A Appendix: Supplementary Material

A.1 Indexing with Space-filling Curves

Space-filling curves are maps from the unit interval to a hypercube of any dimensionality [54]. Most space-filling curves are constructed by fractal, self-similar, recursive procedures. Although these curves have many fascinating properties, here we are interested in their ability to induce a “vicinity-sensitive” total order in a dataset. They preserve neighborhood relations with good probability: if point A is closer to point B than it is to point C in a space, that relationship tends to remain in the curve (Figure 16).

Space-filling curves have been implicitly used to perform similarity searches in multidimensional spaces for a very long time. Indeed, one of the first multi-dimensional indexes ever proposed [45] employed space-filling curves hidden in the idea of “bit shuffling”, “bit interleaving” or “bit interleaving”, which consisted of interleaving the bits of individual space coordinates to generate a search key. Interleaving bits induces a type of space-filling curve called a Z-order curve, which explains why the method works well. However, Faloutsos [24] was the first to explicitly refer to the concept of curves, and Faloutsos and Roseman [26] were the first to suggest the use of curves other than Z-order curves (Gray-code and Hilbert curves).

These pioneering methods worked simply: they use a curve to map a multi-dimensional vector onto a one-dimensional key representing a position in the curve (which we call here an extended-key). The position is then used to perform a search by similarity. For example, when performing a kNN search, a good heuristic is to use the nearest elements in a curve as the nearest elements in a space because of the “vicinity-sensitiveness” explained above. The biggest problem from using curves stems from the existence of boundary regions in which neighborhood-relation preserving properties are violated and points closer in space are placed further apart in the curve (Figure 16). This issue substantially worsens as dimensionality increases [36, 58].

To overcome boundary effects, Megiddo and Shaft [42] have proposed the use of several curves simultaneously. As is done with the multiple straight lines of MEDRANK [22] or the multiple hash-tables of LSH [32] (see Section 2.1), Megiddo and Shaft build an independent subindex for each curve. The query is then sought on all subindexes, in the hope that in at least one of them it will not fall close to a boundary region. Megiddo and Shaft have presented the idea in very general terms, without describing which types of space-filling curves should be used and what operations must be performed to differentiate them. Shepherd et al. [58] have developed the idea, specifically recommending the use of several identical Hilbert curves, where different copies of vectors are transformed by random rotations and translations. Whether or not those transformations could be optimized was left unanswered. Finally, Liao et al. [36] has solved the problem of choosing the transformations by devising the necessary number of curves and an optimal set of translations to obtain bounds on the approximation error for a kNN search.

A distinct class of methods was proposed by Mainar-Ruiz and P´erez-Cort´es [41]. Rather than using multiple curves, Mainar-Ruiz and P´erez-Cort´es have proposed using multiple instances of the same element in only one curve. Before inserting the element’s instances in the curve, their algorithm randomly disturbs the position of the instances, giving those instances the opportunity to fall into different regions of

Fig. 16 Space-filling curves provide a “vicinity-sensitive” map: relative closeness in the space tends to be preserved in the curve (points A, B and C). However, in some boundary regions, those properties are violated (points A, B and D).
Space-filling curves are usually obtained by a recursive refinement procedure, resulting in a fractal, self-similar curve. Each refinement step is called an order of the curve. The curve fills the Continuum at the infinite limit of the procedure, but for digital data it is not necessary to reach that limit. The figure shows three successive steps of the Hilbert curve (left) and the Z-order curve (right).

**A.2 Technical Details on Space-filling Curves**

Space-filling curves are fractal curves introduced by G. Peano and D. Hilbert [54], which provide a continuous surjective map $C : [0, 1] \rightarrow [0, 1]^d$ from the unit interval to a hypercube of any dimensionality. Space-filling curves are usually constructed by recursive procedures and, in the limit, fill the entire space (Figure 17).

It is known (due to a result of E. Netto) that such a $C$ mapping cannot be at once bijective and continuous. Dropping the condition of injectivity, Peano built the first known continuous surjective map from a line to a space. Interestingly, in the recursive procedure used to build the curve, all finite steps are bijective, but the limiting (and thus, effectively space-filling) curve becomes self-intersecting. For applications using digital data, we can always consider $C$ to be bijective because the limit needed to address the true Continuum is never reached.

When using a space-filling curve map for indexing, we are interested in the pre-images of the query and the data points. Using the same notation as before, for $b_i \in B$ and $q \in Obj$, we are interested in those $C^{-1}(F(b_i))$ that are close to $C^{-1}(F(q))$ (remember that $F$ is the function that maps multimedia objects into feature vectors). We call the pre-image $C^{-1}(x)$ the extended-key of feature vector $x$.

There is a direct relationship between the number of refinements we must use in the recursive curve (called the order of the curve) and the precision of the data we wish to index. If dyadic curves are employed (like the Z-order, Gray-order or Hilbert curves), we must use $m^{th}$ order curves to index coordinates of $m$ bits. Note that the bijective map $C^{-1}$ preserves the number of bits: from $d$ coordinates of $m$ bits each to a single extended-key with $d \times m$ bits.

It is important to emphasize that a curve does not have a concrete representation in the indexes. This fact is a common source of confusion for those who learn for the first time about indexing based on space-filling curves. The curve is a useful abstraction, employed to create the map $C^{-1}$, which generates “neighborhood-sensitive” extended-keys. Then, the extended-keys are used in conventional, one-dimensional, indexing structures (a hash-table, a B-tree, etc.).
The actual computation of $C^{-1}$ depends on the type of space-filling curve being used. For the Hilbert curve, several recursive algorithms have been proposed, but the most efficient scheme is an iterative one [15]. For the Z-order curve, the computation is extremely simple: it suffices to intercalate the bits of the coordinates.

It is interesting to analyze which types of data can be indexed by space-filling curves. Space-filling curves are able to organize vectors of any fixed-length ordinal data, provided that the order is the “natural” one: the order of the data is the same order as that of the numbers (binary codes) in which they are encoded. Otherwise, a transformation must be used to translate the vector of data into a vector of orders.

For multimedia descriptors, we are primarily interested in vectors of numeric data. When the coordinates are integers, it is easy to see that the scheme works, although the programmer must be sure to treat negative numbers in $C^{-1}$ correctly. Although less obvious to see, the scheme works with almost no modifications for (IEEE 754) floating-point numbers. In that encoding, the bits of the exponent are in more significant positions than the bits of the mantissa, and the order of the encoded numbers is “natural”. Again, the only caveat is to ensure that one is correctly handling the most significant sign bit, which is used for negative numbers.

A.3 The Choice of Multicurves Index

This section provides a comparison of our space-filling similarity search methods (Multicurves and Hypercurves) with FLANN [46], which is one of the most popular and efficient packages for approximate search in high-dimensional spaces. The evaluation considers the performance (execution times) under baseline single node scalability conditions. We anticipate, however, that an extensive comparison is beyond the scope of this paper, and this section intends to demonstrate that the performance of our space-filling method is comparable to state-of-the-art algorithms in our use cases. Additional comparisons of our space-filling based method are provided into previous publications, where our approach has been shown to outperform some of the most efficient methods available in the literature [72,73].

Among the various available methods in the literature, FLANN [46] has been selected for this comparison because it combines several state-of-the-art algorithms and data structures used in approximate search in high-dimensional spaces, such as: KD-Trees, Hierarchical K-means Trees, LSH and brute force linear search, and provides an “auto-tune” mode that chooses and parameterizes the best algorithm for a given dataset. Therefore, the comparison to FLANN allows us to directly compare to the best algorithm automatically selected among that set of methods it implements.

The methods were executed using a machine equipped with an Intel Xeon 2.27 GHz processor, a NVIDIA Tesla M2090 GPU, and 12GB of main memory. We employed the same dataset used by the original FLANN evaluation [46], which consists of two sets of 100K image patches from Winder/Brown [76] for the reference and the query datasets. Reproducing the protocol proposed in the original FLANN evaluation, we varied the dimensionality of the feature vectors (descriptors) by subsampling dimensions from the image patches. In this experiment, the methods were required to find the 20 nearest neighbors of each query descriptor.

FLANN was parameterized with its autotune mechanism, which was configured to use 1% of the dataset for training purposes. We then recorded FLANN’s precision (45%, 50%, 54%, and 58%, respectively, for 16, 64, 256, and 1024 dimensions) and hand-tuned the sequential algorithm Multicurves aiming, on each case, at similar compromises of execution time and precision.

The performance results are presented in Figure 18. As shown, the execution times achieved by Multicurves (using a single CPU
core) are comparable to those attained by FLANN in all configurations, but in the lowest dimensionality (16 dimensions) case. In that case, the KD-Tree method selected by FLANN is expected to perform very well, since the effects of the curse of dimensionality are lesser. Multicurves, however, presents better scalability in terms of dimensionality, and in the 1024 dimensions case it is already slightly ($1.08 \times$) faster than FLANN.

In addition to the two sequential methods, Figure 18 also presents, for reference, the performance of the parallel method Hypercurves (using a single GPU and a single node). As shown, this baseline version of Hypercurves greatly outperforms Multicurves and FLANN for all configurations. In addition, the Hypercurves GPU-based execution achieves higher speedups in comparison to the other sequential methods as the data dimensionality increases. Further, for the dataset with 1024 dimension, Hypercurves attains a speedup of about $18\times$ on top of FLANN. It is important to keep in mind that this is a baseline comparison, and that Hypercurves scalability improves superlinearly as we add computing nodes, as shown in our main results sections (Section 7).

As presented in this section and in previous works [72,73], our space-filling based methods are comparable or outperform some of the most efficient methods available in the literature. In addition, it is important to bear in mind that the sorted-list approach used by the space-filling curves is essential for the approximate-equality proof (Section 4.2) that allows partitioning the search using smaller probe-depths, and this, in turn, is a key aspect that enables Hypercurves to achieve super-linear scale-ups, demonstrated experimentally in Section 7.4. This would not be possible with other methods, as the basic premises of our equivalence proof would then be violated.