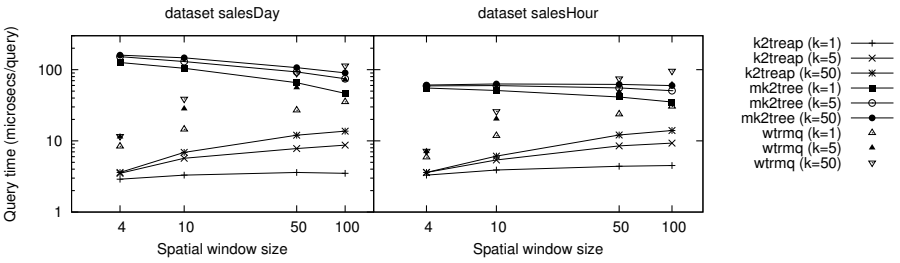

Fig. 5. Query times of top-k queries in real datasets

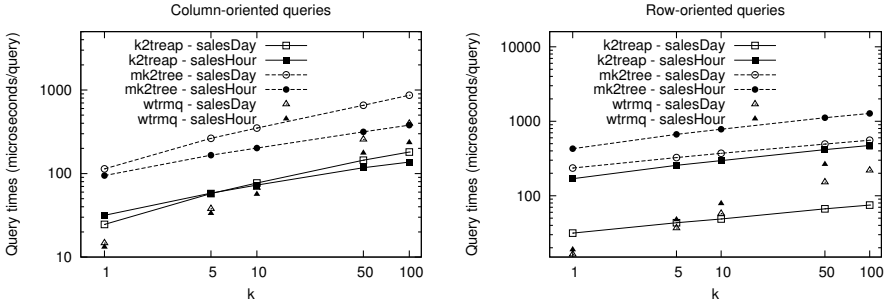

Fig. 6. Query times of row-oriented and column-oriented top-k queries

queries, the K^2-treap is forced to check many submatrices to find only a few results. The $mk2tree$ suffers from the same problem of the K^2-treap, being unable to filter efficiently the matrix, and obtains the worst query times in all cases.

In row-oriented queries (Fig. 6, right) the $wtrmq$ is even more competitive, obtaining the best results in many queries. The reason for the differences found between row-oriented and column-oriented queries in the $wtrmq$ is the mapping between real and virtual columns: column ranges are expanded to much longer intervals in the wavelet tree, while row ranges are left unchanged. Notice anyway that our proposal is still competitive in the cases where k is relatively large.

6 Conclusions and Future Work

We have introduced a new compact data structure that performs top-k range queries on grids up to 10 times faster than current state-of-the-art solutions and requires as little as 30% of the space, both in synthetic and real OLAP databases, and including uniform distributions, which is the worst scenario for K^2-treaps.

The K^2-treap can be generalized to represent grids in higher dimensions, by simply replacing our underlying K^2-tree with its generalization to d dimensions, the K^d-tree [3] (not to be confused with kd-trees [2]). The algorithms stay identical, but an empirical evaluation is left for future work. In the worst case, a grid

of t points on $[n]^d$ will require $O(t \lg \frac{n^d}{t})$ bits, which is of the same order of the data, and much less space will be used on clustered data. Instead, an extension of the wavelet tree will require $O(n \log^d n)$ bits, which quickly becomes impractical. Indeed, any structure able to report the points in a range in polylogarithmic time requires $\Omega(n(\log n/\log\log n)^{d-1})$ words of space [6], and with polylogarithmic space one needs time at least $\Omega(\log n(\log n/\log\log n)^{\lfloor d/2 \rfloor - 2})$ [1]. As with top-k queries one can report all the points in a range, there is no hope to obtain good worst-case time and space bounds in high dimensions, and thus heuristics like K^d-treaps are the only practical approaches (kd-trees do offer linear space, but their time guarantee is rather loose, $O(n^{1-1/d})$ for n points on $[n]^d$).

References

1. Afshani, P., Arge, L., Larsen, K.G.: Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. In: Proc. SCG. pp. 323–332 (2012)
2. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Comm. ACM 18(9), 509–517 (1975)
3. de Bernardo, G., Álvarez-García, S., Brisaboa, N.R., Navarro, G., Pedreira, O.: Compact querieable representations of raster data. In: Proc. 20th SPIRE. pp. 96–108 (2013)
4. Brisaboa, N., Ladra, S., Navarro, G.: DACs: Bringing direct access to variable-length codes. Inf. Proc. Manag. 49(1), 392–404 (2013)
5. Brisaboa, N., Ladra, S., Navarro, G.: Compact representation of web graphs with extended functionality. Inf. Sys. 39(1), 152–174 (2014)
6. Chazelle, B.: Lower bounds for orthogonal range searching I: The reporting case. J. ACM 37(2), 200–212 (1990)
7. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Proc. 15th SPIRE (2008)
8. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comp. 40(2), 465–492 (2011)
9. Golin, M., Iacono, J., Krizanc, D., Raman, R., Rao, S.S.: Encoding 2D range maximum queries. In: Proc. 22nd ISAAC. pp. 180–189 (2011)
10. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA. pp. 841–850 (2003)
11. Konow, R., Navarro, G., Clarke, C., López-Ortíz, A.: Faster and smaller inverted indices with treaps. In: Proc. 36th SIGIR. pp. 193–202 (2013)
12. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: Proc. 7th LATIN. pp. 703–714 (2006)
13. Martínez, C., Roura, S.: Randomized binary search trees. J. ACM 45(2), 288–323 (1997)
14. McCreight, E.M.: Priority search trees. SIAM J. Comp. 14(2), 257–276 (1985)
15. Munro, I.: Tables. In: Proc. 16th FSTTCS. pp. 37–42 (1996)
16. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: Proc. 23rd SODA. pp. 1066–1078 (2012)
17. Navarro, G., Nekrich, Y., Russo, L.: Space-efficient data-analysis queries on grids. Theor. Comp. Sci. 482, 60–72 (2013)
18. Seidel, R., Aragon, C.: Randomized search trees. Algorithmica 16(4/5), 464–497 (1996)