BitPart: Exact Metric Search in High(er) Dimensions

Alan Dearle*, Richard Connor*

School of Computer Science, University of St Andrews,
St Andrews, KY16 9SX, Scotland, UK

Abstract

We define BitPart (*Bit*wise representations of binary *Part*itions), a novel exact search mechanism intended for use in high-dimensional spaces. In outline, a fixed set of reference objects is used to define a large set of regions within the original space, and each data item is characterised according to its containment within these regions. In contrast with other mechanisms only a subset of this information is selected, according to the query, before a search within the re-cast space is performed. Partial data representations are accessed only if they are known to be potentially useful towards the calculation of the exact query solution.

Our mechanism requires $\Omega(N \log N)$ space to evaluate a query, where N is the cardinality of the data, and therefore does not scale as well as previously defined mechanisms with low-dimensional data. However it has recently been shown that, for a nearest neighbour search in high dimensions, a sequential scan of the data is essentially unavoidable. This result has been suspected for a long time, and has been referred to as the *curse of dimensionality* in this context.

In the light of this result, the compromise achieved by this work is to make the best possible use of the available fast memory, and to offer great potential for parallel query evaluation. To our knowledge, it gives the best compromise currently known for performing exact search over data whose dimensionality is too high to allow the useful application of metric indexing, yet is still sufficiently low to give at least some traction from the metric and supermetric properties.

Keywords: Similarity search, metric space, metric indexing, metric search, four-point property

1. Context

To set a formal context, we are interested in searching a (large) finite set of objects S which is a subset of an infinite set U, where (U, d) is a metric space: that is, an ordered pair (U, d), where U is a domain of objects and d is a

*Corresponding author
Email address: `alan.dearle@st-andrews.ac.uk` (Alan Dearle)

total distance function $d : U \times U \to \mathbb{R}$, satisfying postulates of non-negativity, identity, symmetry, and triangle inequality [**?**]. The general requirement is to efficiently find members of S which are similar to an arbitrary member of U given as a query, where the distance function d gives the only way by which any two objects may be compared. There are many important practical examples captured by this mathematical framework, see for example [**? ?**]. The simplest type of similarity query is the *range search* query: for some threshold t, based on a query $q \in U$, the solution set is $R = \{s \in S | \, d(q, s) \leq t\}$.

The essence of metric search is to spend time pre-processing the finite set S so that solutions to queries can be efficiently calculated. In all cases distances between members of S and selected reference or "pivot" objects are calculated during pre-processing. At query time the relative distances between the query and the same pivot objects can be used to make deductions about which data values may, or may not, be candidate solutions to the query.

Many pre-processing mechanisms attempt to organise the data set in terms of locality. To achieve this, they construct a data structure (typically a recursively defined tree of some kind) where objects that are close to each other within the metric space are similar in terms of navigational paths within the data structure. In the best possible case, this allows data within a small distance of a given query to be almost deterministically identified by navigation of the structure at query time. In turn this gives a query cost which is logarithmic with respect to the size of the data.

As the dimensionality of the data increases, however, the property of locality is increasingly lost: from the perspective of an individual data item, sampled distances become increasingly similar and the concept of data clustering is entirely lost. This is just one effect of the attribute commonly referred to as the *curse of dimensionality*.

Navigational solutions become ineffective in these conditions. Most research aimed at searching high-dimensional spaces is dedicated to approximate search, where a selection of pre-determined reference objects is used, and distances to all these are calculated for every element during construction. These distances are used to re-cast the original space into some other space where indexing properties are better, distance calculations are cheaper, or both. Typically the main tradeoff is that extra query efficiency is achieved in return for a loss of semantic query effectiveness. Such mechanisms are either approximate, or produce candidate sets of results from within which the true results must be determined by re-accessing the original data.

In our work, we also pursue the strategy of performing all of the reference point distances at the start of a query, rather than incrementally during a navigational calculation. A fixed set of reference points is chosen, and used to partition the infinite space into a large set of discrete subspaces. During pre-processing, every element of the finite space is characterised according to which of these subspaces it is in. The novel aspect of our approach is that, at query time, a subset of these subspaces is identified, with respect to the query, according to those which must, or alternatively cannot, contain solutions to that query. For only these subspaces, the containment information is used to calculate an accurate set of potential query solutions. In this way, the mechanism maximises the use of main memory for a search calculation.

This paper is an extended version of one originally published in the 11th International Conference on Similarity Search and Applications (SISAP18) [?]. That paper introduced the core mechanism, of which we now give a more detailed description, and demonstrated its credibility via promising experimental results over the SISAP benchmark data sets. In this paper we add a full analysis of the mechanism applied to higher-dimensionality spaces and experimental analysis of the parallelisation potential of the mechanism. All of the code used to perform the many experiments is available online[1].

The novelty of the work lies in the combination of using a fixed set of reference objects to characterise the data, followed by their use in an exact metric search algorithm which maximises the efficacy of memory use. There are many mechanisms which use similar fixed sets of reference objects (see for example Section 2); our mechanism is particularly well-suited to exact search as the dimensionality of the underlying data increases, by minimising the memory footprint required for the search algorithm. Furthermore our algorithm is inherently decomposable and parallelisable, and is well-suited to implementation on modern processors.

2. Related Work

There are a large number of metric search algorithms described in the literature. In Section 2.1 we outline some others which, like ours, use a fixed number of reference objects in a "one-time" manner. Section 2.2 gives a short outline of a conditional hardness result which contextualises the approach followed in this paper. Our approach can also benefit greatly from recent work which establishes stronger mathematical properties for an important subset of metric spaces, as outlined in Section 2.3. Finally, Section 2.4 describes some relational database technology from which this work can potentially benefit.

2.1. Search using Fixed Reference Objects

There are a number of well-known mechanisms in which the distances between a query and a fixed set of reference objects are used to guide a first phase of search. All such mechanisms may either be regarded as approximate, or subsequently checked against the original data set for accuracy.

LAESA [?] has typically been used for metric filtering, rather than approximate search. For each element of the data, distances to a fixed set of reference points are recorded in a table. At query time, the distances between the query and each reference point are calculated; the table can then be scanned row at a time, and each distance compared; if, for any reference object p_i and data object s_j the absolute difference $|d(q, p_i) - d(s_j, p_i)| > t$, then it is impossible

[1] `http://github.com/aldearle/BitPart`

for s_j to be within distance t of the query, and the distance calculation can be avoided. LAESA can be used as an efficient pre-filter for exact search when memory size is limited. The same data can be re-cast into a metric space using the Chebyshev metric but this does not typically result in any significant performance increase.

The best known mechanisms which use a fixed partition of the original data for approximate search are based on *permutation orderings* [**?** **?** **?**]. There is a significant variety of techniques, but the essence of the approach is to characterise each element of the data in terms of its distances to a set of pre-selected reference objects, and then to compare elements using various aspects of similarity in this ordering. The approach is similar in that this effectively creates a large number of regions within the universal space, one for each ordering. However in all cases that we know of, an approximate search mechanism is created based on some cost function over the resulting orderings; there does not seem to be any clear mechanism for using the regions thus defined as an exact search mechanism. Especially in higher dimensional spaces, two very similar objects in the original space may lie on opposing sides of a number of the important boundaries, which then appear as distant objects and fail to appear in the search results.

Sketches [**?**] uses a fixed set of reference objects. A large number of enclosed regions is established based on these objects, in which each element of the data can be judged to be contained or otherwise. An ordering is imposed on the regions, allowing each element of the data to be characterised by a bitstring, where each bit position represents membership of the corresponding region. This bitstring is referred to as the *Sketch* of the object.

The idea is that, if two objects are close to each other within the data space, then they will be in many of the same regions, and thus have many of the same bits set. Therefore, the Hamming Distance over the Sketches should give a good proxy to distance in the original space. The purpose of the technique is that the proxy objects should be much smaller, and the Hamming metric much cheaper, than those of the original metric space.

The technique suffers from the same probabilistic issues as permutation orderings, in that for any two very similar objects there is a finite probability that they will appear very different in the proxy space.

Other probabilistic mechanisms have also been proposed, for example [**?** **?** **?**]. These are all quite similar in outline but with different ways of defining regional boundaries and treating the proxy space. Our work differs significantly in that we describe a mechanism which is guaranteed to give all correct results from the original space, and in the way that the increased cost of increasing dimensionality or query threshold can be controlled.

2.2. Hardness of High-Dimensional Querying

A recent conditional hardness result given in [**?**] shows that a sequential scan is virtually unavoidable for nearest neighbour search in high dimensional spaces. This has of course been suspected for a long time by the indexing community, and the observed phenomenon has been named the *curse of dimensionality* in this context.

Our mechanism does comprise a sequential scan, but this scan is highly optimised. For each element of the data, an n-bit representation is required in main memory, where n requires to be greater than $\log_2 N$. The result of the computation over this data however is a single bitmap of size N, whose positive bits correspond to data which may be a valid solution to a query. The necessary sequential scan then occurs over this bitmap, considering only the bits that are set. If these comprise a small percentage of the total, then this scan can be achieved very efficiently in engineering terms: that is, while the computation is conceptually $\Omega(N)$, this is associated with a very small constant factor.

2.3. Metrics and Supermetrics

Much work on finite isometric embeddings was conducted in the 20th century, by e.g. Blumenthal [?], Wilson [?] and Menger [?]. Blumenthal uses the phrase *four-point property* to mean a space that is 4-embeddable in 3-dimensional Euclidean space.

This property means that, for any selection of four objects from the original space, it is possible to construct a tetrahedron where the vertices correspond to the objects, and the length of each side is equal to the corresponding inter-object distance. Any metric space is 3-embeddable in 2-dimensional Euclidean space; this property is equivalent to triangle inequality [?]. Spaces with the four-point property are thus also metric spaces but also have some stronger properties, which can have profound consequences in metric search. The four-point property exists in many commonly-used metric spaces, including Euclidean, Cosine[2], Jensen-Shannon, Triangular and Quadratic Form distances.

Connor and Vadicamo have applied these results in theoretical mathematics to this more practical domain [? ?] and the term *supermetric* is now used to refer to metrics with the property.

Use of the supermetric property gives better exclusion conditions, as detailed in Section 4.2. The concepts given in this paper are equally applicable to both metric and supermetric spaces; as would be expected from a space with tighter geometric properties, however the results we show using the four-point property are significantly better than results relying only on the three-point property.

2.4. Column Databases

Column databases differ from Relational and NoSQL databases in that data is stored in columns rather than as records (rows) or in an object, semi-structured or unstructured manner. Column databases were introduced to the academic community in 2005 with the presentation of C Store [?] which was later developed into the commercial product Vertica [?]. Column databases are optimised for data warehousing applications in which read operations dominate; they permit the database to read only those columns required for specific

[2]for the correct formulation, see [?].

queries, for example finding the addresses of all customers, and avoiding fetching attributes that are not required for that particular query. Even with complex indexing mechanisms this is hard to achieve using relational technology. The metric search application described in this paper is a perfect fit with column databases since there are a low number of writes (often 1), few updates (if any) and the requirement to be able to select columns of bit values attributes.

A column database would therefore provide a perfect implementation agent for a major part of our mechanism, referred to as Phase 2 in Section 4. Commercial implementations are available, although at this time we are yet to perform any experimental analysis of their use.

3. Conceptual Overview of BitPart

The core of the technique is to define a large set of binary partitions over the original space; the data is stored as a set of bitmaps according to containment with these regions. At query time, the query is assessed against this containment information, but without reference to the original data representation.

For each region, one of three possible conditions may be determined: (a) the solution set to the query must be fully contained in the region, or (b) there is no intersection between the region and the solution set, or (c) neither of these is the case. In either case (a) or (b), the containment information stored may be useful with respect to solving the query, and is used as part of the query computation. In case (c) however the containment information is of no value, and is not accessed as a part of the computation. This approach maximises the effectiveness of memory used for calculation against the original data representation: the amount of memory required depends on the cardinality of the original data, but not on the size of its individual objects.

3.1. Illustrated Example

Figure 1 shows a simple example within the 2D plane, comprising four reference objects p_1 to p_4 and a set of six regions defined by them. The regions are respectively: A, the area to the left of the line between p_1 and p_2; B, the area above the line between p_1 and p_3; and the areas within the variously sized circles drawn around each p_1 to p_4, labelled C, D, E and F respectively. Note that each regional boundary defines a binary partition of the total space, such that each element of the space is either in, or out, of the region, and that this membership is defined only in terms of distances from defined reference objects. Thus in Figure 1, $A = \{u \in U | d(u, p_1) \leq d(u, p_2)\}$, $C = \{u \in U | d(u, p_1) \leq \mu\}$ for some value of μ, etc.

Figure 1 also shows a range query q drawn with a threshold t. It can be seen that all solutions to this query must lie within the area highlighted on the right hand side of the figure. The circle around the query intersects with two regional boundaries (A and E), and so no information is available with respect to these; however it is completely contained within two of the defined regions (B and D), and fails to intersect with the final two (C and F). Such containment

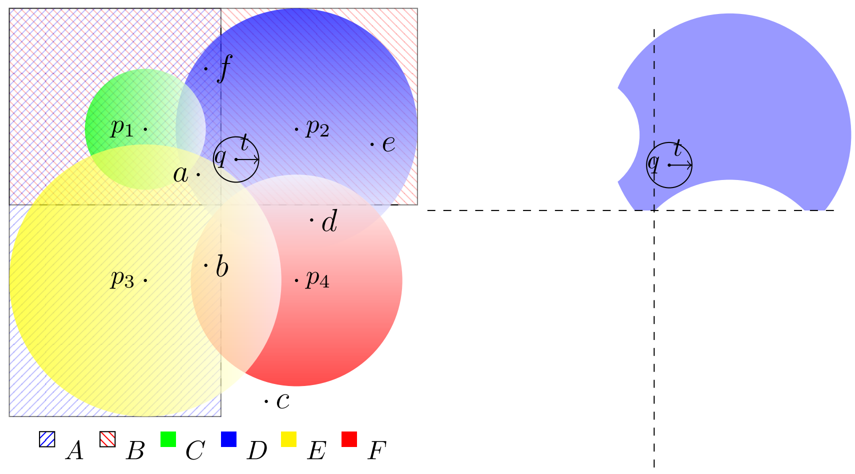

Figure 1: Any solution to the query q with threshold t must be in the circle centred around p_2, and above the dashed horizontal line; it cannot be in the circles centred around p_1 or p_4. No information is available wrt the circle around p_3 nor the vertical line; these partitions take no part in the exclusion calculation as the region boundaries intersect the query solution boundary. The shaded area shown on the right shows the possible loci of query solutions with respect to these regions.

and intersection is derivable only from the measurement of distances between the query and the reference objects, the definition of the regions, and the search radius. Here for example the possible solution area shown is determined using only the four distance calculations $d(q, p_1)..\, d(q, p_4)$.

Table 1 shows how the example data objects a to f are stored in terms of their regional containment. The row labelled $\mathcal{Q}$ shows the containment relation between the region containing all solutions to the query, and each pre-defined region A to F. Where the boundaries intersect, a $\cap$ is shown; where they do not, a Boolean value shows whether the query ball is contained or otherwise. Therefore, the only possible solutions to the query are those which match on all non-intersecting fields; in this case, the objects a, e and f. Note that this is equivalent to the Conjunctive Normal Form (CNF) expression $B \wedge \neg C \wedge D \wedge \neg F$. This expression therefore covers the set of all possible solutions to the query.

3.2. Overview of the Query Algorithm

The concepts shown in the example may be implemented as follows:

Phase 1 The query is checked against the regional definitions, and the two types of regions with non-intersecting boundaries are identified.

Phase 2 The regions thus identified are used to conduct a series of logical operations, thus identifying a set of candidate solutions for the query.

Table 1: The data representation according to containment of regions (see Figure 1). The representation of the query $\mathcal{Q}$ is equivalent to the CNF expression $B \wedge \neg C \wedge D \wedge \neg F$

Point	Regions					
	A	B	C	D	E	F
a	true	true	false	true	true	false
b	true	false	false	false	true	true
c	false	false	false	false	false	false
d	false	false	false	true	false	true
e	false	true	false	true	false	false
f	true	true	false	true	false	false
$\mathcal{Q}$	$\cap$	true	false	true	$\cap$	false

Phase 3 These candidate solutions are checked against the original data set to identify the exact solution to the query.

This gives interesting performance tradeoffs. Phase 1 is likely to be dominated by the cost of the distance calculations; as will be seen, the computation of non-intersecting regional boundaries requires only simple arithmetic calculations based on these. Phase 2 cost is dominated by the number of selected regions, and efficiency with which the logical operations can be performed; this is highly efficient on modern hardware, and also highly optimisable. The cost of Phase 3 directly depends on how much exclusion can be achieved, and will always be better with a larger number of regions, which will in turn require an increase in the cost of Phase 1.

However, a larger than necessary number of regions will result in redundancy during Phase 2 and therefore may increase the cost of this phase of the calculation more than a larger residue would do so. Finding exactly the best parameters for these phases depends on the size and dimensionality of the data as well as the cost of the distance metric, and is further discussed in Section 8.

4. Detailed Mechanism Description

4.1. Data structures

Before describing the algorithm in more detail, we describe the data structures used in the algorithm and their initialisation. We refer to a finite space (S, d) which is a subset of an infinite metric space (U, d).

A set P of enumerated reference objects p_0 to p_m is first selected from the finite metric space S. Based on this, we define a set of surfaces within U, defined according to the distance function d, each of which divide U into two parts. Surfaces within U are either *balls*, for example $\{u \in U \mid d(u, p_i) \leq \mu\}$ for some values i, μ, or *sheets*, for example $\{u \in U \mid d(u, p_i)) \leq d(u, p_j)\}$ for some values i, j. For each such surface, it is easy to categorise any element of u_i of U as being inside or outside an associated region, according to whether it is on the same side of the surface as p_i or otherwise. That is, for a ball region "in" means within the ball, and for a sheet region "in" means on the same side of the sheet as the first reference point used to define it. Note that there can be many more regions than reference objects; for example a set of m reference objects defines at least $\binom{m}{2}$ sheet regions, plus m ball regions, and can be used to define many more than this.

We now define the notion of an *exclusion zone* (EZ) as a containment map of S based on a given region; this is the information we will use at query time to perform exclusions and derive a candidate set of solutions. We impose an ordering on S, then for each s_i map whether it is a member of the region or otherwise. This logical containment information is best stored in a bitmap of n bits, where $n = |S|$. One such exclusion zone is generated per region and stored as the primary representation of the data set.

It is worth noting that an essential difference between our mechanism and others that use the same characterisation of the data (see Section 2) is that each of our bitmaps represents the containment of the whole data set within an individual region, rather than those regions which contain an individual object. The same information is thus divided with the opposite orientation; with reference to Table 1, we store the columns rather than the rows.

4.2. Query

The query process comprises three distinct phases as mentioned above.

Phase 1 Initially, the distance from the query q to each reference object p_i is measured. For each region, it can be established if the boundary of the solution ball intersects with the boundary of the region. If there is such an intersection, then the region is of no value to the ongoing computation. If there is no intersection, any solution to the query must lie on one or other side of the region's boundary, and that region is carried forwards to the computation in Phase 2.

For a ball region defined by reference object p_i and a radius μ, then the condition for intersection is:

$$|d(p_i, q) - \mu| \leq t$$

For a sheet region, the condition depends on whether the metric d has the supermetric property (see Section 2.3) or otherwise: if it does, the intersection condition is:

$$\frac{|d(p_i, q)^2 - d(p_j, q)^2|}{2d(p_i, p_j)} \leq t$$

otherwise the weaker[3] condition is:

$$\frac{|d(p_i, q) - d(p_j, q)|}{2} \leq t$$

[3]As this condition is weaker, the probability of an exclusion is lower. A full explanation is given in [**?**].

If the intersection condition holds, then the exclusion zone related to the region is not considered further; if it does not, then the exclusion zone is brought into the query calculation in one of two sets, depending on whether the query solutions are fully contained within, or without, the region in question. We will name these two sets of bitmaps B_{in} and B_{out}.

Phase 2 The second phase comprises the manipulation of the bitmaps deriving from the first phase to identify a set of candidate solutions. This may be efficiently achieved by a series of bitwise operations over these bitmaps. The solution to the query is guaranteed to lie within the intersection of the inclusion sets (derived by bitwise and operations) and not in the union of the exclusion set (derived by bitwise or). Thus any solution is guaranteed to be identified by the bitmap deriving from the following logical expression:

$$\left(\bigwedge_{b \in B_{in}} b\right) \wedge \left(\neg\left(\bigvee_{b \in B_{out}} b\right)\right)$$

Phase 3 The last phase consists of performing a sequential scan of the single bitmap resulting from Phase 2 and checking the data corresponding to any remaining set bits using the original space and distance metric in order to produce an exact solution to the query.

4.3. Balancing

In our initial experiments, we have tried both balanced and unbalanced bitsets. Balancing can be achieved by selecting a set of *witness* objects from the finite space S and finding a median distance or offset for these, so that the regional boundary divides the finite set into two equal parts. A large enough set of witness objects will give a good statistical approximation to the distribution of S.

For ball partitions, the median distance to the centre is used. For sheet partitions, an offset can be selected to balance the partition as explained in Section 7.

For unbalanced experiments we used a fixed radius partition for balls initially set to a median sampled distance from the space. For sheet partitions we used an offset of zero.

5. Experiments with SISAP data sets

The efficacy of the algorithm was first tested by running queries against the SISAP *colors* and *nasa* benchmark data sets [**?**]. The “colors” set contains 112,682 feature vectors of dimension 112, representing colour histograms from an image database. The “nasa” data set contains a set of 40,700 20-dimensional feature vectors, generated from images downloaded from NASA. In each case ten percent of the data is used as queries over remaining 90 percent of the set, at threshold values which return 0.01%, 0.1% and 1% of the data sets respectively.

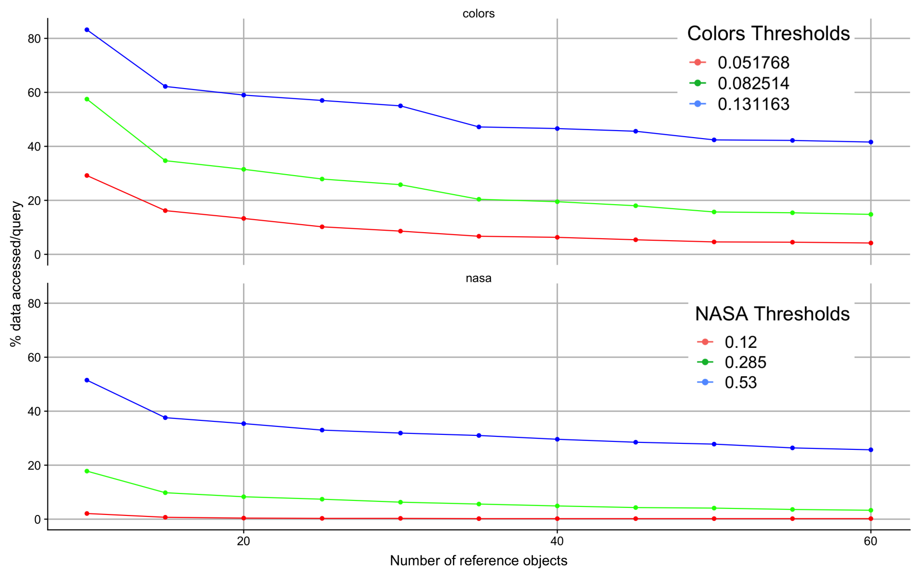

Figure 2: Percentage of data accessed for different numbers of reference points. Three different thresholds are used for the SISAP *colors* and *nasa* datasets. Although this space has the supermetric property it is not exploited in this experiment.

The benchmark thresholds for this purpose are $t_0 = 0.052, t_1 = 0.083, t_2 = 0.13$ for *colors* and $t_0 = 0.12, t_1 = 0.285$, and $t_2 = 0.53$ for *nasa*.

The metric used for both datasets is Euclidean distance. Reference objects are selected randomly from the dataset. The number of reference objects is varied from 10 to 60 in steps of 5. In this set of experiments the number of ball exclusion zones per reference point is set to 5, with the radii being set to a mean radius of 1.81[4] and mean $\pm$ 0.3 and mean $\pm$ 0.6. A single partition is used for the sheet portion exclusion zones. We report only residual distance calculations, which are the number of calculations made in phase 3 of the algorithm, excluding the (fixed number of) reference object distance calculations. The results of this experiment are shown in Figures 2 - 4, with the figures for 60 reference objects (the right hand side of the graph) given in Table 2. To put these figures into the metric indexing context, the top two rows of the table give the number of distance calculations per query reported in [**?**] for the Distal SAT operating with both normal metric and supermetric exclusion mechanisms.

The three sets of graphs shown in Figures 2 - 4 show the percentage of data points accessed by the query algorithm against the number of reference points for the *colors* and *nasa* datasets. These illustrate a number of interesting facets

[4]For balanced versions (see Section 7) the central radius is reset, but the same increments are applied

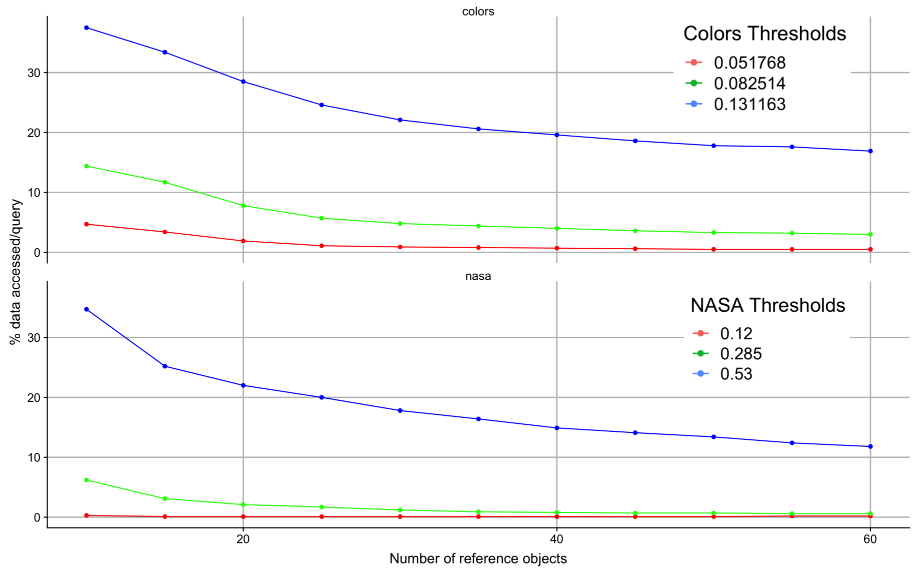

Figure 3: Percentage of data accessed for different numbers of reference points. Three different thresholds are used for the SISAP *colors* and *nasa* datasets exploiting the supermetric property.

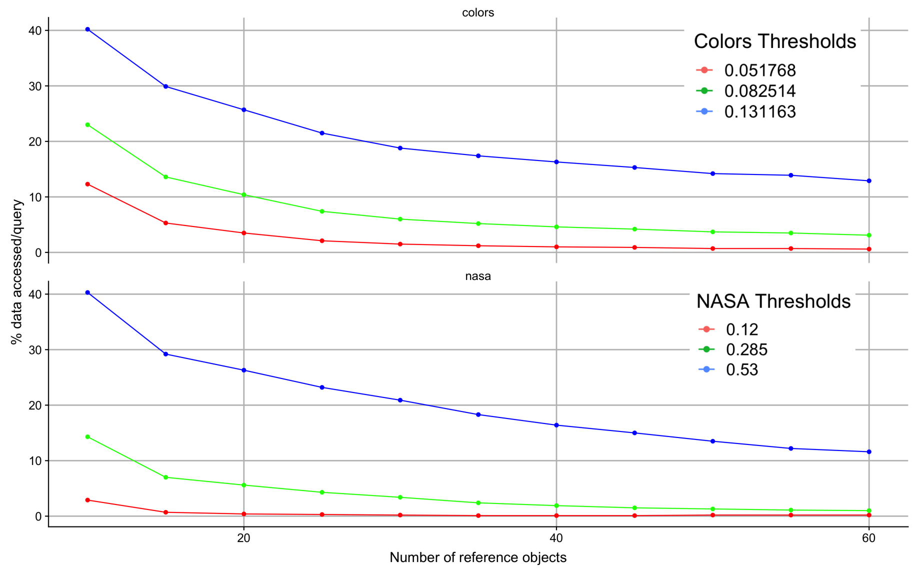

Figure 4: Percentage of data accessed for different numbers of reference points. Three different thresholds are used for the SISAP *colors* and *nasa* datasets exploiting the supermetric property and balancing.

Table 2: Residual distance calculations required when 60 reference objects are used (the numbers reported do not include the 60 distance calculations required for these.) The top two rows give comparable figures for the state-of-the-art Distal SAT. The data size is 90% of the original data after queries have been removed, i.e. 101,414 for *colors* and 36,630 for *nasa*.

	colors			nasa		
	t_0	t_1	t_2	t_0	t_1	t_2
DiSAT: metric	4049	9112	19745	554	2176	6448
DiSAT: supermetric	2015	5737	16199	320	1300	5444
metric, unbalanced	4207	14930	42139	14	1120	9202
metric, balanced	2246	9610	30250	11	822	7671
supermetric, unbalanced	544	3114	**13045**	3	296	**4122**
supermetric, balanced	**434**	**2961**	17080	**1**	**141**	4182

of the algorithm. As would be expected, increasing the number of reference objects has a significant effect of the number of distance calculations performed. However the number of reference objects used in these experiments is really quite small, much smaller than has been reported for other regional approaches e.g. [**?** **?**] given the accuracy shown by the relatively small number of residual distance calculations required. In particular, as shown in Table 2 we have the quite stunning result that, using 60 reference objects, the supermetric property, and balancing the data structures, we have an almost perfect characterisation of the *nasa* data set, with less that one false positive result per query.

Finally, we note that using balanced versions of the EZs helps, sometimes quite substantially, in all cases other than the larger search radii in the *colors* space. In cases where balancing makes the performance worse, this is because balancing the data set can reduce the probability of an individual EZ being selected during Phase 1, and this issue can be overcome by increasing the number of reference points. We return to this topic in Section 7.

6. Increasing Dimensionality

It is now understood that, as dimensionality increases, any search mechanism will degenerate to a linear scan of the data [**?**]. In this section we demonstrate that BitPart performs differently from conventional indexing methods in this respect. For low dimensional search, indexing mechanisms can achieve logarithmic performance ($\mathcal{O}(\log n)$ where n is the size of the data.) This approach will therefore have better scalability for low dimensions than BitPart, which always requires a sequential scan of order $\mathcal{O}(n)$.

However, at some point, increasing dimensionality will cause the spread of sampled distances to be smaller than the smallest distances within the set. When this happens, it is always impossible to achieve any traction from metric or supermetric properties.

Table 3: Values used in experiments for uniformly generated Euclidean spaces between 1 and 20 dimensions. Figures shown are: the mean distance sampled from within the space; the Intrinsic Dimensionality, and the search radius used, in each case representing the radius of a hypersphere whose volume is one-millionth of the unit hypercube volume.

dim	1	2	3	4	5	6	7	8	9	10
mean	0.333	0.521	0.662	0.778	0.878	0.969	1.052	1.128	1.2	1.268
IDIM	1.0	2.2	3.5	4.9	6.3	7.7	9.1	10.5	11.9	13.4
radius	$5.0e^{-7}$	$5.6e^{-4}$	$6.20e^{-3}$	0.021	0.045	0.076	0.111	0.149	0.189	0.229
dim	11	12	13	14	15	16	17	18	19	20
mean	1.332	1.393	1.452	1.508	1.562	1.615	1.665	1.715	1.763	1.81
IDIM	14.8	16.2	17.6	19.1	20.5	21.9	23.3	24.8	26.2	27.6
radius	0.269	0.309	0.348	0.387	0.425	0.462	0.498	0.533	0.568	0.602

Our intention with this mechanism is to explore the use of the $\Omega(n \log n)$ space required by our mechanism as the best possible use of a finite sized memory in which exclusion calculations can be performed. In the best case, we require only $\log_2 n$ bits per datum. In this context we are pushing to find the limits of exact search, rather than resorting to approximate mechanisms. As we show in Section 6.4, our mechanism continues to work well at dimensionalities that are sufficiently high to stop metric indexing mechanisms from functioning usefully.

6.1. Experimental Framework

In order to investigate increasing dimensionality, we use an experimental framework where we control dimensionality by using generated Euclidean spaces of increasingly higher dimensions, up to 20. For each of these spaces, data are randomly generated using a uniform distribution in each dimension within the unit hypercube. Range searches are conducted over queries generated within the same space, using a query radius which corresponds to one-millionth of the unit hypercube volume. Table 3 gives the figures used in experiments for each of these spaces.

It is worth noting that while the Euclidean dimensions vary from 2 to 20, the Intrinsic Dimensionality (IDIM, [**?**]) ranges from 2 to 28. While IDIM is not a perfect measure of tractability, it is usually a reasonable estimate. Among our results we present both elapsed clock times, and the percentage of the data accessed, to perform queries.

The percentage data access figures should be representative of other metric spaces with similar IDIM values. Such spaces may include physically much larger data objects with more expensive metrics; for example the IDIM of the Profiset[5] data set, which comprises 4,096-dimensional vectors taken from the

[5]`http://disa.fi.muni.cz/profiset/`

post-Relu *fc7* layer of the AlexNet convolutional neural network, is 25.3. This means not only that the distance metric cost is much greater, it also implies increased costs due to the necessity of moving large volumes of data within the memory hierarchy and caching effects. Therefore a mechanism which shows a clock time similar to exhaustive search in our framework may perform orders of magnitude better if the data size and metric cost are much greater in a different space with similar indexing characteristics.

For every generated space, one thousand queries are executed against one million data and the values reported are the mean values over these experiments, repeated as necessary to reduce the error of the mean to an acceptable level. All experiments described in this section were written in Java and conducted on a MacBook Pro with a quadcore 3.1 GHz Intel Core i7 and 16 GB of 2133 MHz LPDDR3 memory; no other processes were running and network was disabled. All data fits in main memory and, as is normal with metric space experiments, only relative values should be regarded as being significant.

6.2. Phase Costs

As previously noted, the cost of each of these queries is divided into three phases: (1) the cost of measuring distances between query and reference points, and assessing which of the exclusion zones can be used in the calculation; (2) the cost of performing the bitwise operations over the exclusion zones, and (3) the cost of performing distance calculations over those elements of the data which have not been excluded.

All experiments described in this section are conducted using the ℓ_2 distance which is relatively cheap for a memory-resident implementation; for a more expensive metric, the value of the BitPart mechanism is increased as shown in Section 8.1.

In all experiments, the Phase 1 cost is negligible. The distance calculation cost ranges between around 10^{-8} and 10^{-7} seconds between 2 and 20 dimensions which does not feature as our results are presented in terms of milliseconds per query; the maximum contribution for 60 reference points being less than 0.1ms. The cost of testing whether an individual EZ may be excluded or not is always a cheap arithmetic calculation based only on these calculated distances, which we have measured at around 10^{-8}s per EZ. This gives a cost of up to 0.2ms for our largest experiments, so again does not contribute significantly to the results presented. This is not to say that Phase 1 costs can always be ignored, as our figures do show potential for increasing the number of reference points, and therefore hugely increasing the number of zones, to address a further rise in dimensionality.

In the experiments we describe, at low dimensions almost all of the defined EZs are selected for use in Phase 2. This therefore gives the maximum possible cost for Phase 2, which depends linearly on the number of selected EZs. Again at low dimensions, this yields an almost perfect exclusion of data, therefore minimising the Phase 3 costs. As the dimensionality increases however, ever fewer EZs are selected by Phase 1, therefore the cost of Phase 2 decreases. As a direct consequence of this, the cost of Phase 3 increases, as more objects which

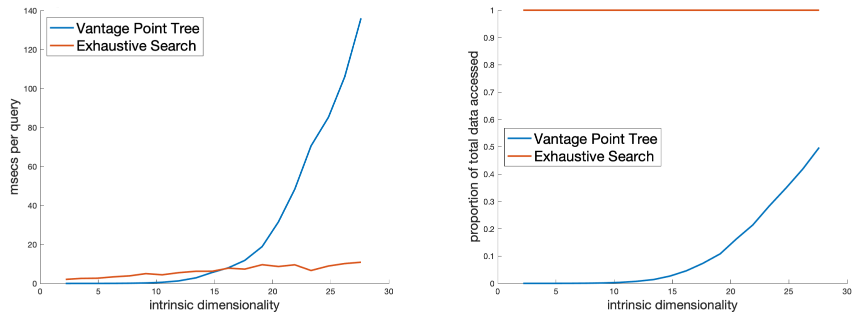

Figure 5: Elapsed time (left) and distance calculations (right) for the Vantage Point Tree as dimensionality increases, compared with exhaustive search. Distance calculations are shown as a proportion of the number required for exhaustive search.

are not solutions to the query fail to be excluded. The cost balances between these two phases depend on the characteristics of the space, particularly the dimensionality and the cost of distance evaluation. These issues are highlighted for our experimental framework in Section 6.4.

6.3. Comparison with Indexing Mechanisms

To demonstrate the effect of dimensionality increase over standard scalable mechanisms, the left hand side of Figure 5 shows the performance of these searches using a balanced Vantage Point Tree (VPT) index structure compared with an exhaustive search. It can be seen that, as dimensionality increases, the value of using the data structure becomes less, until the cost actually exceeds the cost of a simple exhaustive search of the data. This is a well-known phenomenon, caused by the joint effects of (a) ever fewer distance calculations being avoided by the exclusion mechanism, and (b) the cost overhead of the indexing mechanism itself, which increases as locality is lost due to increases in dimensionality. Where the cutoff occurs depends on both the intrinsic dimensionality and the absolute cost of the distance function; here the distance function is simple Euclidean (ℓ_2) distance which is relatively cheap. It can be seen here that the crossover point occurs at an IDIM of around 15 (which occurs at 11 dimensions for our generated data.) The figure on the right shows the proportion of the data set being accessed per query by the VPT. At the crossover point this is only around 2.7% of the data being searched. However the overhead of the search mechanism even for this simple index is such that its use is no longer cost-effective.

Of course our mechanism will also suffer from the effects of increasing dimensionality, however the cost parameters are quite different from those of a standard indexing approach, and here we examine the effect of these different parameters as dimensionality increases.

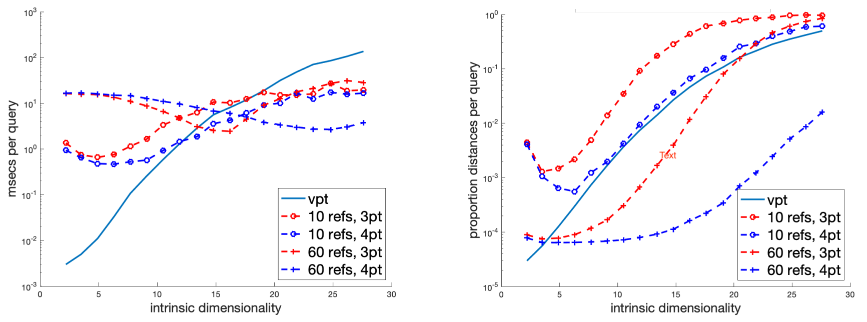

Figure 6: Elapsed Time (left) and Distances (right) per query for generated 20-dimensional Euclidean space. Distances are given as the proportion of the data accessed per query. Figures are shown for simple metric and supermetric variants of the BitPart mechanism with 10 and 60 reference points, compared with equivalent figures for a VPT query index. The distance used (ℓ_2) is relatively cheap so these figures show mostly the mechanism cost; true costs with a more expensive metric can be estimated from the combination of these values. Note the log scales on both Y-axes.

6.4. BitPart Cost Profile

Next, we show the essentially different properties of the BitPart mechanism. Figure 6 shows graphs of elapsed time and distance calculations performed for four variants of the BitPart, with the previous VPT figures also shown as a guide.

For these experiments we demonstrate the mechanism with 10 and 60 reference points respectively, and used the supermetric (four-point) and non-supermetric versions of the exclusion mechanism. In all instances, for n reference points n ball exclusion zones and $\binom{n}{2}$ sheet exclusion zones were created, and for each query as many of these as possible were brought into Phase 2 of the calculation. These are not optimal parameters for the mechanism but give the best insight into the relative costs for different numbers of dimensions and reference points.

There are several points of interest to note here. First, the lowest dimensional figures for elapsed time effectively gives the maximum overhead cost of the mechanism itself - around 1ms for 10 reference points and 15ms for 60. This is because in both cases almost all exclusion zones are being selected for all queries (55 for 10 reference points, 1,830 for 60) and consequently almost all of the data is excluded from the Phase 3 calculation. In the lower dimensions, for all configurations of BitPart shown in Figure 6, this means that many redundant EZs are being included in the Phase 2 calculation. As can also be seen, VPT will always be best at these low dimensions as the indexing structure effectively gives a deterministic route to the correct solutions as almost all incorrect branches are excluded.

Also for all configurations it can be seen that the elapsed time cost reduces, at first, as dimensions are increased; this is due to less EZs being brought into the second phase calculation, until the cost starts to rise again as this smaller number of EZs become less effective in excluding Phase 3 calculations.

To the left of this inflection point, redundant information is being used during both Phase 1 and Phase 2 computations. For example, using 60 reference points and the 3-point property, the inflection point occurs at a dimensionality of around 16. This implies that 60 references is more than optimal for this space, and fewer should be used. In this sense, the smallest elapsed time values for each of the four mechanisms represent a maximum cost for any space up to that dimensionality if the mechanism is correctly tuned.

Considering the 60 reference point versions, this means that both 3-point and 4-point mechanisms start to become more effective than the VPT as dimensionality exceeds around 15. One underlying reason for this is clear from the right-hand graph, where it can be seen that these mechanisms exclude all but 0.1% and 0.01% of the original distance calculations respectively, whereas the indexing structure at 10 dimensions accesses 1.7%. However the main extra cost is the manner in which the memory is accessed, as discussed in Section 9.

At the right-hand side of the elapsed time graph, it can be seen that the cost for all mechanisms other than the four-point at 60 reference points has essentially become the cost of an exhaustive search plus the mechanism cost: less than 50% of the data is being excluded by any mechanism, and at this stage the cost of an exhaustive search is not significantly more that the cost of random access plus distances to the non-excluded data.

6.5. Balls vs Sheets

We have already noted the value of using sheet partitions, in that the number of partitions and therefore EZs which can be generated from a fixed number of reference points is $\binom{n}{2}$, therefore allowing very few Phase 1 distance calculations to generate a huge number of geometrically distinct zones. However this may be of moderate value if the zones themselves are not as useful as those provided by balls, as in many cases a much larger number of calculations performed during Phase 1 would not add significantly to the overall mechanism cost.

To test the relative value of the different zone types, an experiment separating ball-based and sheet-based EZs was performed. Using 60 reference points, $\binom{60}{2}$ (i.e. 1,770) sheets were generated with both three-point and four-point exclusion criteria, and independently 1,770 reference points were used to generate the same number of ball partitions. Across the dimensions, the effectiveness of each type of zone was measured by the proportion of distance calculations, and also by the proportion of zones selected per query. Results are shown in Figure 7.

These results are quite surprising. We had expected the ball EZs to perform better than either sheet EZ type, however the four-point zones clearly perform best except for in very low dimensions. The right-hand chart suggests that the reason for this is that more zones are selected per query. At present, we cannot give a clear reason for this.

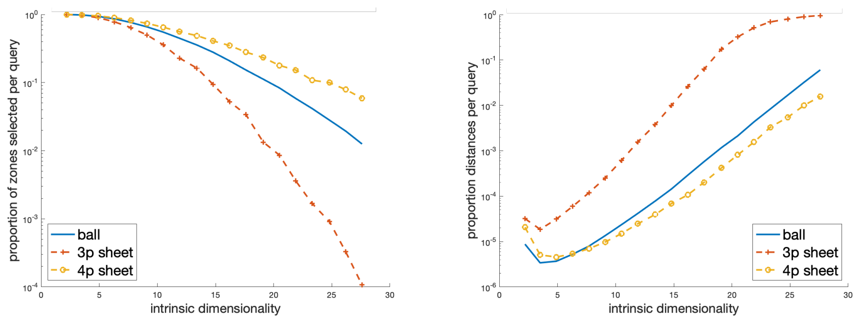

Figure 7: Performance of different zone types for Euclidean spaces of different dimensionalities. 1,770 zones are tested in each case with either ball, simple metric sheet, or supermetric sheet types of EZ. The left-hand sub-figure shows the number of zones selected per query as dimensionality increases, and the right-hand sub-figure shows the number of distance calculation required in Phase 3 as a proportion of the data size.

7. Balancing of Exclusion Zones

For our mechanism to work perfectly, we require two properties from the set of EZs selected for Phase 2 of the calculation:

balance: For each individual EZ, every element of the data will be set to 1 or 0 depending on whether it is an element of the zone or not. An EZ is perfectly balanced when exactly half the set is deemed to lie within the zone, and correspondingly exactly half the set lies outside the zone.

orthogonality: For each pair of EZs, there should be no correlation between the bits generated for individual data items. That is, for every element of the data, the probability of it being represented as a 1 in the first EZ should be independent of its representation within the second EZ.

If it was possible to achieve both of these properties, then only a small number of EZs would be required for Phase 2 in order to give very good exclusion. For an individual datum which is not within the threshold t of the query, then the probability of its representation failing to be excluded from a set of n EZs is $\frac{1}{2^n}$, and therefore from a set of N data, a set of approximately $\log_2 N$ EZs is most likely to exclude all but one datum from the search. As the experiments described here have a data size of 10^6, then if the set of Phase 2 EZs is both balanced and orthogonal, only a few more than 20 EZs should be optimal.

As described below, balanced EZs can be achieved. Finding orthogonal regions in a metric space is well known to be a difficult problem which we do not address further. However, we have noted in our experiments over 10^6 data, around 50-100 EZs in Phase 2 seems to suffice to exclude over 99% of the data in the highest dimensional spaces we have tested.

For a ball partition balancing can be achieved by selecting an appropriate radius, as the median distance between the reference point and the rest of the data. For a particular reference point p_i, a constant α is selected so that the membership condition for a data element s to be within the region becomes

$$d(p_i, s) \leq \alpha$$

and the conditions under which the exclusion zone is selected as a part of the calculation for a given query over q with threshold t are either

$$d(p_i, q) + t \leq \alpha$$

in which case the EZ is added to the B_{in} set, or

$$d(p_i, q) - t > \alpha$$

in which case the EZ is added to the B_{out} set.

Calculating the correct value for α for every EZ would be expensive; in the experiments a witness set of 5,000 values is used to calculate α with no significant loss of accuracy.

For the sheet partitions, an offset can be used to achieve the same effect. For non-supermetric spaces, the normal inclusion condition for a data point s with respect to the pair of reference points p_i, p_j is

$$d(p_i, s) - d(p_j, s) \leq 0$$

However, it is possible to replace this zero with a constant value, i.e. the condition becomes

$$d(p_i, s) - d(p_j, s) \leq \alpha$$

where again α is calculated as the median value of $d(p_i, s_i) - d(p_j, s_i)$ where s_i ranges over a large enough set of witness objects. The condition for inclusion in B_{in} becomes

$$\frac{d(p_i, q) - d(p_j, q)}{2} + \alpha \leq t$$

and inclusion in B_{out} is

$$\frac{d(p_i, q) - d(p_j, q)}{2} - \alpha > t$$

The use of a modifier value for hyperplane exclusion seems not to be well-known in general, but was first justified in [**?**]. The four-point membership and exclusion conditions are also modified equivalently to achieve balanced exclusion zones, this use has also been previously explained in [**?**].

8. Parameter Selection

Unlike most indexing mechanisms, BitPart requires to be parameterised to perform at its peak for a given space by selection of the best number of reference objects. The implications for performance over the three Phases of the computation as the number of reference points increases are as follows:

Phase 1 A larger number of distance calculations are required; typically however this number will be very small compared to those required in Phase 3. Apart from the distances, only simple arithmetic operations are required to identify the non-intersection zones, making the whole cost of Phase 1 insignificant.

Phase 2 More EZs will be selected for this phase, increasing its cost. The number selected is proportional to the number available, but the proportion varies with the dimensionality of the space.

Phase 3 As more EZs are selected for Phase 2, less residual distances are required, thus reducing the cost of Phase 3.

Therefore, the overall effect is that as the number of reference objects increases, Phase 2 cost increases and Phase 3 cost decreases.

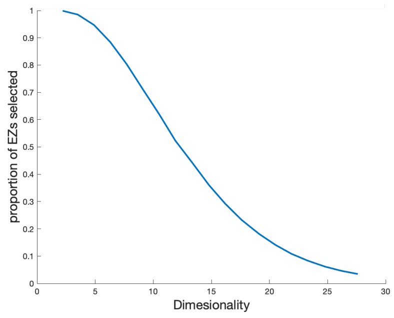

Figure 8: Mean proportion of EZs selected per query as dimensionality increases

Figure 8 shows the rapid decline in the proportion of the available EZs selected as the dimensionality of the space increases.

Figure 9 shows how this effect can be countered without a large increase in the number of distance calculations required. As the number of reference points is increased, the increase in available zones increases very rapidly, as there are $\binom{n}{2} + n$ zones for n references. The left-hand figure shows this increase, while the right-hand figure shows the average number selected per query in a 20-dimensional Euclidean space.

Figure 10 shows the effect of increasing the number of reference points. The left-hand sub-figure shows the residual (Phase 3) distance calculations required when using the same reference objects as in Figure 9. The right-hand sub-figure shows the elapsed time of execution.

In this case it can be seen that a “sweet spot” exists for the number of reference points, in this case at around 70. This occurs because, as explained earlier, as the number of EZs increases, the Phase 2 cost increases, but the Phase 3 cost decreases. The left-hand side of Figure 10 effectively shows Phase 3 costs as these are directly proportional to the number of distance calculations performed. In the right-hand sub-figure it can be seen that at around 70 reference

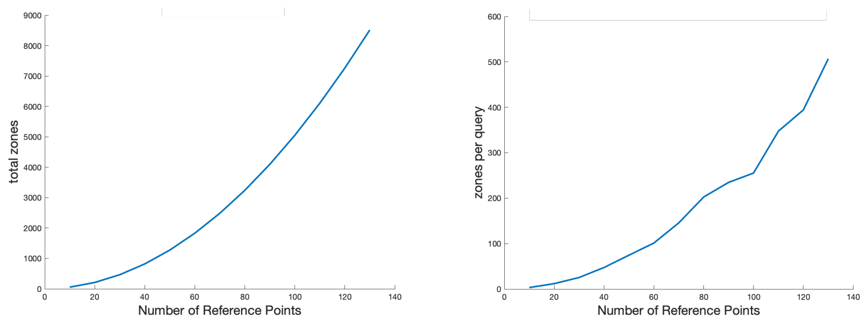

Figure 9: The effect of increasing the number of reference objects on the 20-dimensional Euclidean space. The X-axis represents the number of reference objects used. In the left-hand sub-figure shows the total number of EZs available for selection, i.e. $\binom{n}{2} + n$. The right-hand sub-figure shows the mean number of these EZs selected per query.

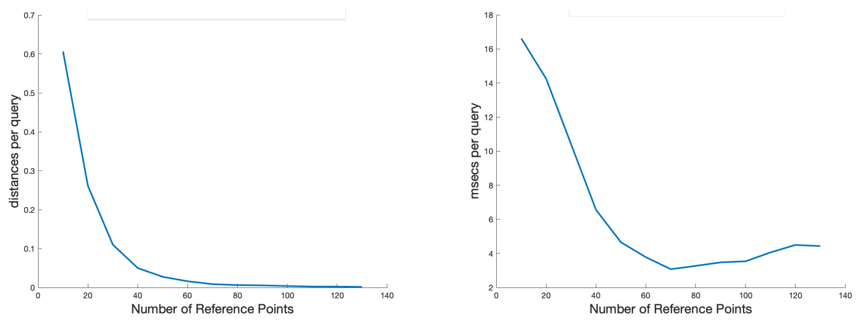

Figure 10: The effect of increasing the number of reference objects on the 20-dimensional Euclidean space. The X-axis represents the number of reference objects used. In the left-hand sub-figure shows the number of residual calculations required for Phase 3, and he right-hand sub-figure shows the mean elapsed time per query.

objects, the increase in the Phase 2 cost starts to outweigh the saving in the Phase 3 costs. This "sweet spot" can in general be established by experiment. We show an example of this for a different metric space in the following section.

8.1. Jensen-Shannon example

The BitPart mechanism will be most useful in metric spaces where the dimensionality is too great to allow effective indexing, yet not so high that no traction is available from the metric properties, and in particular where the inherent cost of a distance measurement is sufficiently great to make the cost of the bit mechanism worthwhile. We now show the example of performing search using the Jensen-Shannon metric over l_1-normalised 20-dimensional spaces. For this space a threshold of 0.126 was calculated to give approximately one result per million data, and one thousand queries were executed over one million data.

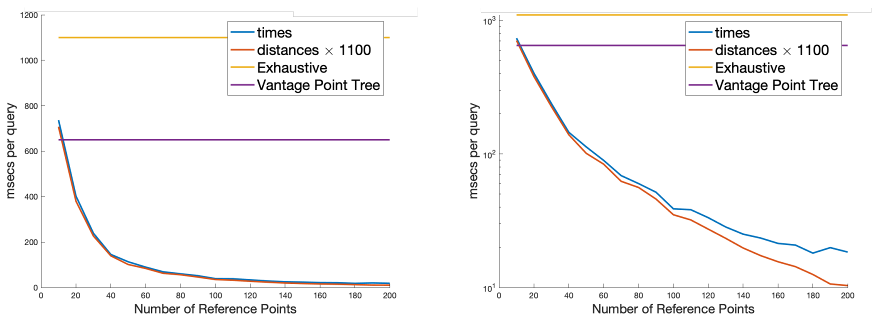

Figure 11: Times and distances for increasing numbers of reference points with 20-dimensional Jensen-Shannon distance, shown with log-scaled Y-axis on the right. It can be seen that the residual distance costs for this expensive metric almost completely outweigh the BitPart mechanism cost until a very large number of EZs is used.

The Jensen-Shannon metric has the four-point property; the IDIM of this space is 20.2.

In an exhaustive search, the elapsed time per query is 1100ms, i.e. a cost per distance calculation of 1μs, around 100 times greater than Euclidean distance over the same data. When the data is used to build a Vantage Point Tree, the time per query is 650ms; only 47.3% of the data is accessed, i.e. the mechanism cost in the context of this space is around 130ms per query over and above the cost of making the distance measurements.

For BitPart, the results using different numbers of reference points are shown in Figure 11. As mentioned above, a "sweet spot" exists where the number of zones being selected per query is optimal; in this case it is around 180 reference points, when an average of 567 of the 16290 available EZs are selected per query; at this point, only around 1.1% of the data is accessed, resulting in a mean query time of 18ms. Note that this figure shows that the mechanism cost is around 6ms, in return for a saving of the cost of $9.9e^5$ distance calculations per query, giving over 100 times speedup.

9. Parallelisation

In the reference implementation of the algorithm the Regions data structure is stored column wise with each column encoded as a Java BitSet. Whilst this makes the algorithm easier to understand and prototype, it does not lead to an efficient implementation due to the stress it places on the memory system of a modern processor. Recall that each row-wise operation requires the logical *and* of the B_{in} sets combined with the negation of the logical *or* of the B_{out} sets. If each of the inclusion bitsets is stored column-wise the row wise operations require many disparate locations to be accessed resulting in poor locality of reference and therefore relatively poor performance. A further problem with this arrangement is that the bitwise operations must be performed one row at a time due to the need to combine the single bits from the columns.

9.1. SIMD Parallelism

The first optimisation that may be made to the implementation is to store the bits in a row-wise rather than column-oriented fashion. The first benefit of doing this is to obtain better locality of reference. However storing rows as vector of bits still requires the operations to be performed sequentially. Most modern processors can perform bitwise operations in parallel. For example, since 2013 the Intel X86 family supports Advanced Vector Extensions (AVX) which support SIMD bitwise operations over data of various sizes (at the time of writing up to 512 bits). On the hardware available to us, storing the exclusion data structure as 2 dimensional arrays of long values with the primary index being the row and the secondary the column permits 64 (size of long) bitwise operations to be processed in parallel rather than sequentially. On more modern processors and if the code were written in a lower level programming language (such as C) 512 bitwise operations could be executed in parallel on a single core.

9.2. Thread level Parallelism

Operation/word level parallelism is not the only parallelism to which the algorithm is amenable. Since the bitwise operations are essentially independent, blocks of bits may be evaluated in parallel. Thus, if n cores are available on some processor, in principle, the rows may be partitioned into n slices and the bitwise operations may be performed in parallel using n threads. This approach is similar to the well-known Map-reduce pattern in which the map operations are executed in parallel rather than sequentially. However, when this approach was implemented, on an Intel(R) Xeon(R) CPU E5-2630 running SElinux with 2 CPUs (each with 6 cores with 2X hyperthreading) (giving 24 virtual cores) and 132 GB of memory in 2 banks) the parallel code ran slower than the sequential version.

The increase in speed for the parallel version is due to the Non Uniform Memory Access (NUMA) architecture on which the code was being executed. NUMA is the most prevalent architectural pattern in usage today and as the name suggests, the access time to memory is non-uniform. For this algorithm (and any algorithm with large shared data structures), if possible, the memory should be allocated on the same processor as the threads which access it. If not, high latencies are experienced when the processor is required to access data stored on a remote memory bank. Therefore rather than have a single data structure created by one thread and accessed by many, the logical conclusion is to make the threads create the data partitions on which they operate. Unfortunately, and somewhat surprisingly, there is a lack of good programming models to support such a memory model. We therefore implemented an actor model to support the required data locality. Each slice abstracts over a partition of a number of rows from the original monolithic data structure. Slices extend Java Threads and are responsible for creating and accessing the encapsulated data.

```
public interface BlockSlice {
    Future<Object> buildSliceBitSetData();
    Future<List<T>> querySlice(double threshold,
                               Metric<T> metric, T q);
    void terminate ();
}
```

Figure 12: The BlockSlice interface

Each slice exports three methods which are called by a choreography thread these are: buildSlice(), querySlice() and terminate() as shown in Figure 12.

Each of the 3 interface methods enqueues a message on an internal message queue which is in turn serviced by an internal thread. This mechanism ensures that the data and the threads are located on (glued to) the same bank of memory associated with a single core. When an operation is requested of a slice a Future is returned. This decouples the choreography thread from the worker threads within the slices and permits parallel execution. Thus when a query is executed, the choreography thread calls the *querySlice* interface methods in each slice and collects the results from the Futures as they become available. Lazy list concatenation of the partial results which creates lists of lists conforming to the List interface results in efficient results collection.

Figure 13 shows the performance of three different parallel implementations of the code. In each case the degree of parallelism was varied between 1 and 24. Recall that the experimental machine has 12 real cores on 2 CPUs and support for 24 hyper-threads. The data used for this experiment was one million 20 dimensional Euclidian array of doubles with a Gaussian distribution for each of the coordinates. The intrinsic dimensionality of this data set is around 27. 50 reference objects were used yielding 1475 exclusion zones (5 ball radii per reference object). Each test consisted of 1000 random queries with a threshold of 0.6 (designed to return 1% of the data) and each test was repeated ten times.

The first (V1) shows a parallel version of the reference code base. The bits are stored column wise in Java BitSet data structures and row-wise operations are performed on the bit matrix by between 1 and 24 threads. As can be seen, there is some performance improvement for a small number of threads but above 4 threads the time taken per query is not significantly less than using a single thread.

The second implementation (V2) has a similar structure to V1 but instead of storing the bits in a column-wise fashion, they are stored row-wise in arrays of longs with each long encoding 64 bits from a single column. The performance of this version is similar (although slightly worse) than the original.

The third implementation V3 follows the *BlockSlice* approach described above. As can be seen, the performance of this algorithm shows the performance increasing as the number of threads is increased. The increase of performance is less than linear however (1.92 improvement for 2 threads, 4.61 for 8 threads and 5.22 for 12 threads). It can also be seen that whilst the use of hyper-threading

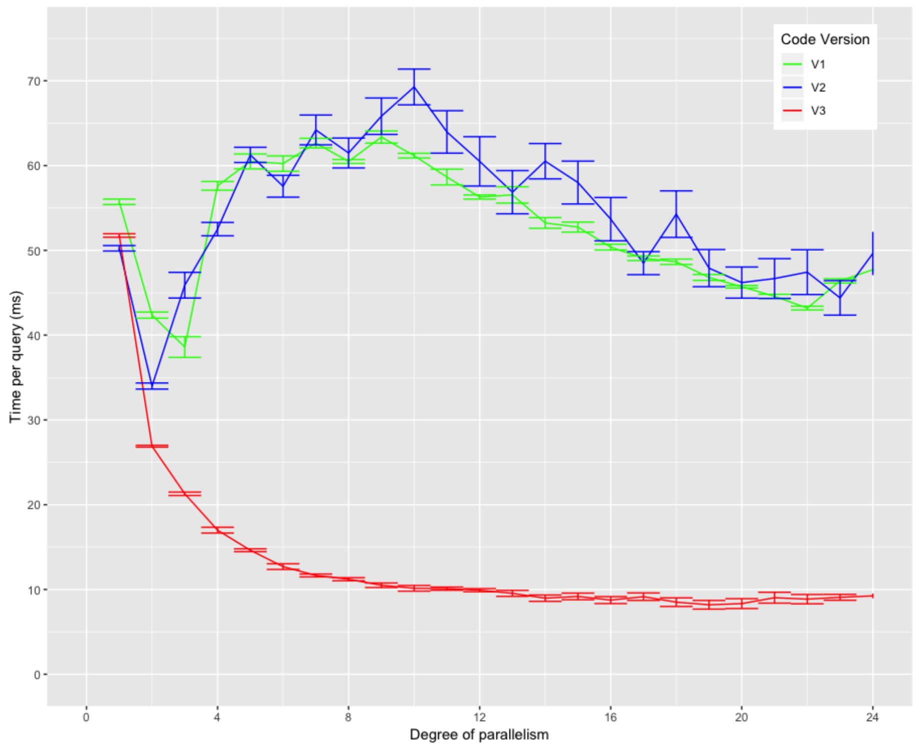

Figure 13: Parallel performance of three parallel code variations

does decrease the time per query a little, the improvement is less marked than is achieved with parallel cores.

10. Conclusions

This paper presents a novel exact search mechanism which is effective in metric spaces whose dimensionality is too high to allow effective navigational indexing. Rather than a navigational solution, we characterise the entire space according to a large number of binary partitions and use this structure to create an exact search mechanism.

The performance tradeoffs are interesting. It is well known that navigational indexing mechanisms start to fail as dimensionality increases, and also that, at a still higher dimensionality, all metric-based mechanisms fail. Furthermore, a recent theoretical result [**?**] shows that at a certain dimensionality, any query requires a linear scan of the data to be performed. Our mechanism inherently requires a linear scan, and therefore does not scale as well as navigational mechanisms at relatively low dimensions. However we have shown that it minimises the cost of this scan between the points where navigational indexes fail, up to the point where all metric-based mechanisms fail. In Section 8.1 we have shown an example speedup of two orders of magnitude for one example metric space within this region.

We have described the algorithm in three phases. The Phase 1 cost is independent of the data size, depending only on the number of Exclusion Zones defined, which may be tuned according to dimensionality and metric distance cost. Phase 2 theoretically uses space and time which are both $\Omega(n \log n)$ for data of size n. In terms of space, the theoretical minimum requirement is $n \log_2 n$ bits, and in one practical experiment described in Section 8 we demonstrate an optimum size of around 16 megabytes for a space of one million objects whose IDIM is 27.6, which is only one order of magnitude greater than the theoretical minimum. Even when $n = 10^9$ the computation can still fit within the memory of a modern laptop computer. In terms of time, we have shown in Section 9 that the computation for this phase is highly parallelisable.

Phase 3 comprises two separate costs: a sequential scan of the bitmap, and a residual distance calculation for each bit which remains set. For the bitmap scan, the space and time cost are both $\mathcal{O}(n)$. The space cost is fixed at precisely n bits, and the time cost is small on modern hardware.

For the residual scan, the practical cost depends entirely on the efficacy of the first two phases, as any number between 0 and n distance calculations may be required. This depends mostly on the dimensionality of the metric space and the number of available exclusion zones. In our example of Section 8 the proportion of residual calculations required is around 0.01 of n.

However it can be noted also that Phase 3 is essentially optional, and required only for exact search. As an alternative, the first two phases alone can be considered as an approximate search technique, with perfect recall but varying precision. Whether this is desirable or not depends on how well the data is characterised according to the selected regions.

The algorithm is inherently decomposable and parallelisable. Every phase of the computation is parallelisable: the distances between queries and reference objects, the assessment of queries against regions, the bitwise operations over the data representation, and finally the filtering of the candidate results. Amdhal's law applies to the end to end pipeline but we have demonstrated that great speedups are possible by exploiting parallel hardware resources due to the nature of the algorithm.

The algorithm exhibits further opportunities for tuning which are yet to be explored. We have demonstrated the effect of changing the number of reference objects and how this may help in combating increasing dimensionality. However it may be possible to employ techniques to choose pivots which could enable queries to perform a number of distance calculations that are closer to the theoretical minimum. We have experimented with choosing pivots to optimise sheet exclusions but reporting on this work here would also be premature. The size, number and uniformity of the radii used for ball exclusions is also worthy of exploration.

[1] Giuseppe Amato, Claudio Gennaro, and Pasquale Savino. Mi-file: using

inverted files for scalable approximate similarity search. *Multimedia Tools and Applications*, 71(3):1333–1362, Aug 2014.

[2] José María Andrade, César A. Astudillo, and Rodrigo Paredes. Metric space searching based on random bisectors and binary fingerprints. In Agma Juci Machado Traina, Caetano Traina, and Robson Leonardo Ferreira Cordeiro, editors, *Similarity Search and Applications*, pages 50–57. Springer International Publishing, 2014.

[3] Leonard M. Blumenthal. A note on the four-point property. *Bulletin of the American Mathematical Society*, 39(6):423–426, 1933.

[4] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[5] Richard Connor, Franco Alberto Cardillo, Lucia Vadicamo, and Fausto Rabitti. Hilbert Exclusion: Improved metric search through finite isometric embeddings. *ACM Transactions on Information Systems*, 35(3):17:1–17:27, December 2016.

[6] Richard Connor, Lucia Vadicamo, Franco Alberto Cardillo, and Fausto Rabitti. Supermetric search with the four-point property. In Laurent Amsaleg, Michael E. Houle, and Erich Schubert, editors, *Similarity Search and Applications: 9th International Conference, SISAP 2016, Tokyo, Japan, October 24-26, 2016, Proceedings*, pages 51–64. Springer International Publishing, 2016.

[7] Richard Connor, Lucia Vadicamo, Franco Alberto Cardillo, and Fausto Rabitti. Supermetric search. *Information Systems*, 80:108–123, 2019.

[8] Richard C. H. Connor and Alan Dearle. Querying metric spaces with bit operations. In Stéphane Marchand-Maillet, Yasin N. Silva, and Edgar Chávez, editors, *Similarity Search and Applications - 11th International Conference, SISAP 2018, Lima, Peru, October 7-9, 2018, Proceedings*, volume 11223 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2018.

[9] Karina Figueroa, Gonzalo Navarro, and Edgar Chávez. Metric spaces library. *Online http://www.sisap.org*, 2007.

[10] E. Chavez Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1647–1658, Sept 2008.

[11] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, August 2012.

[12] Jakub Lokoč and Tomáš Skopal. On applications of parameterized hyperplane partitioning. In *Proceedings of the Third International Conference on Similarity Search and Applications*, SISAP '10, pages 131–132. ACM, 2010.

[13] Karl Menger. New foundation of euclidean geometry. *American Journal of Mathematics*, 53(4):721–745, 1931.

[14] Vladimir Míc, David Novak, and Pavel Zezula. Improving sketches for similarity search. In *Proceedings of MEMICS 2015*, pages 45–57, 2015.

[15] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, January 1994.

[16] Hisham Mohamed and Stéphane Marchand-Maillet. Quantised ranking for permutation-based indexing. *Information Systems*, 52:163 – 175, 2015.

[17] Laura C. Rivero, Jorge Horacio Doorn, and Viviana E. Ferraggine, editors. *Encyclopedia of Database Technologies and Applications*. Idea Group, 2005.

[18] Aviad Rubinstein. Hardness of approximate nearest neighbor search. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1260–1268. ACM, 2018.

[19] Eliezer Silva, Thiago Teixeira, George Teodoro, and Eduardo Valle. Large-scale distributed locality-sensitive hashing for general metric data. In Agma Juci Machado Traina, Caetano Traina, and Robson Leonardo Ferreira Cordeiro, editors, *Similarity Search and Applications*, pages 82–93. Springer International Publishing, 2014.

[20] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[21] Eric Sadit Tellez and Edgar Chavez. On locality sensitive hashing in metric spaces. In *Proceedings of the Third International Conference on Similarity Search and Applications*, SISAP '10, pages 67–74. ACM, 2010.

[22] Wallace A Wilson. A relation between metric and euclidean spaces. *American Journal of Mathematics*, 54(3):505–517, 1932.

[23] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity search: the metric space approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.