

## RESEARCH ARTICLE

# An experimental evaluation of $k^2$ -tree on external memory

Gilberto Gutiérrez<sup>1</sup> | Miguel Romero<sup>1</sup> | Miguel R. Penabad<sup>2</sup> | Fernando Santolaya<sup>1</sup> |  
Mónica Caniupán<sup>1</sup> | Rodrigo Torres<sup>1</sup>

<sup>1</sup>Facultad de Ciencias Empresariales, Universidad del Bío-Bío, Chile

<sup>2</sup>Centro de investigación CITIC, Universidade da Coruña, Spain

## Correspondence

Corresponding author: Miguel R. Penabad.

Email: miguel.penabad@udc.es

## Abstract

The  $k^2$ -tree compact data structure has gained great popularity to represent binary relationships in main memory. It presents a good performance and a good trade-off between storage and execution time. However, datasets being too large, or limited resources, may prevent the dataset from fitting into RAM even in compressed form. This work presents an experimental evaluation of a  $k^2$ -tree in external memory (disk), in terms data access (I/O operation or cache misses) and execution time for 4 types of common queries. We compare the  $k^2$ -tree with other data structures, namely a Quadtree (specifically, a Linear QuadTree, LQT) and the classical adjacency matrix, all of them being in external memory. We used for the test both synthetical as well as large, real world, datasets. Several aspects, such as the size of the memory cache and its replacement scheme, or specific parameters like the arity ( $k$  value) of the  $k^2$ -tree were considered in the experiments. In terms of storage needs, the  $k^2$ -tree clearly outperforms the other alternatives, in extreme cases needing only 1% of the LQT space, or 0.01% of the adjacency matrix (and for some large datasets neither the LQT nor the adjacency matrix could be built). In terms of performance, except for cases with small datasets, where the adjacency matrix is the best option for some queries, the  $k^2$ -tree is either competitive or outperforms the alternatives.

## KEYWORDS

External memory, compact data structures,  $k^2$ -tree

## 1 | INTRODUCTION AND PROBLEM DEFINITION

Typically, the different types of memory make up a hierarchy of levels that range from CPU registers, L2 cache, RAM, secondary memory (online, usually disks) and tertiary memory (offline, such as tapes)<sup>1</sup>. Generally speaking, the memories closer to the CPU are faster, but smaller and more expensive, while secondary or tertiary memories are slower, cheaper and with higher capacity. In particular, accessing (either reading or writing) an item from RAM is orders of magnitude faster than accessing secondary memory. That can be from hundreds or thousands of times faster, in the case of solid state devices, to millions, in the case of the classic HDD magnetic disks. Although technological developments have increased the speed and capacity while decreasing the cost, we expect the speed difference between main memory and disk will remain in the future. It is most probable that we will keep dealing with fast, small and expensive memories and large capacity memories that are slower and cheaper.

In the cost model for secondary memory (we will use from now on secondary memory or disk interchangeably) one of the main components of the processing time is the number of Input/Output (I/O) operations, that is, the number of page reads or writes needed to process the data. It is common to use buffers, such as the OS buffer cache, that store disk blocks or pages in RAM. Together with smart replacement techniques they can minimize the number of I/O operations. The locality of reference, that is, accessing items located in nearby memory positions, is a key element to minimize the processing cost. In general, in the cost model for secondary memory, data structures are encouraged to guarantee a high locality of reference in order to be efficient.

To efficiently use the main memory space we can use Compact Data Structures (CDSs), which are structures that store data in compressed form, thus reducing storage, and are able to process the data without having to first decompress them<sup>2,3,4</sup>. That capability allows us to process bigger datasets in main memory. Moreover, depending on the size of the datasets, CDSs may

allow to store them in higher levels of the memory hierarchy (closer to the CPU), thus further improving the overall performance of the CDSs.

During the last decades, CDSs have attracted much attention, both in academia and in the industry. They have been used in a variety of data types, as integers arrays (for example, see<sup>5</sup>), text (for example, see<sup>6,7,8</sup>), and binary relations. In<sup>9</sup> the authors describe the  $k^2$ -tree, a CDS originally devised to represent web graphs, but useful to represent any binary relationship. In the field of search and information retrieval, classical CDSs include the Wavelet trees<sup>6,10,11</sup>, Compressed Suffix Array or CSA<sup>7,8</sup> and the FM-Index<sup>12</sup>. Enhancing spatial queries in Geographical Information Systems with CDSs was addressed in<sup>13,10</sup>. Moreover, in<sup>14,15,16</sup> the authors propose *succinct Quadrees* to encode points and support queries over them, and in<sup>17</sup> they are used to store line segments and compute queries efficiently. More recently, compact data structures have also been used to solve classical geometric problems<sup>18</sup>. The book<sup>3</sup> has an excellent review on compact data structures.

Although CDSs make a great effort to reduce the storage costs and allow to process large data sets directly in main memory, the volume of information increases on a daily basis, and the necessity to use secondary memory becomes unavoidable. Just the huge amount of data generated by the Internet of Things or the social networks are a real proof of this statement. An additional example is, in the field of astronomy, the telescope networks that generate datasets of terabytes per hour<sup>19</sup>. Consequently, it will be very common that the datasets that applications must process will be too large to be stored and processed in main memory<sup>1</sup>, even in its compressed form.

A well designed CDS, with a good use of locality of reference when accessing the data, should be efficient enough, even when not all the information is stored in main memory and secondary memory must be used. The goal of this work is develop a version of the  $k^2$ -tree<sup>9</sup> that can use secondary memory to store its information. We aim at preserving the advantages of both the  $k^2$ -tree (reducing the storage needs, the information is self-indexed) and the secondary memory (persistence, greater storage capacity, lower prices).

As a summary, the main contributions of this work are the following:

- An implementation of the  $k^2$ -tree compact data structure that resides in external memory, and can be accessed without cache, or having a cache with LRU or LFU buffer replacement schemes (code publicly available at <https://gitlab.lbd.org.es/lbd-open/k2tree-disk>).
- A detailed theoretical evaluation of the cost of the most common operations over  $k^2$ -trees.
- A comprehensive empirical evaluation of the  $k^2$ -tree, comparing it with a Linear Quadtree and an adjacency matrix, all of them residing in external memory.

The rest of the paper is organized as follows. Sections 2 and 3 introduce some preliminary concepts and a review of the related work. Section 4 describes our proposal: a  $k^2$ -tree that can use external memory to store and query the information. Section 5 shows the empirical results of the experiments we conducted. Finally, Section 6 offers some concluding remarks and directions for future work.

## 2 | BACKGROUND

In this section, we introduce foundational concepts pertaining to binary relations and outline the quadtrees and  $k^2$ -trees, data structures designed for the representation of binary relations or spatial data.

Formally, a binary relation  $R$  is defined as a subset of the Cartesian product between two sets  $A$  and  $B$ , denoted as  $R \subseteq A \times B$ . Numerous problems can be represented or modeled through binary relationships (friendship relationships, web graphs of pages pointing to other pages, among others). Typically, these relationships are represented using graphs (directed/undirected), trees, inverted lists, or most basic approaches such as a binary matrix. In this case, the relationship can be seen as a set of points in a two dimensional space.

### 2.1 | Quadtree

In general terms, a quadtree<sup>20</sup> is a tree-like data structure used to represent spatial information (thus, it is also capable of representing any binary relationship). It works by hierarchically decomposing the space. There are several variations of this data structure, the most well known being the region quadtree.

Consider a binary matrix representing the space, where each cell is either a 1 or a 0. A region quadtree partitions the space into 4 equal sized regions (quadrants: NW, NE, SW and SE). If all values in a quadrant are not of the same value, it is recursively subdivided in further subquadrants, possibly up to cell (pixel) level. If all cells share the same value, the node is left undivided. Thus, the quadtree is represented by a nonbalanced 4-ary tree.

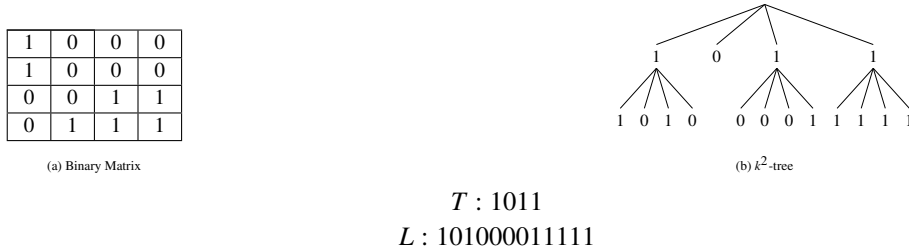
There are also different ways to represent a region quadtree. One of the most popular is the Linear Region Quadtree, or simply Linear Quadtree (LQT)<sup>21</sup>, which basically assigns a code to each leaf of the quadtree based on its path from the root node of the conceptual tree. We will use the implementation of the LQT in<sup>21</sup> for our experiments.

## 2.2 | $k^2$ -tree

The  $k^2$ -tree compact data structure was initially designed to compactly represent a web graph<sup>2</sup>, but it is able to represent any binary relationship or point in a (discrete) 2-D space. Actually, with  $k = 2$ , a  $k^2$ -tree can be seen as a compact version of a region quadtree.

The  $k^2$ -tree is represented by a bitmap, which is created as follows. The building process divides the matrix into  $k^2$  equal-sized submatrices, from left to right and from top to bottom. Each submatrix full of zeroes is assigned a 0, and it is not further decomposed. If a submatrix contains at least a 1, it is recursively subdivided into  $k^2$  submatrices. The final level (the individual cells) are represented in the bitmap with its cell value. The final representation is the breadth-first traversal of this conceptual tree.

For instance, consider Figure 1 to create a  $k^2$ -tree with  $k = 2$ . The NE quadrant is full of zeroes, so its node gets a 0. The remainder quadrants have ones, so they get a 1 and they are recursively decomposed. The bitmap  $T$  is the breadth-first traversal of the internal tree nodes, and the  $L$  bitmap corresponds to the values at leaf level. The  $k^2$ -tree is represented just using the concatenation of  $T$  and  $L$ .



**FIGURE 1** A binary matrix, its  $k^2$ -tree conceptual representation, and the  $T$  and  $L$  bitmaps used to store it.

Being a compact data structure, the  $k^2$ -tree does not include pointers in its representation. In order to navigate it, the most important primitive would be to access the children of a node. The  $getChild(TL, i)$  function obtains, in a  $k^2$ -tree represented by  $TL$ , the first child of a node located at offset  $j$  in the bitmap (the siblings of this children would be the following bits). We define  $getChild(TL, j) = rank_1(TL, j) \cdot k^2$ . The  $rank_1(TL, j)$  call returns the number of bits set to 1 in the bitmap, from its beginning up to offset  $j$ .

For example (be aware that the bitmap starts at offset 0), the first child of the third child of the root (node 2) is  $getChild(TL, 2) = rank_1(TL, 2) \cdot k^2 = 2 \cdot 2^2 = 8$ .

The  $rank_1()$  method is very frequently used in compact data structures. Extending the bitmap with additional structures that occupy  $o(n)$  space on the length of the bitmap, it can be executed in constant time ( $O(1)$ ).

Considering a  $k^2$ -tree that stores a relationship  $R \subseteq A \times B$ , or a region in a 2D space, there are some fundamental types of queries we can pose against it:

- $CheckLink(a, b)$  returns true if the element  $a$  is related to the element  $b$  in  $k^2$ -tree that stores the relationship. Sometimes this operation is named  $Access(a, b)$ , and it would be useful to test whether the point  $(a, b)$  exists, if the  $k^2$ -tree is used to represent spatial information.
- $DirectNeighbors(a)$  returns the set of elements  $\{b_i \in B | (a, b_i) \in R\}$ . If the  $A$  elements are the  $Y$  axis and the  $B$  elements are on  $X$ , this operation would retrieve all the ones in the row of  $a$ .

- $ReverseNeighbors(b)$  returns the set of elements  $\{a_i \in A | (a_i, b) \in R\}$ , or the ones in the column of  $b$ .
- $Range(a_1, b_1, a_2, b_2)$  is a range or window query, which requires that the elements are sorted by some criteria. It would retrieve all pairs  $(a_i, b_i)$  that exist in this range. Viewing the information as a region in a 2D space, it would retrieve all points in the selected window. In our experiments we will also use  $Range\langle N \rangle(a_1, b_1)$ , which would be equivalent to  $Range(a_1, b_1, a_1 + N, b_1 + N)$

### 3 | EXTERNAL MEMORY

Although compact data structures use lower amount of memory and allow us to store larger datasets in main memory, there are many situations where datasets will not fit completely into the main memory. The rapid growth of data in many fields, specially those related to the Internet of Things (IoT), social networks, or Big Data in general, is a reason for this. Another reason is the use of smaller devices, such as embedded systems, with very limited RAM and CPU resources.

In these cases, when all data cannot fit into main memory, we can resort to the use of external (secondary) memory, usually hard drives. In order to empirically evaluate data structures and algorithms to manipulate them over external memory, we need a cost model. Vitter<sup>1</sup> defined PDM (Parallel Disk Model), a model that bases its performance measurements in three aspects: the number of I/O operations, the space used in disk by the data structure, and CPU efficiency, basically the time needed to answer a query over the data structure. PDM makes some simplifications, such as considering that all I/O operations have essentially the same cost, and it does not distinguish between random or sequential I/O operations. In this regard, Arge<sup>22</sup> had suggested that reading index structures, which implies random I/O operations, can be slower than accessing more data blocks in sequential reads.

Another important aspect to consider is the hard drive technology, if we want to consider the time needed for the algorithms and data structures to complete an operation. The latency of RAM is usually some nanoseconds (about 15 nanoseconds in DDR4 memories). For classic, magnetic hard drives, it can be some milliseconds (so, 6 orders of magnitude slower), but for SSD or NVMe devices can be as low as 250 microseconds, which is just 2 orders of magnitude slower than RAM. Additionally, SSD devices have large asymmetrical latencies between reads and writes, due to the nature of writes. In order to write (program) a block or SSD page, the target page must be empty, so it may need to be erased before the write. Other problems that degrade the efficiency of SSDs are write amplifications (having to erase and program a larger number of pages than strictly needed for a write) or interleaving read and write operations. Regarding the use of SSDs for algorithms and data structures, there is an excellent survey of the state of the art on<sup>23</sup>. This work includes a proposal to adapt spatial indexes on external memory to SSDs, specially adapting the eFIND<sup>24</sup> and FAST<sup>25</sup> frameworks.

In the case of our work, we will be conducting experiments that perform only searches, that imply read operations, but no writes. Thus, the latter observations will not affect our measurements. Therefore, we will base our measurements on Vitter's model, basing it on the number of I/O operations, the space taken up by the data structures, and the time use to complete the queries.

Finally, an aspect that can greatly affect the efficiency, mainly because it affects the number of I/O operations, is the main memory cache used to cache the previously accessed external memory pages. The size of the cache, as well as the implemented page replacement technique (LRU: Least Recently Used or LFU: Least Frequently Used, among others) are parameters that must be considered in the evaluation.

### 4 | OUR PROPOSAL: $k^2$ -tree IN EXTERNAL MEMORY

In this section we describe the design and implementation of the  $k^2$ -tree on external memory. We also offer a theoretical analysis of the performance of the 4 main types of queries posed over  $k^2$ -trees.

#### 4.1 | Implementation Design

Figure 2 shows the classes and their relationships that are considered in the implementation of the  $k^2$ -tree in secondary memory. The *Storage* class implements read and write operations of a sequence of pages (objects of the *page* class) directly to/and from the disk. The *Storage* class allows reading and updating a page randomly and inserting new pages at the end of the sequence. The *LFUCacheStorage* and *LRUCacheStorage* classes extend the *Storage* class by implementing the LFU (Least Frequently

Used) and LRU (Least Recently Used) page/block replacement policies, respectively. This design allows I/O operations to be performed taking into account or not one of the replacement policies.

The *ExternalBitVect* class implements a bitmap in secondary memory. This class knows and manages the structure of the pages in such a way that it can represent the bits of a bitvector in a sequence of these pages, and efficiently perform the *Rank()* and *Access()* operations. The data structure for storing a bitvector is simple and based on the proposal in<sup>26</sup> which divides the bitvector into pages. Since the *ExternalBitVect* class is essential in our design of implementation, we provide more details below.

Let  $w$  be the size in bits of a word and  $D$  the number of words in a page, so  $D \cdot w$  is the size of a page in bits. On each page, a word is allocated, which we call  $h$ , to store the number of bits set to 1 in the bitmap, up to the previous page (that would correspond to the value of the *Rank* operation). Thus  $h = 0$  for the first page. This value is used to calculate *Rank* at any position on the page without accessing previous ones. With the structure given to the pages it is very easy to implement the *Access* and *Rank* operations. For *Access*( $j$ ),  $b = \lfloor j / ((D - 1) \cdot w) \rfloor$  returns the page number  $b$  of the sequence, where the  $j^{\text{th}}$  bit is located. Then, the page  $b$  is retrieved and through simple operations the value of the  $j^{\text{th}}$  bit is obtained. In a similar way, we can solve *Rank*<sub>1</sub>( $j$ ), that is, the page  $b$  is obtained and retrieved and the number of bits set to 1, up to the  $j^{\text{th}}$  bit, is added to  $b \cdot h$ . Note that both operations require only one I/O operation. Later in this section, we show that by using block replacement techniques, it is eventually possible to avoid I/O operations.

The *ExternalK2Tree* class is implemented using the *ExternalBitVect* class to store on disk the bitvector resulting from the concatenation of the bitvectors  $T$  and  $L$  ( $T:L$ ) that compactly encode a  $k^2$ -tree. The  $k^2$ -tree operations are implemented in the *ExternalK2Tree* class in the same way in which is done in its original version (main memory)<sup>9</sup>. This is possible because the *ExternalBitVect* class encapsulates the complexity to manage I/O operations. The theoretical analysis of the main operations is carried out in Section 4.2.

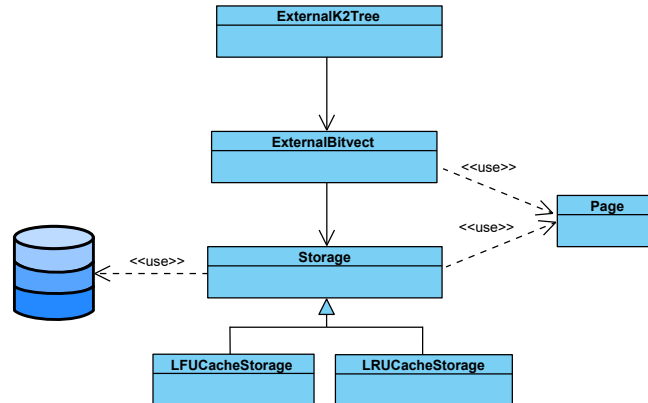


FIGURE 2 Implementation architecture of the  $k^2$ -tree in secondary memory

## 4.2 | Temporal Theoretical Analysis

Our proposal has 4 methods to query the structure, which are analyzed by the amount of access to secondary memory. Some observations about the use of cache are also included. For the analysis, let us assume that the  $k^2$ -tree represents a binary relation  $R$  between sets  $A$  and  $B$ , such that  $n = k^x \geq \max(|A|, |B|)$ ,  $m = |R|$  and every internal node has  $k^2$  children. Also, let us assume  $c$  as the size of cache, and recall that  $P = D \cdot w$  ( $D$  words of size  $w$ ) is the size of a page in secondary memory. Also, in the following complexity analysis, the best case scenario is not considered, as it is always to query nodes with value equal to 0, needing just one page of secondary memory (a 0 means that the node has no children, so the query can return immediately).

## 4.2.1 | CheckLink

Given two elements  $a, b$ , we want to know if they are in the binary relation represented on the  $k^2$ -tree. We will consider only the case where the pair  $(a, b)$  exists in the  $k^2$ -tree, where CheckLink needs to calculate  $\log_k(n)$  children operations, visiting all the  $\log_k(n)$  levels of the conceptual tree. In other case, if the pair does not exist, CheckLink would eventually find a 0 node and return.

Additionally, we can study two scenarios depending on the number of ones in the nodes: the worst case would be when the nodes are filled with ones, but the expected case would have a mixture of zeroes and ones.

**Worst Case Scenario:** Since  $T:L$  is in secondary memory we have to retrieve its first page, which stores the the first  $Lev$  levels. The worst case (maximum number of pages) would be when all the internal nodes up to level  $Lev$  are 1 (they have children), which would be

$$\sum_{i=0}^{Lev} k^{2i} = P.$$

Therefore, we can assume that  $Lev = \Theta(\log_k P)$ . As the amount of internal nodes grows with levels, we can assume that from level  $Lev$  beyond, every page contains (part of) one level.

Then, the number of access to secondary memory is

$$T_W(n, m, k, P) = \log_k(n) - \Theta(\log_k P).$$

**Expected Case Scenario:** If the data is clustered and the density of the data is low, we can expect that the amount of levels on the first page to be higher. Let us suppose that  $d < k^2$  is the average amount of nodes with value 1 per level, which clearly depends on the density and clustering of the data. As such, the first page contains the first  $Lev_d$  levels, such that

$$\sum_{i=0}^{Lev_d} d^i \cdot k^2 = P.$$

Therefore, we can assume that  $Lev_d = \Theta(\log_d \frac{P}{k^2})$ . Again, as the amount of internal nodes grows with levels, we can assume that from level  $Lev_d$  beyond, every page contains (part of) one level.

Also, we can expect to visit a fraction of the levels, as the leaf can be higher than the last level. Then, number of access to secondary memory is

$$T_E(n, m, k, P, d) = \Theta(\log_k(n)) - \Theta\left(\log_d \frac{P}{k^2}\right).$$

## 4.2.2 | Direct and reverse neighbors

Direct and reverse neighbors (also known as Successors and Predecessors) work in the same manner in  $k^2$ -tree, so the analysis is the same.

**Worst Case Scenario:** The worst case scenario is having to report a row (or column) where it is needed to get to the last level of the conceptual tree for every element. So, in every level, exactly  $k$  nodes are visited. As such, the amount of visited nodes are

$$\sum_{i=0}^{\log_k n} k^i = \Theta(n).$$

Now, we know that in a worst case scenario, the number of levels on the first page is  $Lev = \Theta(\log_k P)$ . Therefore, the amount of the visited nodes on the first page is

$$\sum_{i=0}^{Lev} k^i = \Theta(P).$$

Again, as the amount of internal nodes grows with levels, we can assume that from level  $Lev$  beyond, every page contains (part of) one level. Then, the worst case scenario calls

$$T_W(n, m, k, P) = \Theta(n - P)$$

pages.

**Expected Case Scenario:** In<sup>9</sup>, an analysis of a non-clustered scenario, it was expected to visit  $O(\sqrt{m})$  nodes. As this is an uniform distribution of the data, it is expected that a big fraction of the nodes in the first few levels must be accessed and, as before, we expect that  $Lev = \Theta(\log_k P)$  levels are in the first called page. At such, the amount of consulted nodes in the first page is  $O(P)$ .

Again, as the amount of internal nodes grows with levels, we can assume that from level  $Lev$  beyond, every page contains (part of) one level. Then, the expected case scenario gets

$$T_E(n, m, k, p) = O(\sqrt{m} - P)$$

pages. If the data is clustered, the performance is indeed better than under an uniform distribution.

### 4.2.3 | Range

Let us use  $occ$  as the number of reported relations inside range  $r = p \times q$ .

**Worst Case Scenario:** The worst case scenario is that the algorithm needs to get the page with the first levels, every  $occ$  is in the last level of the conceptual tree, and no internal node is shared between them after the first levels on the first page. Therefore:

$$T_W(n, m, k, P, occ) = occ \cdot (\log_k(n) - \Theta(\log_k P)) + 1.$$

**Expected Case Scenario:** In<sup>3</sup>, the author studied the amount of visited nodes in an expected range query. The visited nodes are separated in two types: nodes whose area is inside the range, called type 1, and nodes whose area intersects (but it is not completely inside) the range, called type 2.

The amount of type 1 nodes (considering its empty children) is proven to be

$$O\left(occ \cdot k \cdot \log_k \frac{p \cdot q}{occ}\right).$$

The amount of type 2 nodes (considering its empty children, but excluding its type 1 children) is proven to be

$$O(k \cdot \log_k n + p + q).$$

Now, only type 1 nodes whose parents are type 2 nodes are expected to be alone in a page, as children are stored continuously. In that case, the expected amount of accessed pages containing type 1 nodes are

$$O\left(\frac{occ \cdot k \cdot \log_k \frac{p \cdot q}{occ}}{P}\right).$$

Type 2 nodes are in the border of the range, so an upper bound is to have one node per page in the lower levels, and the amount of query nodes on the first few levels is expected to be almost linear (1 node per level), so the amount of type 2 nodes in the first page on an expected case is the already calculated  $Lev_d = \Theta(\log_d \frac{P}{k^2})$ . Therefore, the expected amount of accessed pages containing type 2 nodes are

$$O\left(k \cdot \log_k n + p + q - \log_d \frac{P}{k^2}\right).$$

And then, the amount of access to secondary memory is expected to be

$$\begin{aligned} T_E(n, m, k, P, d) &= \\ O\left(\frac{occ \cdot k \cdot \log_k \frac{p \cdot q}{occ}}{P} + k \cdot \log_k n + p + q - \log_d \frac{P}{k^2}\right) &= \\ O(p + q + (occ/P) \cdot k \cdot \log_k n). \end{aligned}$$

### 4.2.4 | Use of cache

Using cache benefits our structure greatly when performing consecutive queries. Indeed, as the memory required to store the  $k^2$ -tree is orders of magnitude less than other secondary memory data structures, the probability of a cache hit is much higher than with other structures.

It is important to note that, for taller trees, a LFU strategy could not be so beneficial as a LRU strategy, as using LFU would imply to have only the upper pages of the structure on cache, so every time a query needs to dig into the lower pages, it is likely that those pages are not on cache and they are immediately replaced.

## 5 | EXPERIMENTS

We present in this section a series of experiments to test the performance of  $k^2$ -trees on external memory. We have three different scenarios: First, we compare external memory  $k^2$ -trees with a Quadtree. Specifically, we have used the Linear Quadtree (LQT) implemented in<sup>21</sup>. Then, we compare the  $k^2$ -tree with a classical data structure used to represent binary relations: an adjacency matrix, also using external memory. Finally, we evaluate the behavior of the external memory  $k^2$ -tree by itself, considering the impact of variations of some parameters, such as the value of  $k$ . We used large real datasets to perform these last experiments.

All the coding was done in C++, and compiled with GNU g++ 11.4.0. The implementations of the external memory  $k^2$ -tree and adjacency matrix are our own, and are publicly available<sup>†</sup>. The LQT implementation was borrowed from Corral et al.<sup>21</sup>.

The experiments were run in two different machines, both having 16GB of RAM and using Ubuntu 22.04 as operating system. The first one was a laptop with an AMD Ryzen 7 5800H processor, and an SSD disk capable of reading at 1196.27 MB/s. The second one was a server with an Intel(R) Xeon(R) CPU E3-1225 v6 @ 3.30GHz processor and a magnetic HDD with a read speed of 199.32 MB/s.

The experiments recorded many different variables, but we will focus on two of them for the discussion in this work: cache misses and time to execute the query. A cache miss occurs when the algorithm tries to read a page and it is not in the cache, so it leads to an I/O operation, which is one of the predominant factors when dealing with external memory. Timing is also important, and it is greatly affected by the hardware architecture of the system running the code.

For the external memory, it is obvious that the disk technology has a great impact on the timings. However, the technology is not so important when comparing the relative behavior of the different algorithms. We compared the times for 6 operations using HDD (magnetic disk) and SSD devices, showing the results in Table 1. The *Time* column is the average time (for a batch of 50 queries) taken to complete the query, and *CMiss* is the average number of cache misses, that is, the number of pages that were not found in the  $k^2$ -tree cache. Taking reverse neighbor queries (the slowest one) as base (100%), we can see that, even when the time values are different, the percentages are almost identical for HDD and SSD, so the efficiency of the algorithm is the same regardless of the disk technology. For this reason, the rest of the results shown in this paper will include only one disk technology, namely the SSD.

Another decision we had to make was whether to use the operating system buffer cache. We decided to use it, like many applications (including relational database management systems) do: use an internal cache plus the operating system buffer cache. In any case, we performed some tests using the `O_DIRECT` flag when accessing the files, to bypass the operating system buffer cache, and the results were consistent with those presented in this paper. Works like<sup>1,27</sup> indicate that the efficiency of algorithms that use external memory are typically measured by the number of I/O operations (specially when, like our case, the total time is dominated by the I/O time). Therefore, even when we also show timings in our results, we consider the cache misses (where a searched page in our internal cache is not found and it must be transferred from disk/operating system buffer cache) the main measure of the efficiency of our algorithms.

We take into account other parameters that affect the behavior of the algorithms:

- The type and size of the cache. We considered three types: Cacheless, LRU, and LFU. We decided to establish the page size as 1KB for the smallest datasets and 4KB for the larger ones, and measure the cache size in number of pages.
- The  $k$  parameter of the  $k^2$ -tree. We have considered values 2, 4, and 8. This parameter affects the size of the  $k^2$ -tree (and thus the achieved compression ratio) and its height.

For the execution of the queries we considered an initial phase, what we called *warm-up*, which consists on posing a certain number of CheckLink queries over the data structure. The objective of this phase is to avoid (for timing purposes) the bias that can be introduced by the Operating System buffer cache, which can have a (positive or negative) impact on the time taken up by the query.

The procedure to execute each batch of queries (all of the same type) is the following:

<sup>†</sup> Available at <https://gitlab.lbd.org.es/lbd-open/k2tree-disk>

Query	HDD			SSD		
	Time( $\mu$ s)	CMiss	Time %	Time( $\mu$ s)	CMiss	Time %
CheckLink	60.88	4.54	0.06%	37.66	4.54	0.06%
Range100	77.00	3.74	0.07%	46.00	3.74	0.08%
Range1000	285.94	3.10	0.27%	169.76	3.10	0.29%
Range10000	10,732.82	5.26	10.13%	6,418.54	5.26	11.03%
Direct	32,743.22	374.80	30.89%	18,146.58	374.80	31.18%
Reverse	105,996.66	1,753.40	100.00%	58,206.38	1,753.40	100.00%

**TABLE 1** Performance of HDD and SSD over *road-network* dataset,  $k^2$ -tree ( $k = 4$ ), LRU (cache size=100 pages). Time is the average time taken up by the query, and CMiss is the average number of cache misses, for a batch of 50 queries.

1. The data structure cache is emptied.
2. The warm-up queries are executed.
3. The batch of queries is executed.

Each batch consisted of a number of queries that ranges from 50 to 1,000, depending on the dataset size. The reported query time (in  $\mu$ s) to complete the query, as well as the cache misses (CMiss), correspond to the average of every batch of queries.

For the CheckLink query batches, 50% of the queries used a point that belonged to the dataset, and 50% did not. Range queries were created by generating a random point and building a rectangular window around it, of the specified size (testing that the window will fit into the matrix boundaries). For direct and reverse neighbor queries, random values, without duplicates, were generated, in the range from 0 to  $n - 1$  (being  $n$  the matrix size).

## 5.1 | $k^2$ -tree vs Linear Quadtree and adjacency matrix

In this section, we compare the  $k^2$ -tree with the linear quadtree (LQT) and the adjacency matrix using generated synthetic data sets. The chosen implementation of the LQT<sup>21</sup> presents some constraints on the number of points it can index, as well as the used cache (only LRU, not LFU) and query operations (only CheckLink is supported).

We created datasets with sizes ranging from (roughly) 52,000 to 838,000 points, with uniform and gaussian distributions. We considered two matrix sizes: 1,024 and 2,048, and densities ( $m/n^2$ , being  $m$  the number of points and  $n$  the matrix size) of 5%, 10% and 20%.

The tests include the CheckLink query with no cache, and with LRU. In this case, the query batches were composed of 100 queries, both for the warm-up queries and for the real CheckLink queries.

Table 2 shows the disk space required by the studied data structures. As we expected, all versions of the  $k^2$ -trees require less space than the adjacency matrix and especially less than the LQT (mainly due to the usage of pointers in the LQT). This happens for all matrix sizes and all distributions of the points in the space. For example, with  $k = 4$ , the  $k^2$ -tree uses in average between 1.0% and 2.0% of the LQT space, and between 24% and 42% of the space used by the adjacency matrix. We can also see that the  $k^2$ -tree compression ratio is worse (it uses more space) when the value of  $k$  is larger, and also when considering the uniform distribution (which is less clustered than gaussian).

Tables 3 (without any cache) and 4 (cache using LRU replacement scheme) show the performance of the three data structures in terms of disk page accesses, to evaluate the CheckLink query. We can see in Table 3 that the adjacency matrix has much less accesses than LQT or  $k^2$ -tree; about 25% of the LQT and 30% of the  $k^2$ -tree. This is due to the matrix using always exactly one I/O operation to answer the query (in this case, a whole row of the matrix fits into one page), while the  $k^2$ -tree, in the worst case, must descend the tree (which has a height of  $O(\log_k n)$ ) until it reaches its leaves. This is also the reason for the performance of the  $k^2$ -tree to be improved when  $k$  increases, because the height of the  $k^2$ -tree is reduced.

Table 4 shows the behavior of the data structures using a cache with an LRU replacement scheme (the chosen implementation for the LQT does not include LFU). In this case we considered the cache misses (the average cache misses considering 100

Matrix	Num.	Gaussian distribution					Uniform distribution				
		Size	Point	LQT	$k = 2$	$k = 4$	$k = 8$	Adj.	LQT	$k = 2$	$k = 4$
1024	52,429	2,785	27	31	40	130	2,963	53	81	127	130
	104,858	4,452	37	40	48	130	5,936	82	114	131	130
	209,715	4,865	55	53	61	130	11,812	119	135	132	130
2048	209,715	11,142	106	123	159	517	11,871	210	324	505	517
	419,430	17,904	149	157	191	517	23,635	326	455	524	517
	838,861	19,687	218	213	245	517	47,249	474	536	525	517

TABLE 2 Size in KB of data structures: LQT,  $k^2$ -tree ( $k = 2, k = 4, k = 8$ ) and Adjacency matrix.

queries), which would force an I/O operation to retrieve the needed page from disk. We evaluated the CheckLink query with a cache of 5, 10 and 20 pages. As expected, all data structures benefited from the use of cache, and of course the benefit increases as the cache size does.

In general, all three data structures are very competitive under this cache scheme, but it is more significant in the case of the  $k^2$ -tree: it reduces its I/O operations to about a 20%, in average, of those needed for a cacheless scheme. This is easily explained by the reduced space of the  $k^2$ -tree (due to the compression), which allows to hold most of the  $k^2$ -tree in the cache.

The effect of the data distribution can be seen again in this table: the  $k^2$ -tree performs better for the gaussian distribution, because having the data more clustered the  $k^2$ -tree achieves a greater compression ratio. We can also see in this scenario, as it happened in the cacheless scheme (Table 3), the positive effect of the arity ( $k$  value) on the  $k^2$ -tree performance.

Matrix	Num.	Gaussian distribution					Uniform distribution				
		Size	Point	LQT	$k = 2$	$k = 4$	$k = 8$	Adj.	LQT	$k = 2$	$k = 4$
1024	52,429	3.8	2.4	1.8	1.3	1.0	4.0	4.2	2.7	2.5	1.0
	104,858	3.9	2.5	1.8	1.5	1.0	4.0	4.3	2.8	2.5	1.0
	209,715	3.9	3.1	2.1	1.8	1.0	4.0	4.6	2.9	2.6	1.0
2048	209,715	3.9	3.4	2.1	2.1	1.0	4.0	5.2	3.3	2.9	1.0
	419,430	3.9	3.6	2.3	2.1	1.0	4.0	5.4	3.5	2.9	1.0
	838,861	3.9	3.9	2.5	2.4	1.0	5.0	5.6	3.5	2.9	1.0

TABLE 3 Average number of Disk Accesses (SSD) with cacheless scheme: LQT,  $k^2$ -tree ( $k = 2, k = 4, k = 8$ ) and Adjacency matrix.

## 5.2 | $k^2$ -tree vs adjacency matrix

The chosen implementation for the Quadtree (LQT) had some limitations on the input size, so we conducted another series of experiments considering only  $k^2$ -trees and adjacency matrices with larger datasets than those of Section 5.1. Specifically, we used sets of 1, 10, and 100 million points, with a matrix size of  $2^{18} \times 2^{18}$ . Two synthetic datasets were generated, using gaussian and uniform distributions. Regarding the space needed for the data structures, show in Table 5, we can highlight that the adjacency matrix takes up the same space regardless of the data distribution and number of points (it is a binary matrix, so each cell will have a 0 or a 1). As we expected, the  $k^2$ -tree requires much less space, only between 0.04% and 10.5% of the space needed for the adjacency matrix, they use more space when we increase its  $k$  parameter, and obtain higher compression ratios for the gaussian distribution.

For the experiments we considered batches of 50 queries, both for the warm-up phase and the real experiments where the time and cache misses were measured. We report these data only for the LRU replacement scheme because LFU consistently showed worse behavior for both data structures. In this case, all query types defined in Section 2.2 can be considered.

Matrix Size	Num. Point	Cache Size	Gaussian distribution					Uniform distribution					
			LQT	$k = 2$	$k = 4$	$k = 8$	Adj.	LQT	$k = 2$	$k = 4$	$k = 8$	Adj.	
1024	52,429	5	2.00	0.52	0.36	0.40	0.58	2.00	2.06	0.98	0.93	0.86	
		10	2.00	0.00	0.00	0.00	0.30	2.00	0.55	0.59	0.79	0.72	
		20	1.00	0.00	0.00	0.00	0.07	2.00	0.00	0.01	0.47	0.39	
	104,858	5	2.00	0.77	0.42	0.54	0.66	2.00	2.22	1.03	0.85	0.85	
		10	2.00	0.00	0.00	0.07	0.29	2.00	1.26	0.76	0.70	0.70	
		20	2.00	0.00	0.00	0.01	0.08	2.00	0.10	0.26	0.38	0.45	
	209,715	5	2.00	1.41	0.62	0.67	0.72	2.00	2.09	0.99	0.90	0.86	
		10	2.00	0.47	0.23	0.33	0.48	2.00	1.42	0.78	0.76	0.75	
		20	1.00	0.00	0.00	0.01	0.16	2.00	0.58	0.43	0.39	0.41	
	2048	209,715	5	2.00	2.31	1.23	0.90	0.89	2.00	3.15	1.74	1.08	0.96
			10	2.00	1.50	0.77	0.64	0.78	2.00	2.73	1.45	0.95	0.94
			20	2.00	0.46	0.43	0.34	0.61	2.00	1.71	1.01	0.88	0.88
419,430		5	2.00	2.61	1.37	0.87	0.88	2.00	3.35	1.76	1.03	0.97	
		10	2.00	1.70	0.82	0.70	0.76	2.00	2.96	1.59	0.96	0.91	
		20	2.00	0.78	0.39	0.33	0.55	2.00	2.19	1.16	0.85	0.84	
838,861	5	2.00	2.59	1.38	0.99	0.93	3.00	3.23	1.72	1.05	0.99		
	10	2.00	1.86	1.06	0.83	0.86	3.00	2.89	1.45	0.99	0.95		
	20	2.00	1.14	0.61	0.67	0.78	2.00	2.27	1.12	0.94	0.87		

TABLE 4 Average number of Cache Misses (SSD) for CheckLink using LRU.

Num. Points	Gaussian distribution					Uniform distribution			
	Adj.	$k = 2$	$k = 4$	$k = 8$	Adj.	$k = 2$	$k = 4$	$k = 8$	
1,000,000	8,405,028	3,216	5,940	14,728	8,405,028	3,836	7,208	17,464	
10,000,000	8,405,028	24,008	43,056	99,772	8,405,028	30,232	55,696	138,988	
100,000,000	8,405,028	159,724	269,616	612,840	8,405,028	221,192	391,988	881,028	

TABLE 5 Size in KB of data structures: Adjacency matrix and  $k^2$ -tree ( $k = 2, k = 4, k = 8$ ).

Tables 6 and 7 compare the behavior of both data structures for the uniform distribution. We can highlight from Table 6 that all operations run faster in the  $k^2$ -trees, except for CheckLink. In this case, the adjacency matrix performs better (from 2 to 10 times faster), for all cache sizes and for all values of  $k$ . As in the previous subsection, this can easily be explained: for the current dataset sizes, a row of the matrix fits into one disk page, so  $CheckLink(x, y)$  can be resolved by accessing just the page that contains the row  $x$ , while the  $k^2$ -tree must be navigated until an empty node or a leaf is reached. For the rest of the operations, the  $k^2$ -tree far exceeds the adjacency matrix. As an example, consider the  $k^2$ -tree with  $k = 4$ . For the direct and reverse neighbor queries,  $k^2$ -trees take from 0.8% to 8% of the time taken by the adjacency matrix for the same queries. The performance difference is larger if we consider range queries, reaching 0.03% in average.

The differences in execution times are usually consistent with differences on cache misses. For example, for CheckLink, the adjacency matrix has always less cache misses and its execution time is always lower. However, there is a special case: the direct neighbor queries. In this case, the adjacency matrix has a lower cache miss number than the  $k^2$ -trees, but always a (much) larger execution time. This can be explained by the fact that, in order to solve the query, the adjacency matrix must iterate over all the bits that correspond to the matrix row that the direct neighbor query has to examine. This needs  $O(n)$  (in our case,  $2^{18}$ ) access operations at bit level (which include some shifting and bit-wise operations). Comparing this behavior with the reverse neighbor queries, we can see that the cache misses (and time execution) are again consistent, and extremely larger than the values for direct neighbors. The adjacency matrix is stored “row by row”, so direct neighbor queries explore a contiguous space on the

data structure, but the reverse queries (that retrieve a “column”) need to access separate pages of the data structure, even one access per needed bit (the matrix size is  $2^{18} = 262,144$ , which is exactly the number of cache misses). This explains the large difference in the behavior of the adjacency matrix for direct and reverse neighbor queries. Conversely, the compression and indexing capabilities of the  $k^2$ -tree makes it perform better than the adjacency matrix, and with almost no difference between the two types of queries.

Query	Cache size	Adj. Mat.		$k^2$ -tree (k=2)		$k^2$ -tree (k=4)		$k^2$ -tree (k=8)	
		Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss
CheckLink	10	4.4	1.0	42.4	7.1	21.4	3.3	15.3	2.5
	50	3.7	1.0	39.6	5.4	20.2	2.6	14.3	2.1
	100	8.2	1.0	36.7	4.4	32.0	2.3	15.0	1.9
	500	5.1	1.0	41.0	3.9	24.9	2.3	16.4	1.9
	1000	7.4	1.0	43.4	3.9	28.8	2.3	16.5	1.9
Range100	10	24,384.8	202.2	80.5	8.2	60.6	3.4	125.1	2.9
	50	22,548.0	202.2	75.4	5.7	58.7	2.7	127.0	2.1
	100	22,393.9	202.2	75.4	3.9	59.8	2.1	127.9	1.8
	500	22,509.1	202.2	74.0	3.0	60.2	2.0	144.0	1.8
	1000	22,471.7	202.2	75.0	3.0	60.1	2.0	131.2	1.8
Range1000	10	2,161,077.6	2,122.6	4,943.6	86.7	6,271.3	10.6	7,363.8	7.3
	50	2,173,870.0	2,122.6	4,585.3	14.5	6,278.6	9.5	7,339.7	6.1
	100	2,152,514.9	2,122.6	4,650.5	11.8	6,327.1	7.9	7,374.8	5.0
	500	2,153,782.6	2,122.6	4,563.7	5.2	6,278.7	4.3	7,446.7	3.7
	1000	2,158,048.3	2,122.6	4,558.1	5.2	6,306.2	4.3	7,502.2	3.7
Range10000	10	215,530,665.2	32,235.7	291,245.7	3,282.8	294,816.3	46.4	568,516.8	62.9
	50	216,027,756.2	32,235.7	269,851.6	48.1	285,869.9	43.6	563,148.1	62.3
	100	217,120,979.2	32,235.7	271,773.0	42.4	287,523.7	39.4	562,507.9	59.2
	500	215,400,043.4	32,235.7	272,298.4	12.5	297,420.4	14.9	572,982.3	44.3
	1000	213,717,614.0	32,235.7	272,472.8	10.6	289,909.2	11.8	585,732.5	23.8
Direct	10	70,689.8	4.5	8,135.2	164.5	5,615.2	78.5	5,697.2	71.4
	50	70,220.9	4.5	7,835.4	98.0	5,437.4	78.3	5,438.0	71.3
	100	70,630.2	4.5	7,819.3	91.1	5,393.5	75.5	5,453.7	70.4
	500	70,507.0	4.5	7,811.3	44.1	5,447.7	49.2	5,568.6	58.1
	1000	70,716.5	4.5	7,568.0	17.4	5,487.0	33.7	5,481.8	50.4
Reverse	10	701,274.5	262,144.0	8,172.2	196.1	5,772.1	124.6	5,726.0	172.8
	50	716,795.6	262,144.0	7,860.6	130.1	5,596.6	124.4	5,698.2	172.8
	100	721,115.1	262,144.0	7,796.0	127.6	5,651.8	123.0	5,611.8	172.7
	500	827,554.1	262,144.0	7,850.2	55.7	5,658.4	72.4	5,746.7	135.5
	1000	731,670.6	262,144.0	7,439.9	17.2	5,660.8	39.5	5,656.1	105.6

TABLE 6 Average Time( $\mu$ s) and Cache Misses for 1M points collection (uniform distribution) over SSD, using LRU.

Table 7 shows, in a more condensed format, the behavior of the adjacency matrix and the  $k^2$ -tree with  $k = 4$  for all different sets of points (1, 10, and 100 million points) using a uniform distribution. Let us focus on the range queries. As we can see, for the adjacency matrix, the cache misses remain the same irrespective of the number of points in the matrix and the query times vary only slightly. Since the matrix occupies always the same space, and the algorithm must perform the same iterations, this stability in the cache misses and times was expected. The slight variation in the times is probably due to the time consumed by the algorithm to build the vector containing the result, which we expect to be larger when the density of the matrix increases.

However, the density of the matrix does affect the size of the  $k^2$ -trees, as we already discussed regarding the information shown in Table 5. Higher densities produce larger  $k^2$ -trees, so an increase in the cache misses and running times is logical. In any case, the conclusions drawn from Table 6 remain: the adjacency matrix performs better only for CheckLink, and the  $k^2$ -trees clearly outperform it for all the remaining queries.

Num. Points	Query	Adj. Mat.		$k^2$ -tree ( $k = 4$ )	
		Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss
1,000,000	CheckLink	8.24	1.00	32.00	2.26
	Range100	23,235.47	202.22	59.84	2.08
	Range1000	2,160,958.26	2,122.58	6,327.10	7.86
	Range10000	215,263,893.33	32,235.68	287,523.66	39.38
	Direct	70,630.23	4.50	5,393.48	75.48
	Reverse	721,115.08	262,144.00	5,651.82	122.96
10,000,000	CheckLink	5.26	1.00	23.98	3.16
	Range100	22,246.46	202.22	261.46	3.20
	Range1000	2,160,714.82	2,122.58	45,370.80	18.92
	Range10000	213,788,742.32	32,235.68	2,215,762.06	179.70
	Direct	70,247.14	4.51	17,072.90	229.26
	Reverse	750,843.66	262,144.00	17,055.30	363.42
100,000,000	CheckLink	11.58	1.00	25.20	3.38
	Range100	24,809.06	202.22	1,499.18	4.52
	Range1000	2,178,861.80	2,122.58	317,863.92	45.08
	Range10000	215,362,311.44	32,235.68	15,535,741.38	845.08
	Direct	70,287.40	4.50	50,384.38	522.44
	Reverse	761,419.40	262,144.00	52,339.56	974.48

TABLE 7 Performance of Adj. matrix and  $k^2$ -tree ( $k = 4$ ): uniform distribution, SSD, LRU (cache size = 100).

Table 8 shows the same information for the datasets that follow a gaussian distribution. Again focusing on the range queries, we can conclude that neither the density nor the data distribution affects the behavior of the adjacency matrix: the size of the data structure and the number of cache misses are the same, and the running are similar irrespective of the density and/or the data distribution. For the  $k^2$ -trees, however, both factors affect the size used by the structure, thus it will also affect its performance. It shows a trend similar to the one in Table 7 for the uniform distribution, but with lower times for CheckLink, direct, reverse, and small range queries (which was expected, as  $k^2$ -trees work better with clustered data).

Another aspect that might seem inconsistent is that cache misses are lower for the *Range100* operation than for CheckLink. That can be explained by the fact that the shown value in the table is the average number of cache misses. The range queries must explore more pages and will have more cache misses in the initial queries, but in the last ones many pages will already be in the cache, so the “amortized” cache misses is lower. However, the CheckLink operation may take points that are far from each other, so last queries do not benefit from the initial ones.

Finally, let us examine how the cache size affects the performance of the algorithms over the data structures. Figures 3-6 will show the cache misses when the queries are posed over datasets (1, 10 and 100 million points) using a uniform distribution. All the experiments used an LRU buffer replacement scheme and were run over an SSD device.

Figure 3 shows that, for the CheckLink query, the adjacency matrix has only 1 cache miss (it only has to read one page to test if the given point exists), so having a large cache would only waste memory, and a cacheless scheme would obtain the same results. The  $k^2$ -trees, however, benefit from a larger cache, but as the figure shows, around 50-100 pages would produce the best results for a trade-off between memory usage and performance.

For the direct neighbors query (Figure 4), the adjacency matrix has always a low cache miss (around 5), because the matrix is stored row by row, and it can be explored sequentially. Thus, again a cacheless scheme or a small cache of about 5 pages would suffice. Once more,  $k^2$ -trees would take advantage of a larger cache. Again, in this case, a cache size of 100 pages would produce good enough results. For the reverse neighbors query (Figure 5), we can see that the adjacency matrix has a very large number of cache misses (recall that this query obtains a “column” of the matrix, so in most cases one page read is needed to test each bit of the column). However, since only one bit is tested from each row, having a larger cache would not produce any benefits, as the figure shows. For the  $k^2$ -trees, again, it seems that for most cases the best cache size would be around 50-100 pages.

Num. Points	Query	Adj. Mat.		$k^2$ -tree ( $k = 4$ )	
		Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss
1,000,000	CheckLink	5.02	1.00	20.04	1.78
	Range100	23,549.36	202.22	30.46	0.38
	Range1000	2,208,441.56	2,122.58	9,560.38	4.14
	Range10000	218,626,474.72	32,235.68	362,106.48	31.88
	Direct	70,142.40	4.52	4,040.98	49.28
	Reverse	766,077.46	262,144.00	3,914.40	82.34
10,000,000	CheckLink	15.94	1.00	22.88	2.44
	Range100	24,857.48	202.22	145.60	1.10
	Range1000	2,179,865.06	2,122.58	66,859.62	12.78
	Range10000	216,723,924.76	32,235.68	2,567,005.84	154.34
	Direct	70,238.18	4.50	15,359.46	169.86
	Reverse	774,030.86	262,144.00	12,988.24	266.34
100,000,000	CheckLink	5.08	1.00	36.82	2.54
	Range100	24,849.96	202.22	929.54	1.94
	Range1000	2,183,819.56	2,122.58	412,030.54	38.64
	Range10000	215,491,201.46	32,235.68	15,561,157.66	774.30
	Direct	70,328.92	4.52	46,870.20	401.72
	Reverse	771,140.96	262,144.00	39,106.88	648.02

TABLE 8 Performance of Adj. matrix and  $k^2$ -tree ( $k = 4$ ); gaussian distribution, SSD, LRU (cache size = 100).

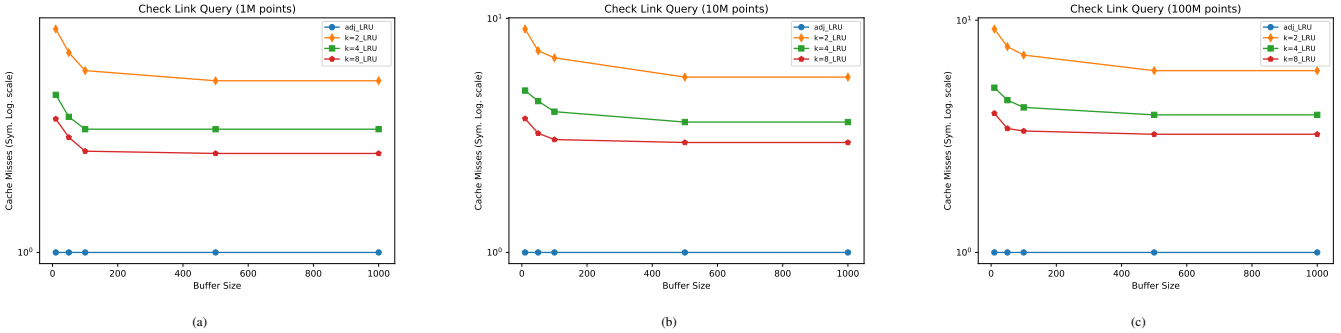


FIGURE 3 Performance of CheckLink queries (uniform distribution, LRU, SSD).

Finally, range queries (Figure 6) exhibit a similar behavior than reverse neighbor queries: it appears that increasing the cache size would have no beneficial effects for the adjacency matrix, and a cache of about 100 pages can offer a fairly good performance for  $k^2$ -trees, or possibly larger for the wider range queries.

### 5.3 | $k^2$ -tree using large real world datasets

We present in this section the experiments conducted to evaluate the behavior of the  $k^2$ -tree on external memory using large real world datasets. We will verify again the impact of parameters such as the type and size of the used cache, or the  $k$  parameter of the  $k^2$ -tree. In this case, we evaluate the behavior of  $k^2$ -trees alone, because the other alternatives were not able to reasonably handle such large datasets: the chosen implementation of LQT was not able to build the Quadtree, and the adjacency matrix would take up about 8 Petabytes of space.

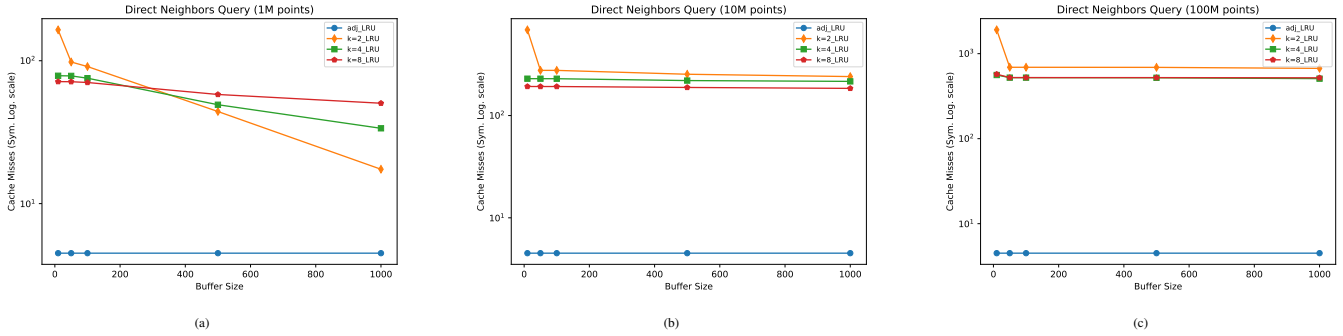


FIGURE 4 Performance of direct neighbors queries (uniform distribution, LRU, SSD).

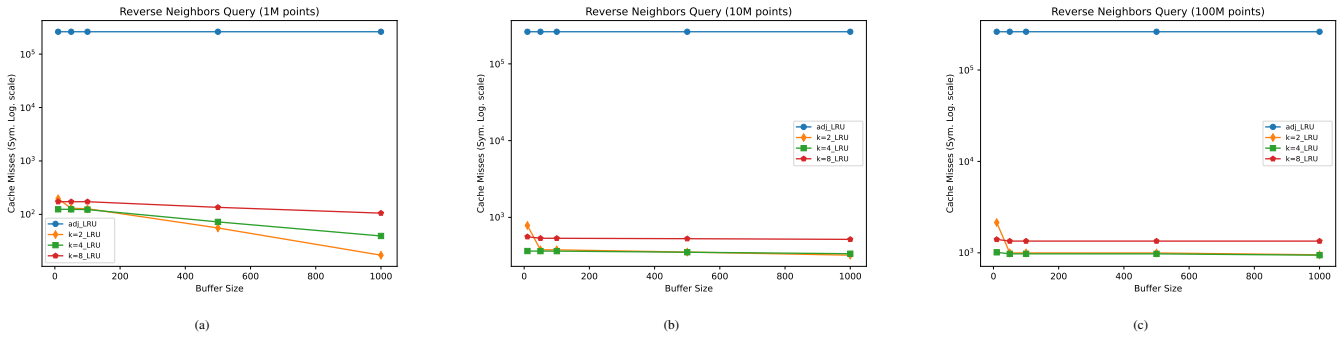


FIGURE 5 Performance of reverse neighbors queries (uniform distribution, LRU, SSD).

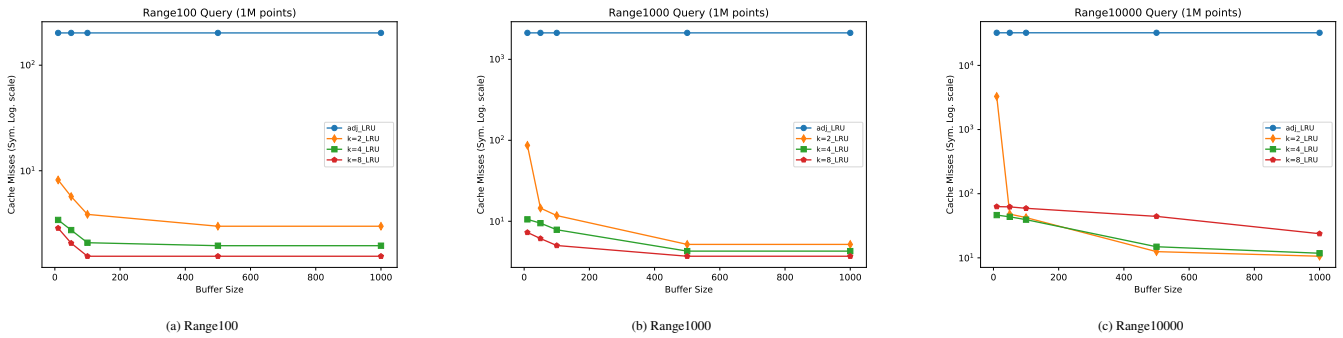


FIGURE 6 Performance of range queries on 1M dataset (uniform distribution, LRU, SSD).

The datasets contain spatial information coming originally from OpenStreet Map, and are offered by the SpatialHadoop group<sup>28</sup>. They correspond to different geographical elements on a planetary scale, namely parks, buildings, and road networks.

The original datasets were in CSV files, having the geometrical information stored as WKT (Well Known Text). We have extracted all the relevant points, with a resolution of  $1/10^6$  of a degree, and used them to build our  $k^2$ -trees. Figure 7 shows the shape of the datasets which, as they store terrestrial information, all resemble a world map. This figure shows the degree of clusterization of the information, which is very relevant for the degree of compression obtained by the  $k^2$ -trees.

The matrix size is the same for all datasets:  $360M \times 180M$  points (longitude ranges from  $-180$  to  $+180$ , latitude from  $-90$  to  $+90$ , and the precision was  $1/10^6$  of a degree). The number of points, as well as the file sizes and compression ratio, are shown in Table 9. Note that the *Orig. size* column represents, in MB, the theoretical size of a file that stores the points using a 32-bit integer for each coordinate. The CSV files we used to generate the  $k^2$ -trees were much larger.

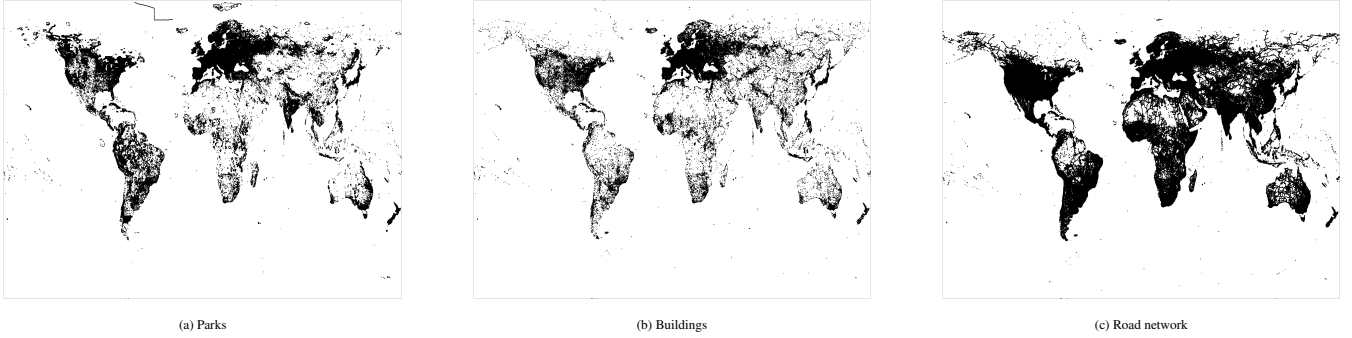


FIGURE 7 Shape of the real datasets coming from OpenStreetMap data.

Dataset	Num. Points	Orig. Size	$k^2$ -tree ( $k = 2$ )	$k^2$ -tree ( $k = 4$ )	$k^2$ -tree ( $k = 8$ )
Parks	305,326,851	2,329	1,296 (56%)	2,446 (105%)	6,135 (263%)
Buildings	613,819,101	4,683	1,914 (41%)	3,536 (76%)	8,701 (186%)
Road network	677,663,983	5,170	2,897 (56%)	5,470 (106%)	13,731 (266%)

TABLE 9 Sizes of the real datasets (in MB)

For the experiments, like in the previous cases, we launched batches of 50 queries (after another 50 warm-up queries). For CheckLink, half the query points were in the  $k^2$ -tree and half of them were not. These same query points were used to perform direct and reverse neighbor queries, and the range queries were also centered in them.

First, we examine the behavior of the  $k^2$ -tree to evaluate the queries using different page replacement schemes, cache sizes, and values of  $k$ .

We can highlight, from what Table 10 shows, the better performance, measured in terms of cache misses and execution time, of LRU versus LFU for most of the queries and all of the values for  $k$  and cache sizes. This better performance is more noticeable in direct and reverse neighbors queries, which are usually the most expensive in computational terms. For example, for direct neighbors, both cache misses and execution time for LRU are about one half of what LFU obtains, and for some cases of reverse neighbors LRU can achieve less than 6% of the LFU cache misses. This is consistent with what we found in the previous tests, with smaller, synthetic datasets. We do not show in this table the results for the cacheless scheme, because (again, consistent with previous experiments) it always has a worse performance.

Let us use a more graphical approach (Figures 8-10) to show the behavior of the operations depending on the cache size and the value of  $k$ , which will produce similar conclusions as the previous experiments. For almost all the situations, a bigger value of  $k$  obtains better performance, with less cache misses (at the expense of more space being used), and a cache size of around 50-100 pages provides a good performance.

Figure 8 shows the results for the CheckLink query. We can see that the pattern is similar for all datasets, and a cache of 100 pages would produce a good trade-off between memory usage and performance.

Figure 9 shows that, for *Range10000* queries, that the  $k^2$ -tree with  $k = 2$  has a bad behavior without cache, but with 50 or more pages it performs reasonably well. For higher values of  $k$ , the cache size seems more irrelevant, but a value of 50-100 pages would again produce a good performance. This figure also shows that values of  $k = 4$  or  $k = 8$  have virtually the same behavior.

Finally, for the direct neighbor queries (and that would also be the case for reverse neighbors) we find again that the  $k^2$ -tree obtains similar results for  $k = 4$  and  $k = 8$ , but slightly lower performance (more cache misses) with  $k = 2$ .

From the data in these figures, we can conclude that queries that are more selective (like CheckLink) are more affected by the arity of the  $k^2$ -tree (the value of  $k$ ), while for less selective ones it is not so important. In most cases, regardless of the  $k$  value and the type of query, a cache size of around 50-100 pages is a good trade-off between used memory and performance.

As a final summary, Table 11 shows the  $k^2$ -tree performance for the 3 datasets, considering a  $k^2$ -tree with  $k = 4$  and what we concluded is a good configuration: LRU buffer replacement scheme, and a cache size of 100 pages. We can see that the  $k^2$ -tree times do not always escalate proportionally to the dataset sizes or its number of points (see Table 9). This happens because

Query	Cache Size	$k^2$ -tree ( $k = 2$ )				$k^2$ -tree ( $k = 4$ )				$k^2$ -tree ( $k = 8$ )			
		LFU		LRU		LFU		LRU		LFU		LRU	
		Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss
CheckLink	10	63.82	9.32	57.48	10.54	93.38	4.76	30.68	5.54	58.54	3.22	20.10	3.46
	50	66.40	9.00	59.04	9.40	34.12	4.48	31.16	4.60	26.48	3.12	21.14	3.26
	100	64.62	8.70	59.32	8.94	34.92	4.46	37.66	4.54	29.80	3.10	34.24	3.10
	500	75.68	8.44	67.94	8.44	37.60	4.38	35.42	4.38	26.70	3.08	25.12	3.08
	1000	72.72	8.44	67.86	8.44	37.30	4.38	35.52	4.38	26.86	3.08	24.72	3.08
Range100	10	86.70	13.50	63.84	9.04	65.22	6.88	46.16	4.56	131.62	5.88	113.26	3.24
	50	80.94	11.42	63.44	7.72	55.18	4.58	45.72	3.80	121.04	3.32	90.54	3.00
	100	76.04	9.08	63.46	7.28	52.58	3.70	46.00	3.74	133.98	2.88	92.36	2.88
	500	75.34	6.86	69.88	6.86	55.38	3.60	48.48	3.60	122.68	2.86	94.28	2.86
	1000	74.50	6.86	70.12	6.86	55.12	3.60	48.62	3.60	122.54	2.86	95.56	2.86
Range1000	10	203.88	36.68	133.10	9.28	265.88	24.38	171.44	3.84	509.62	16.10	340.26	2.66
	50	187.72	27.80	118.06	5.60	236.82	9.28	168.78	3.16	521.14	3.06	366.48	2.40
	100	158.04	14.40	116.94	5.26	218.10	3.16	169.76	3.10	481.48	2.28	338.66	2.30
	500	137.86	4.88	118.62	4.88	220.72	2.98	170.08	2.98	479.84	2.28	343.20	2.28
	1000	138.46	4.88	118.26	4.88	223.58	2.98	170.44	2.98	473.70	2.28	339.56	2.28
Range10000	10	10,498.56	1,927.56	5,990.12	233.70	10,537.84	920.42	6,640.32	6.24	18,200.86	539.36	12,152.78	6.30
	50	10,140.88	1,741.72	5,289.36	8.60	10,236.44	796.54	6,291.94	5.38	17,891.38	453.88	11,958.52	5.82
	100	9,624.06	1,516.42	5,237.08	8.28	9,658.20	502.32	6,418.54	5.26	17,118.90	188.20	11,807.42	5.76
	500	6,639.74	7.80	5,205.24	7.80	8,717.90	5.10	6,312.74	5.10	17,198.86	5.60	11,866.02	5.60
	1000	6,566.24	7.80	5,218.06	7.80	8,616.52	5.10	6,318.46	5.10	16,865.60	5.60	11,723.90	5.60
Direct	10	54,271.60	11,969.92	29,794.04	1,700.26	39,346.26	6,964.96	18,832.38	412.30	36,410.90	4,984.40	19,087.24	371.70
	50	54,624.02	11,852.16	27,151.54	530.64	35,628.10	6,922.50	18,621.80	376.28	33,476.70	4,899.98	18,594.98	353.34
	100	53,380.32	11,785.08	26,632.14	527.04	35,942.04	6,781.18	18,146.58	374.80	33,240.88	4,862.10	18,938.42	352.14
	500	52,552.64	11,242.88	26,447.68	525.96	35,280.96	6,454.70	18,044.00	374.32	32,719.48	4,576.40	18,586.14	351.34
	1000	50,839.10	10,751.80	26,244.22	524.70	33,768.12	6,046.94	17,986.02	373.56	31,826.50	4,205.06	18,295.48	350.86
Reverse	10	171,304.34	38,319.96	93,872.54	5,310.30	124,148.24	22,320.94	60,930.44	1,924.44	120,151.58	15,841.34	61,852.88	2,071.36
	50	170,626.64	37,832.00	84,647.00	2,016.60	111,963.22	21,965.58	58,589.22	1,761.76	105,601.56	15,603.62	60,540.16	1,969.26
	100	168,006.02	37,533.16	84,042.68	2,000.14	111,298.96	21,830.96	58,206.38	1,753.40	103,874.62	15,355.02	60,173.60	1,963.10
	500	166,500.58	36,160.64	82,380.26	1,996.10	110,286.06	20,864.78	58,048.90	1,750.46	105,202.46	14,716.66	60,064.78	1,958.88
	1000	164,858.14	34,965.64	81,967.10	1,993.78	109,682.52	20,174.66	57,995.44	1,748.76	102,326.12	14,196.00	60,853.56	1,957.78

TABLE 10 Performance of  $k^2$ -trees for the dataset *road-network*.

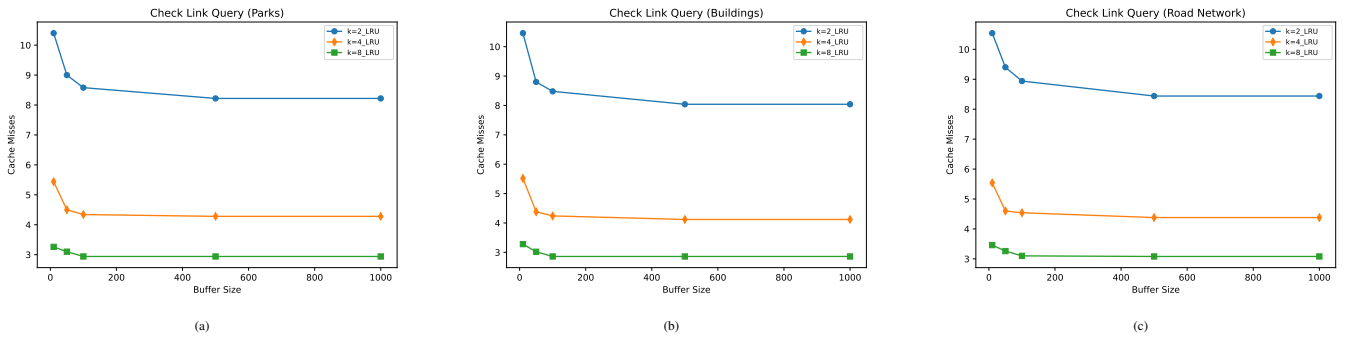


FIGURE 8 CheckLink queries on (a) *parks*, (b) *buildings* and (c) *road-network*, different cache sizes, SSD.

$k^2$ -tree are specially affected by the degree of clustering of the data and, as Figure 7 shows, the shapes of the dataset are different, having different clustering patterns. This is particularly noticeable for the Buildings dataset and the range operations.

## 6 | CONCLUSIONS

This work presents an implementation of the  $k^2$ -tree, a compact data structure that can be used to efficiently represent and query binary relationships or any other data represented as a binary matrix, that resides in external memory. This disk-based  $k^2$ -tree can be accessed without a cache, or with it, considering different sizes, and LRU and LFU as the replacement schemes. We implemented the algorithms to solve the most common queries over  $k^2$ -trees: CheckLink (or Access), direct and reverse neighbors, and range queries. We provide a theoretical analysis of the cost of these algorithms.

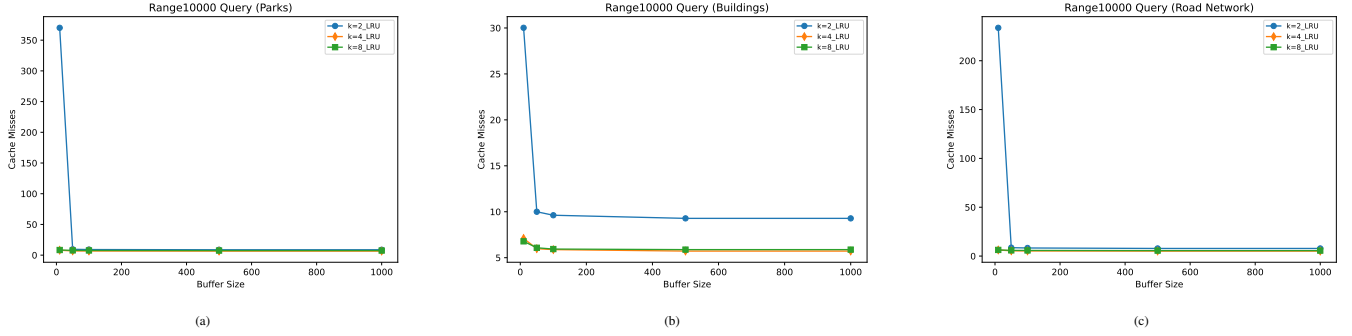


FIGURE 9 Range10000 queries on (a) parks, (b) buildings and (c) road-network, different cache sizes, SSD.

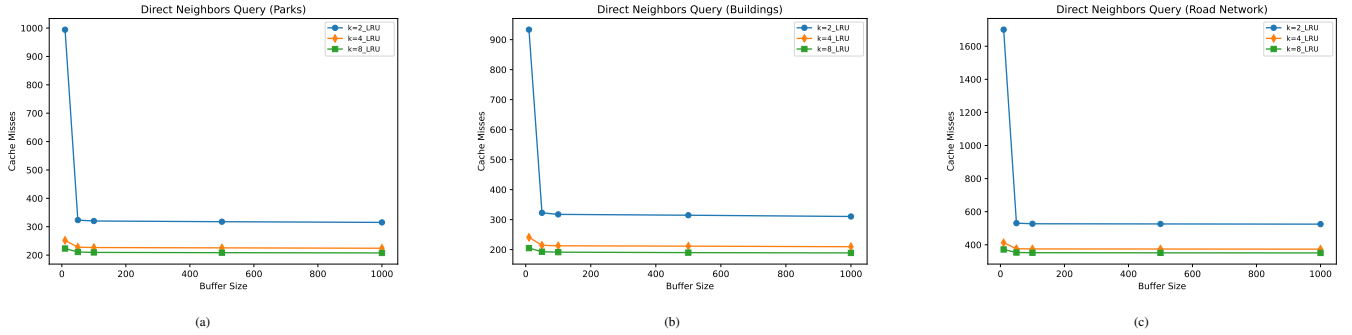


FIGURE 10 Direct neighbors queries on (a) parks, (b) buildings and (c) road-network, different cache sizes, SSD.

Query	Parks		Buildings		Road Network	
	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss	Time( $\mu$ s)	CMiss
CheckLink	31.36	4.34	32.36	4.24	37.66	4.54
Range100	52.30	3.46	106.60	3.58	46.00	3.74
Range1000	393.44	3.48	4,429.90	5.88	169.76	3.10
Range10000	16,319.72	6.78	185,072.34	18.26	6,418.54	5.26
Direct	11,339.86	226.58	9,474.92	212.74	18,146.58	374.80
Reverse	35,934.02	1,016.48	21,795.74	594.30	58,206.38	1,753.40

TABLE 11 Performance of  $k^2$ -trees ( $k=4$ ) for real datasets, with LRU (cache size=100).

We have also conducted a thorough empirical evaluation of the  $k^2$ -tree, considering different values for  $k$ , different cache sizes and replacement schemes. The  $k^2$ -tree performance, considering I/O operations (usually counting cache misses that would lead to I/O operations) and the running time to execute the queries, was compared with two well established data structures: a Linear Quadtree, and an Adjacency matrix.

From these experiments we can conclude that, in terms of storage space, the  $k^2$ -trees are clearly the best option, even considering values of  $k$  greater than 2. It is important to point out that, for large datasets, neither the LQT nor the adjacency matrix could be built. For the latter, it would require over 8 Petabytes to build our large, real world datasets.

In terms of performance,  $k^2$ -trees are also the best option in general: it is always competitive, and in most cases it outperforms its alternatives. The only exception is the CheckLink query, for which the adjacency matrix always solves with only one I/O operation. However, for more complex operations, like range queries, the  $k^2$ -tree can be up to hundreds of times faster.

As for the scalability, we showed that  $k^2$ -trees escalate well, and that the use of a cache clearly improves their performance, without needing extremely large cache sizes. As the experiments demonstrate, with about 50 to 100 pages,  $k^2$ -trees perform well

for large datasets. A  $k^2$ -tree with  $k = 4$ , a cache of 100 pages, and an LRU replacement scheme, showed to be the best alternative in our experiments.

Our proposal has two aims: become an alternative to represent compact data structures in external memory, and being a reference for future works on this same line.

As a future work, we plan to improve the implementation of the  $k^2$ -tree, taking into consideration not only the compression but also other fundamental properties such as the locality of reference. That would probably diminish the number of cache misses and thus improve the overall performance for the most frequent query types over the  $k^2$ -tree.

In this sense, we are considering an approach that follows the idea of B-heaps<sup>29</sup> where the  $k^2$ -tree bitmap would not represent the breadth-first traversal of the conceptual  $k^2$ -tree, but something closer to a depth-first traversal (this could directly benefit, for example, the CheckLink operation). To ensure a fairer comparison, several adaptations of the adjacency matrix (such as Morton codes or Hilbert curves) will also be taken into account. Other versions of  $k^2$ -trees that would be suitable to be implemented on disk include the hybrid  $k^2$ -tree (that has 2 values of  $k$ , usually a higher value for upper levels and a lower value for lower levels) or a version where the bitmap  $L$  is encoded using a matrix vocabulary. Additionally, more advanced statistical analyses (significance tests or confidence intervals for experimental results, among others) will be performed.

We are also considering the study, under the external memory model, of different compact data structures, such as the wavelet tree, DACs, compressed suffix arrays, or the FM-Index.

## ACKNOWLEDGEMENTS

All researchers from U. Bío-Bío were partially funded by ANID Grant 1230647, ALBA group code 2130591 GI/VC, Project INES I+D 22-14 and Projects 2130520 IF/R and 2230534 IF/R. Miguel R. Penabad work was partially funded by MCIN/AEI/10.13039/501100011033 and NextGenerationEU/PRTR projects TED2021-129245B-C21 (PLAGEMIS), PDC2021-121239-C31 (FLATCITY-POC) and PID2020-114635RB-I00 (EXTRACompact); by GAIN/Xunta de Galicia project GRC: ED431C 2021/53; CITIC is funded by the Xunta de Galicia through the collaboration agreement between the Department of Culture, Education, Vocational Training and Universities and the Galician universities for the reinforcement of the research centers of the Galician University System (CIGUS).

## References

1. Vitter JS. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.* 2001;33(2):209–271. doi: 10.1145/384192.384193
2. Brisaboa NR, Ladra S, Navarro G.  $k^2$ -Trees for Compact Web Graph Representation. In: Karlgren J, Tarhio J, Hyvärinen H., eds. *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariseikä, Finland, August 25-27, 2009, Proceedings*. 5721 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009:18–30
3. Navarro G. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 570 pages.
4. Raman R, Raman V, Rao SS. Succinct Dynamic Data Structures. In: . 2125 of *WADS '01*. Springer Berlin Heidelberg, Springer-Verlag 2001; Berlin, Heidelberg:426–437.
5. Brisaboa N, Ladra S, Navarro G. DACs: Bringing Direct Access to Variable-length Codes. *Information Processing and Management*. 2013;49(1):392–404.
6. Grossi R, Gupta A, Vitter JS. High-order Entropy-compressed Text Indexes. In: SODA '03. SIAM. Society for Industrial and Applied Mathematics 2003; Philadelphia, PA, USA:841–850.
7. Grossi R, Vitter JS. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* 2005;35(2):378–407. doi: 10.1137/S0097539702402354
8. Sadakane K. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. In: ISAAC '00. Springer. Springer-Verlag 2000; Berlin, Heidelberg:410–421.
9. Brisaboa N, Ladra S, Navarro G. Compact Representation of Web Graphs with Extended Functionality. *Information Systems*. 2014;39:152–174.
10. Brisaboa N, Luaces M, Navarro G, Seco D. Space-efficient Representations of Rectangle Datasets Supporting Orthogonal Range Querying. *Information Systems*. 2013;38(5):635–655.

11. Navarro G. Wavelet trees for all. *Journal Discrete Algorithms*. 2014;25:2–20.
12. Ferragina P, Manzini G. Opportunistic Data Structures with Applications. In: FOCS '00. IEEE Computer Society. 2000; Washington, DC, USA:390-398.
13. Brisaboa NR, Luaces MR, Navarro G, Seco D. A Fun Application of Compact Data Structures to Indexing Geographic Data. In: Boldi P, Gargano L., eds. *Fun with Algorithms* Springer Berlin Heidelberg. 2010; Berlin, Heidelberg:77–88.
14. de Bernardo G, Gagie T, Ladra S, Navarro G, Seco D. Faster compressed quadtrees. *Journal of Computer and System Sciences*. 2023;131:86-104. doi: <https://doi.org/10.1016/j.jcss.2022.09.001>
15. Gagie T, González-Nova J, Ladra S, Navarro G, Seco D. Faster compressed quadtrees. In: University of Arizona and IEEE Signal Processing Society. IEEE Computer Society Conference Publishing Services 2015:93–102.
16. Venkat P, Mount D. A succinct, dynamic data structure for proximity queries on point sets. In: CCCG community. 2014; Halifax, Nova Scotia:216–225.
17. Ishiyama K, Kobayashi K, Sadakane K. Succinct Quadtrees for Road Data. In: Beecks C, Borutta F, Kröger P, Seidl T., eds. *Similarity Search and Applications* Springer. Springer International Publishing 2017; Cham:262–272.
18. Castro JF, Romero M, Gutiérrez G, Caniupán M, Quijada-Fuentes C. Efficient computation of the convex hull on sets of points stored in a k2-tree compact data structure. *Knowl. Inf. Syst.*. 2020;62(10):4091–4111. doi: 10.1007/s10115-020-01486-9
19. Navarro G. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Computing Surveys*. 2021;54(2):article 29.
20. Samet H. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.*. 1984;16(2):187–260. doi: 10.1145/356924.356930
21. Corral A, Vassilakopoulos M, Manolopoulos Y. Algorithms for Joining R-Trees and Linear Region Quadtrees. In: Güting RH, Papadias D, Lochovsky F., eds. *Advances in Spatial Databases* Springer. Springer Berlin Heidelberg 1999; Berlin, Heidelberg:251–269.
22. Arge L, Vahrenhold J. I/O-efficient dynamic planar point location. *Computational Geometry*. 2004;29(2):147-162. doi: <https://doi.org/10.1016/j.comgeo.2003.04.001>
23. Carniel AC, Roumelis G, Ciferri RR, Vassilakopoulos M, Corral A, Aguiar Ciferri dCD. Porting disk-based spatial index structures to flash-based solid state drives. *GeoInformatica*. 2022;26(1):253–298. doi: 10.1007/s10707-021-00455-w
24. Carniel AC, Ciferri RR, Aguiar Ciferri dCD. A Generic and Efficient Framework for Spatial Indexing on Flash-Based Solid State Drives. In: Kirikova M, Nørnvåg K, Papadopoulos GA., eds. *Advances in Databases and Information Systems - 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings*. 10509 of *Lecture Notes in Computer Science*. Springer. "Springer International Publishing" 2017:229–243
25. Sarwat M, Mokbel MF, Zhou X, Nath S. Generic and efficient framework for search trees on flash memory storage systems. *GeoInformatica*. 2013;17(3):417–448. doi: 10.1007/s10707-012-0164-9
26. Quijada Fuentes C, Penabad MR, Ladra S, Gutiérrez Retamal G. Compressed Data Structures for Binary Relations in Practice. *IEEE Access*. 2020;8:25949-25963. doi: 10.1109/ACCESS.2020.2970983
27. Arge L, Thorup M. RAM-Efficient External Memory Sorting. In: Cai L, Cheng SW, Lam TW., eds. *Algorithms and Computation* Springer Berlin Heidelberg 2013; Berlin, Heidelberg:491–501.
28. Eldawy A, Mokbel MF. SpatialHadoop: A MapReduce Framework for Spatial Data. In: IEEE. 2015:1352–1363
29. Kamp PH. You're doing it wrong. *Commun. ACM*. 2010;53(7):55–59. doi: 10.1145/1785414.1785434