Journal of Big Data

RESEARCH Open Access

# Polytope: an algorithm for efficient feature extraction on hypercubes

Mathilde Leuridan[1,2*], James Hawkes[1], Simon Smart[1], Emanuele Danovaro[1], Martin Schultz[2,3] and Tiago Quintino[1]

*Correspondence:
Mathilde Leuridan
mathilde.leuridan@ecmwf.int
[1]European Centre for Medium-Range Weather Forecasts (ECMWF), Reading, United Kingdom
[2]Department of Mathematics and Computer Science, University of Cologne, Cologne, Germany
[3]Jülich Supercomputing Centre, Forschungszentrum Jülich (FZJ), Jülich, Germany

**Abstract**
Data extraction algorithms on data hypercubes, or datacubes, are traditionally only capable of cutting boxes of data along the datacube axes. For many use cases however, this returns much more data than users actually need, leading to an unnecessary consumption of I/O resources. In this paper, we propose an alternative feature extraction technique, which carefully computes the indices of data points contained within user-requested shapes. This enables data storage systems to only read and return bytes useful to user applications from the datacube. Our main algorithm is based on high-dimensional computational geometry concepts and operates by successively reducing polytopes down to the points contained within them. We analyse this algorithm in detail before providing results about its performance and scalability. In particular, we show it is possible to achieve data reductions of up to 99% using this algorithm instead of current state of practice data extraction methods, such as meteorological field extractions from ECMWF's FDB data store, where feature shapes are extracted a posteriori as a post-processing step. As we discuss later on, this novel extraction method will considerably help scale access to large petabyte size data hypercubes in a variety of scientific fields.

**Keywords** Data management, Data processing, Data extraction, Datacube, Computational geometry

## Introduction

In the past century, fields in science and technology have entered a new era - the era of "big data" [1]. From weather forecasting to astronomy and medicine, scientific advances have led to a surge in the quantity of data produced daily. Indeed, scientific data has been steadily growing in the past decades and in recent years especially, it has experienced exponential growth [2, 3]. y promises for major scientific developments in the years to come, the question arises of how to efficiently use this wealth of data.

The scientific data collected nowadays often depends on a number of different variables and can thus be represented as a multidimensional array, or datacube [4]. Organising data inside such datacubes has attracted a lot of interest in the past few years, with many tools now available to work on such data representations [5–8]. Most modern software architectures provide support to handle such data structures, from cells in

Matlab [9] to Xarray [10] in Python and xtensor [11] in C++. However, in each of these tools and software, datacubes can only be accessed orthogonally to their axes by selecting specific values or ranges along given dimensions. This can most easily be seen in datacubes which use SQL-type queries through the WHERE keyword [12–15], but it also holds true more generally for other datacube implementations which only support axis-aligned subsetting operations on the data [16–19].

Such limited data access mechanisms in the form of bounding boxes are sub-optimal for a wide range of applications. Consider for example the case where a user wants to access temperature data over a country. For this particular example, the bounding box data extraction approach proves to be quite inconvenient as country shapes are not well-represented by bounding boxes. Alternatively, consider a user who wants to access data over some spatio-temporal path [20]. If the entire bounding box of data around the requested path is extracted, the user then has to mask out the wanted path in order to find the data points relevant to his application [21]. This thus not only implies that much more data than is necessary is read and returned from the datacube, thereby consuming more I/O resources, but it also places the additional burden of post-processing on the user after retrieval.

To address this issue, we introduce a new alternative way of accessing datacubes. Our extraction algorithm, Polytope[1] [22], enables users to efficiently query arbitrary high-dimensional shapes from a datacube, slicing non-orthogonally at arbitrary angles against the datacube's axes.

As shown with the country slicing example, traditional bounding-box extraction methods are insufficient for handling complicated non-rectangular requests. The Polytope algorithm however was designed especially with these requests in mind and is able to directly extract such shapes from very large data hypercubes. Because our algorithm computes the exact data points that users are interested in and only reads those from the datacube, it scales well to large high-dimensional request shapes unlike the traditional bounding-box extraction techniques which scale with the tensor product of each dimension. The Polytope algorithm will thus enable scientists to efficiently make use of their ever increasing data, whilst improving the efficiency of their I/O system.

The Polytope algorithm will be used as part of the Polytope service in both the ECMWF Software EnginE (ESEE), which facilitates all data provision and workflow management services at the European Center for Medium-Range Weather Forecasts (ECMWF), and the Destination Earth (DestinE) Digital Twin Engine (DTE) [23].

In this paper, we first introduce the idea behind the Polytope algorithm, before describing its inner mechanism in detail. We then expose some of its possible applications in different scientific fields before finally performing an analysis of the algorithm's performance on selected test cases.

## Concept

Before diving into a technical description of the algorithm, let us first explain in more detail the conceptual approach we take. With the Polytope algorithm, we developed a data extraction algorithm which supports the retrieval of arbitrary high-dimensional request shapes, called *features*, from arbitrary multi-dimensional, high-volume data

[1] https://github.com/ecmwf/polytope

hypercubes. Our algorithm is not restricted to any particular request shapes or application field and is in fact intended to be generic and work seamlessly in any scientific application involving multi-dimensional datacubes.

## Motivation

Rather than pre-defining a set of shapes which can be extracted from the datacube, the Polytope algorithm takes n-dimensional *polytope* shapes as input, giving the algorithm its name. In computational geometry, a polytope (or bounded convex polytope) is defined as the convex hull of a given point set $\mathcal{P} = \{p_1, \ldots, p_n\}$ [24, 25].

Note that polytopes are convex by definition. Polytopes can in fact be thought of as multi-dimensional convex polygons.

In the Polytope software, we use polytopes because any arbitrary high-dimensional shape, even a concave shape, can be either approximated by or decomposed into simpler convex polytopes [26]. Indeed, polytopes can be seen as the building blocks of high-dimensional geometry. They form the basis of most modern meshing softwares and are widely used in computer graphics to model intricate objects [27]. We thus see that by formulating data requests as polytopes, users will in theory be able to request almost any feature of interest to them from a datacube.

The underlying idea behind the Polytope algorithm can be visualised in Fig. 1.

## Related work

There are many alternative data extraction methods based on spatial indices present in literature. Similarly to the solution we propose here, some of these methods were especially designed to extract polytopes, and 2-dimensional polygon shapes in particular, from spatial datasets.

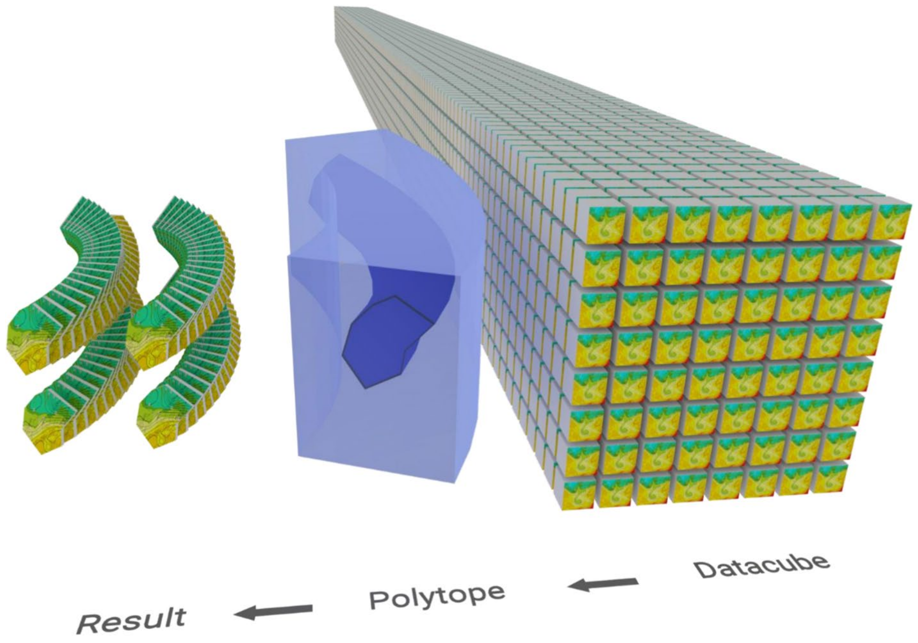

**Fig. 1** Concept of the Polytope algorithm. A 6-dimensional polytope stencil is used to extract data of this shape from the datacube. In dark blue, we picture the innermost 3-dimensional shape to extract, which is extruded into a 6-dimensional shape by taking its tensor product with the light blue shape over the outer 3-dimensional datacube

One popular such approach is to build R-trees [28] of 2- or 3-dimensional datasets to allow for efficient querying of an area. This approach already constitutes an important improvement to more traditional bounding box methods where the entire bounding box of the area is extracted, as it allows the system to extract a smaller bounding box coverage of the requested polygon. For very large high-resolution datasets however, this method might still return many more data points than needed and would require expensive point-in-polygon tests to further refine the output to only the points of interest. In particular, this method strongly depends on the constructed R-tree and its depth and resolution. As the R-tree is a static object for each dataset, the initial choice of how it is constructed will therefore significantly affect the subsequent extraction speeds of different areas. For example, when extracting smaller polygons, a larger R-tree with smaller leaf bounding boxes would return data closer to the user request, but having such a larger tree would also make extraction more costly for larger polygons. Our proposed Polytope algorithm does not require building such a static object and the extraction thus remains completely dynamic regardless of the requested shape. Furthermore, as the proposed Polytope algorithm only returns the points of interest, no post-processing masking operation is needed either.

Another alternative method of interest here is the RasDaMan [15] approach. RasDaMan first subdivides datasets into rectangular tiles to then be able to read those independently and perform queries on them once loaded in memory. Similarly to R-trees, this method implies that a rectangular coverage of the shape of interest is accessed. The main aim of the Polytope algorithm is to prevent such unnecessary data access and to instead pre-compute the points of interest in a file before accessing them, which results in a very different extraction method. Furthermore, RasDaMan and other similar methods, such as the clipping operation available in PostGIS [29], only support 2-dimensional polygon masking as a post-processing operation. In comparison, the Polytope method described in this paper has many more degrees of freedom and allows the extraction of arbitrary polytope shapes in any dimension.

## Polytope extraction algorithm

We now introduce the Polytope feature extraction algorithm, highlighting in particular the way in which it achieves polytope-based feature extraction on datacubes.

Note that, even though the Polytope algorithm is quite generic, the datacubes on which it operates need to possess some particular properties. We thus first discuss some of these datacube properties before describing the complete mechanism behind the feature extraction algorithm.

### Datacube

Datacubes can be thought of as multi-dimensional arrays. In particular, they store data points along different datacube dimensions [30]. Each datacube dimension has an associated coordinate metadata "axis" with a discrete set of indices stored on it [31]. A data point is then located at each of these indices, forming a datacube. Datacubes can store structured point clouds or raster data for example.

A datacube can have different properties depending on the data that it stores. It can for example exhibit splitting behaviour when two datasets are incompatible, be unbalanced when some experiments gather more data than others or even have gaps between data points if results are not regularly recorded. In this section, we identify and describe these properties, as well as other essential datacube features, such as the datacube axes.

This helps us construct a common framework for treating various types of datacubes throughout our algorithm.

### *Axes*

Axes in a datacube refer to the coordinate dimensions along which the data is stored. Values along these axes are called indices. In the following, we differentiate between two main types of axes, the ordered and unordered categorical axes. These two types of axes cannot be treated in the same way within the slicing step of the algorithm, which leads to their distinction here.

**Ordered Axes** These axes only accept sets of comparable indices which can be ordered. In particular here, this means that values on ordered axes need to be comparable to each other, such that they must meaningfully support comparison operators ($==, <, \leq, >, \geq$). This property then directly implies an ordering between indices on ordered axes. Importantly, note that indices on ordered axes do not have to be integers, but can in fact be any countable type that supports a comparison operation, such as time entities, floating point numbers and of course integers. For such axes, it is possible to query ranges of indices as well as individual axis values.

**Categorical Axes** The other type of axes which can be handled by our algorithm are categorical axes. These axes only support distinct indices which are not comparable to each other, such as string indices for example. In this case, unlike for ordered axes, it does not make sense to query ranges of indices. Instead, the only possible queries on categorical axes are specific index selections.

Note that, in practice, indices on a datacube will always be discrete as there will always be a gap between values of digitally stored data. All ordered axes are thus countable axes, for which indices can be ordered and numbered using natural numbers [32]. Note also that the indices on ordered axes do not have to be uniformly spaced. In particular, the datacube axes can be irregular and sparse in their indices [33]. Lastly, observe that ordered axes can exhibit special behaviours, such as cyclicity along their indices. These behaviours are applied as post-processing operations on the axis indices and therefore do not modify the ordering of the axes.

All categorical axes can be treated in the same way. Similarly, all ordered axes can also be treated in the same fashion. This allows us to take a common approach towards extracting indices on those axes and thus facilitates the data extraction algorithm.

### *Datacube structure*

A datacube can be viewed as a possibly non-regular imbalanced tree. This can be seen in Fig. 2 and we now explain each of these two datacube properties, non-regularity and imbalance, in more detail with the help of an example.

Note that the datacube does not necessarily have the same dimensionality in all directions. On some axes, it is possible to have axis indices which give rise to different subsequent axes or axis values. Consider for example the datacube in Fig. 2 which has indices **val4** and **val5** on its **ax2** axis. If we pick index **val5**, the leaf datacube is 2-dimensional with axes $u$ and $v$, whereas if we pick index **val4** instead, the leaf datacube is 3-dimensional with axes $x$, $y$ and $z$. This phenomenon can be viewed as a non-regular branching of the datacube axes. This is an important feature of the datacube, which we should take into consideration when thinking

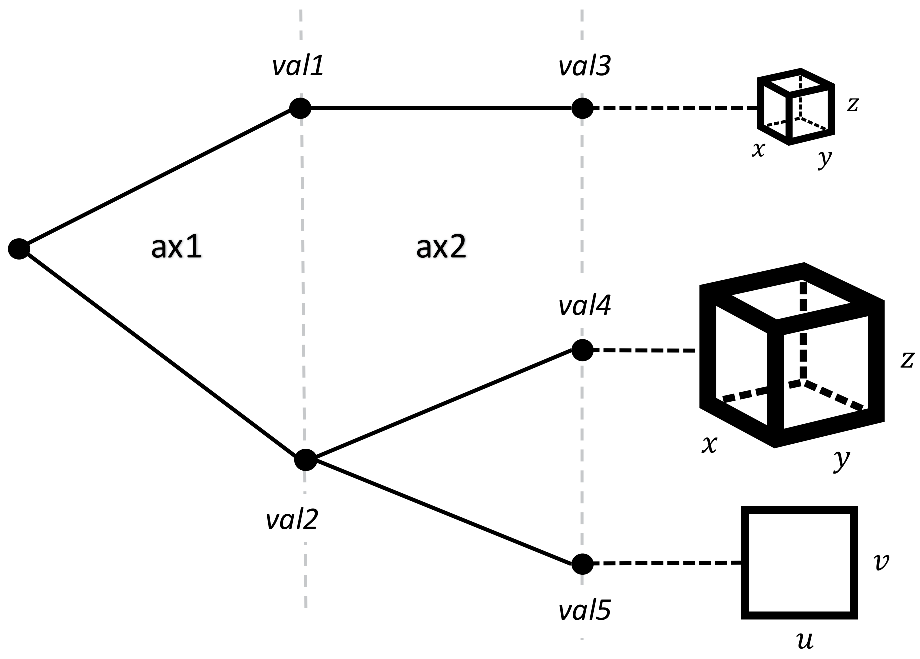

**Fig. 2** Datacube represented as a non-regular imbalanced tree. Each tree layer represents different axis dimensions with the nodes being corresponding axis index values. Here, the first axis dimension is ax1, which has indices val1 and val2. The second axis dimension ax2 with indices val3, val4 and val5 then branches out onto datacubes of different sizes and dimensions, causing both imbalance and non-regularity in this tree

about the datacube structure. In particular, this suggests that there is a natural ordering of the axes, which we should follow when extracting data.

The imbalance in the datacube tree comes from the fact that some datacube axis can possibly have many more indices than others. In our example datacube above, imagine for instance that the $u$ and $v$ axis each only have 200 index values, whereas the $x$, $y$ and $z$ axis each have 1000 index values. This implies that the **val4** index has many more children than the **val5** index and makes this particular datacube very imbalanced.

## Slicer

The core of the Polytope feature extraction algorithm is the *slicer*, which contains a novel slicing step on the datacube indices. The slicing algorithm introduced here is of particular relevance, as it supports non-orthogonal slicing across arbitrary ordered axes. This is in contrast to most current state of practice data extraction techniques, which often only cut boxes of data [34–36], whereas our slicing algorithm has the capability of cutting polytopes of data. Importantly, note that, as the algorithm introduced here is able to handle shapes of arbitrary dimensions, it can be used to extract both low- and high-dimensional data queries.

### *Concept*

By leveraging results in the field of computational geometry, the slicing algorithm used in Polytope can extract any convex polytope from the original datacube. The underlying concept is that we successively slice the requested polytope along each axis in the datacube's natural axis ordering using hyperplanes, reducing the dimensionality of the polytope at each step until we are left with a list of all points contained in this polytope.

### *Ordered vs categorical axes*

As mentioned earlier, the slicer handles ordered and categorical axes slightly differently.

In particular, categorical axes do not support range queries and thus we can only ask for specific values on these axes, instead of polytopes. For categorical axes, the algorithm therefore only has to check whether the queried indices exist in the datacube, as would happen in every other traditional extraction algorithm.

The true innovation of the Polytope extraction technique is its ability to handle arbitrary polytope requests, which it achieves by introducing a new slicing step along the ordered axes. Note however that this slicing technique only works on ordered axes for two reasons.

Firstly, since it is only possible to define and request ranges on ordered axes, it also only makes sense to define polytopes along such axes. Secondly, the slicing step introduced below only works on indices which can be interpolated. As we now explain, these are in fact precisely the ordered axes' indices. Indeed, note that, for the purposes of our algorithm, we assume that all of the ordered axes are measurable and linear axes, which can have continuous index values. We make this assumption even for ordered axes which are only truly countable with gaps between their indices. Because all ordered axes have some comparison operation, this is a valid assumption. This then implies that ordered axes support interpolation on their indices, which is needed for the slicing step described in the next subsection.

### *Slicing step*

The actual slicing step is quite straightforward with the slicing mechanism merely consisting in finding the intersection of a polytope with a hyperplane along a datacube axis.

We first separate all vertices in the polytope into two separate sets, $P_{\leq}$ and $P_{\geq}$, each set consisting of points on either side of the hyperplane. For each pair of vertices $(p_{\leq}, p_{\geq})$, where $p_{\leq} \in P_{\leq}$ and $p_{\geq} \in P_{\geq}$, we take the intersection of the line passing through these two vertices and the hyperplane. This line-hyperplane intersection results in an intersection point which lies on the hyperplane. Once we have done this for all pairs, we therefore obtain a lower-dimensional polytope on the hyperplane, which is in fact just the intersection of the original polytope with the hyperplane, as wanted. This can be seen in Fig. 3 for a 2D example and in Fig. 4 for a 3D example.

As the original polytope is convex, this new intersection polytope is trivially also convex. As an optimisation step, we can thus take the convex hull of the intersection points at the end, using the QuickHull algorithm [37] for example. This does not change the lower-dimensional polytope because it is convex, but removes all interior vertices in its definition. As we slice high-dimensional polytopes, this can lead to major performance improvements. Indeed, without this last step, the number of vertex points in the polytope definition grows quadratically with each slice, which would considerably slow down the algorithm. However, even though this optimisation step can significantly help the performance of the algorithm, the line-hyperplane intersection operation is still quadratic in the number of polytope vertices. The algorithm's performance thus scales at least quadratically with the number of vertices, which will impact its runtime for input polytopes with a large number of vertices.

It is important to note here that this slicing step works on all ordered axes, without any specific constraints about the type of indices stored on these axes.

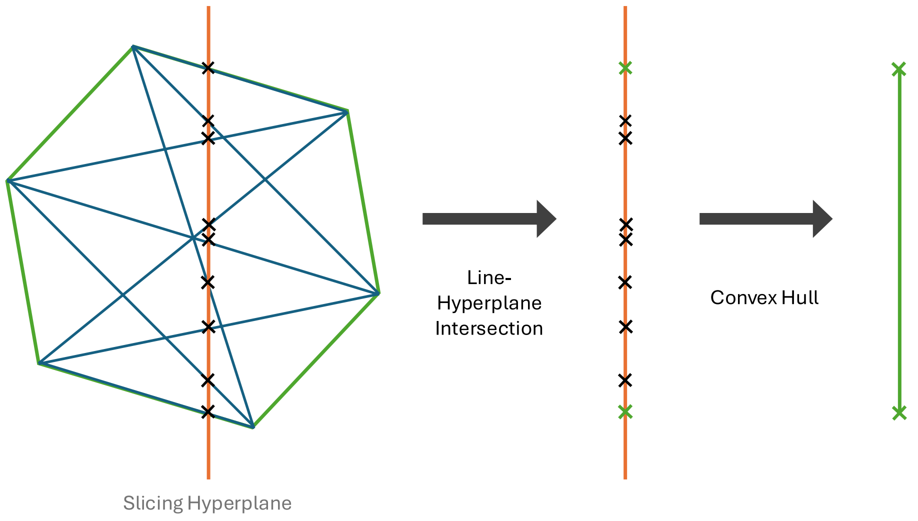

**Fig. 3** 2D example of the Polytope slicing mechanism on ordered axes

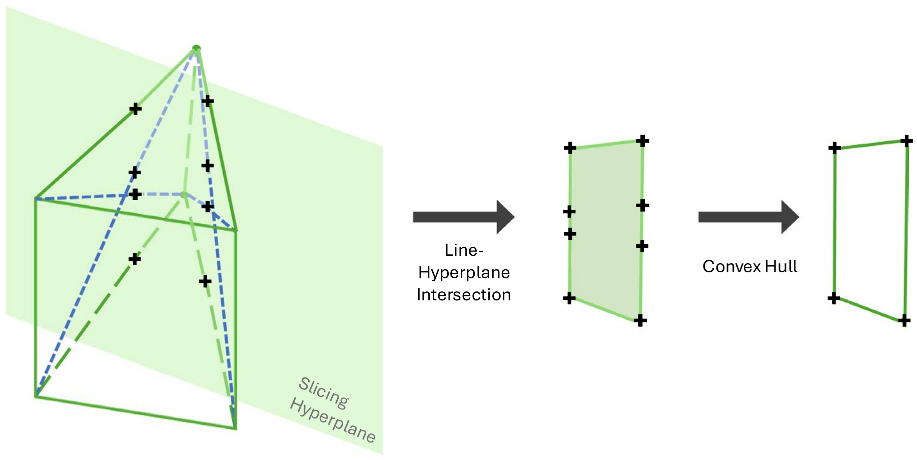

**Fig. 4** 3D example of the Polytope slicing mechanism on ordered axes

***Index tree construction***

To ensure that we slice through all the requested polytopes defined on different axes of the datacube, we need to carefully keep track of which step in the extraction we are in. The way we achieve this in the Polytope extraction technique is to iteratively build a trie compressed set of results, where the indices are the found datacube axis indices found inside of the polytope shape. In the following, we refer to this trie as an index tree.

We build the index tree by slicing along successive axes in the datacube's natural axis ordering one after another. For each axis of the datacube, we first find the polytopes defined on that axis. For each of these polytopes, we find the extents $(low, high)$ of the lowest and largest value of the polytope's boundary along the axis. We then collect all discrete indices contained within the extents $(low, high)$ on the axis and add them as children to the index tree. Next, we slice the necessary polytopes along each of the discrete datacube indices to obtain lower-dimensional polytopes. As shown in Fig. 5, these lower-dimensional polytopes are the intersection of the higher-dimensional polytopes with each of the axis indices slice hyperplanes. These new polytopes are the next polytopes we would like to now extract from the datacube. The algorithm therefore continues as before on these lower-dimensional polytopes if they exist. This process is re-iterated in Algorithm 1.

Note that this works well on ordered axes. On categorical axes however, the slicing step is ill-defined as interpolation between indices is not possible. Nevertheless, recall that polytopes defined on categorical axes are in fact 1D points and thus instead of slicing, we only need to check whether those points exist in the datacube. Indeed, slicing does not matter in this case as the points are 1 dimensional and slicing, if it were well-defined, would therefore not produce any lower-dimensional polytopes anyway. We thus conclude that the process for constructing index trees presented in Algorithm 1 does in fact work well for categorical axes as well.

Algorithm 1 implies that we construct the index tree breadth-first (layer by layer), instead of depth-first (constructing branches one after the other). This approach ensures that the algorithm does not loose track of what values inside the requested polytopes have already been found. It thus ensures that users get back all the points that are contained in the shape they requested.

To gain a better understanding of how Algorithm 1 works in practice, we provide a detailed example of its mechanism on a small datacube in Appendix A.

**Algorithm** 1: Polytope Slicing Algorithm

```
Input: list of polytopes $\mathcal{P}$,    Output: Index tree of results $T$

for polytope $p$ in $\mathcal{P}$ do
    Remove duplicate points in $p$
end for
Instantiate root node $r$ of an empty index tree $T$
Initialise a set $\mathcal{P}_{unsliced}$ of unsliced polytopes associated to $r$ as $\mathcal{P}^r_{unsliced} := \mathcal{P}$
Initialise set of current index of tree nodes $current\_nodes := \{r\}$
for $axis$ in datacube axes do
    Initialise set of next nodes to consider $next\_nodes := \{\}$
    for node $n$ in $current\_nodes$ do
        Find subset of polytopes $\mathcal{P}_{defined}$ on $axis$ from $\mathcal{P}^n_{unsliced}$
        for polytope $p$ in $\mathcal{P}_{defined}$ do
            Find extents $(low, high)$ of $p$ on $axis$
            Find set $\mathcal{I}_{axis}$ of datacube indices between $(low, high)$ on $axis$
            for index $i$ in $\mathcal{I}_{axis}$ do
                Add a child $c$ to $n$ with $c$.axis := $axis$ and $c$.value := $i$
                Initialise the set $\mathcal{P}^c_{unsliced} := \mathcal{P}^n_{unsliced} \setminus \{p\}$
                if $axis$ is a categorical axis then
                    Skip
                else if $axis$ is an ordered axis then
                    Slice $p$ along $i$ to get lower-dimensional polytope $p_{reduced}$
                    Add $p_{reduced}$ to $\mathcal{P}^c_{unsliced}$
                end if
                Add $c$ to $next\_nodes$
            end for
        end for
    end for
    Update $current\_nodes := next\_nodes$
end for
Return final index tree $T$
```

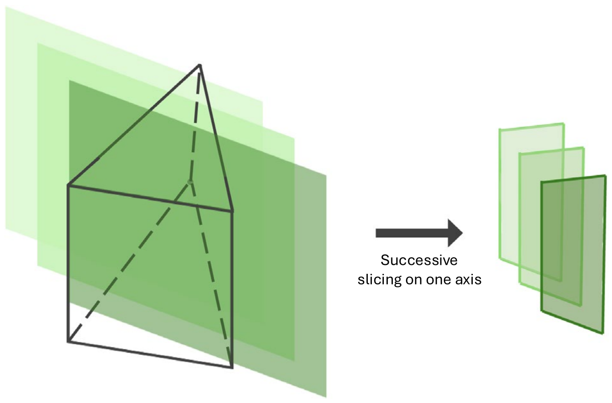

**Fig. 5** Successive slicing of a polytope along one axis at different indices, resulting in a list of lower-dimensional polytopes

## Applications

The Polytope data extraction algorithm has a wide range of interesting applications, from meteorology to healthcare. In this section, we first introduce the different Polytope interface levels before discussing some Polytope applications, describing specific examples and how Polytope has improved access to data in those cases.

### Interface

To facilitate interaction with the Polytope feature extraction algorithm, which only accepts polytopes as input, different interfaces can be implemented. The Polytope interfaces allow the users to interact with the extraction algorithm. In particular, users will submit their request shapes and, after the algorithm has run, retrieve their desired data to and from these interfaces. To accommodate different types of users, several interface levels exist, which let users request a wide range of request shapes, from the low-level generic convex polytope to higher-level specialised requests. The two in-built low- and high-level Polytope interfaces are shown in Fig. 6. It is also possible to build domain-specific interfaces on top of these built-in interfaces, also shown in Fig. 6. Each level is built on top of another with the domain-specific interfaces using shapes from the high-level interface, which itself depends on the low-level interface.

These distinct interface levels are useful because depending on specific needs and familiarity with the Polytope extraction technique, users might want to request different types of shapes from our algorithm.

Through the domain-specific interfaces, users can request domain-specific functions. For example, a meteorological interface could be built to facilitate access to time-series, trajectories or country extraction, similar to the Open Geospatial Consortium (OGC) Environmental Data Retrieval (EDR) [38] standard.

Through the built-in high-level interface, users can request primitive shapes, such as boxes or disks (approximated as dodecagons or 12-sided polygons), and then use constructive geometry operations, such as taking unions [39] or sweeping along a path [40], to build more complicated shapes. At this interface level, concave polygons are also supported by first triangulating the polygon and then taking the union of the resulting set of triangles. Finally, through the low-level interface, users can directly provide a list of convex $n$-dimensional polytopes, specified by a list of their vertices.

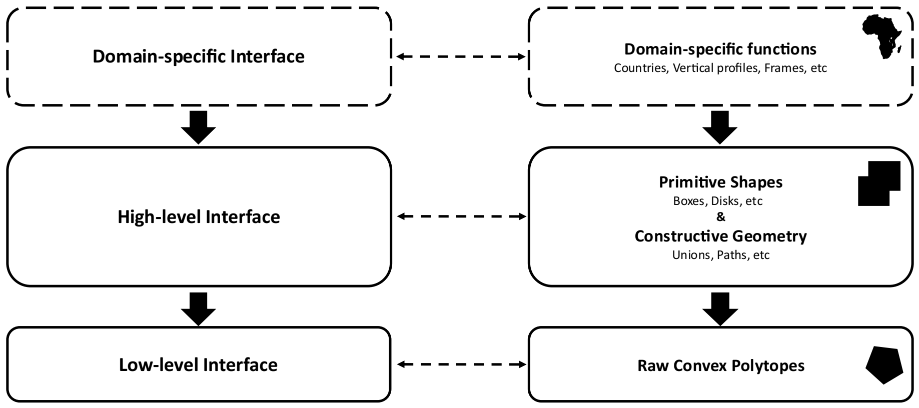

**Fig. 6** Polytope interface levels

Each level is built on top of its lower-level counterpart, so that shapes in a higher level are always defined by shapes in one of the lower levels. This implies that shapes in any of the interface levels are in fact always defined as a combination of convex low-level polytopes. These low-level polytopes are the building blocks of all possible Polytope requests. The interface is responsible for decomposing all user request shapes into these base convex polytopes. In the slicing algorithm, we can then work only on these convex polytopes and take a unified approach towards slicing any user request shape.

## Polytope use cases

We now describe some examples of how the Polytope feature extraction algorithm can be used in the fields of meteorology and healthcare.

### *Meteorology*

At ECMWF, about 400 TiB of Numerical Weather Prediction (NWP) data are produced daily [41]. This data is usually represented as a datacube of 7 or 8 dimensions depending on the forecast type [42]. Over the next few years, following the pioneering work of the DestinE initiative [43] with planned resolution increases in the weather model, data production will grow to over a petabyte of data a day [44].

The current data extraction mechanism implemented at ECMWF in their Meteorological Archival and Retrieval System (MARS) [42, 45] is one of the traditional bounding box approaches. When a user wants to extract data on a country for example, they would need to send a request for a bounding box around that country. Moreover, the current extraction technique requires either full data fields or at very best bounding boxes of data fields to be read from the system even when users only request a smaller portion of data. With future petabyte-scale datacubes, this approach will become impractical, especially when trying to accommodate for thousands of users. The Polytope extraction technique helps alleviate many of the challenges faced by the system in this case. It makes returning data to users much more efficient because only the required bytes are read from the I/O system.

Below, we provide a few practical examples and use cases where Polytope might help meteorological data users extract data more efficiently.

**Timeseries** Imagine a user interested in extracting the temperature over Italy for the next two weeks. She would currently have to transpose the temperature fields along the time axis to be able to then individually extract each temperature field at a given timestep. For each timestep, she would have to cut the shape of Italy from the bounding box she retrieved before finally getting the exact data she wanted. With the Polytope extraction technique, she can instead directly request the timeseries over Italy and get back only the precise bytes she is interested in. This is shown in Fig. 7, where the coloured points represent data points that are stored as byte indices within a data file and are accessed by the algorithm. Although Algorithm 1 only computes the coordinates of the data points, those coordinates can then be mapped to indices within a data file through a simple mapping. Polytope then accesses only those indices from the file instead of larger byte ranges, which are not of interest to the user. Note that compared to the 3D bounding box the user would currently retrieve, we see a data reduction of more than 73% when using Polytope. Furthermore, note that meteorological data users are usually more interested in extracting data over particular cities or specific points in space rather than whole regions, such as in [46, 47]. Since users currently first have to transpose their data and then retrieve bounding boxes around their locations of interest however, in most cases they directly extract data over broader regions than just the specific locations they would like to access. With the Polytope algorithm, expensive pre-processing manipulations before extraction, such as the ones discussed in [48], are not needed anymore and users only retrieve the relevant timeseries data from the datacube.

**Flight Path** Now, imagine a user interested in the flight conditions over his plane journey from Paris to New York. Using the current extraction technique, he would get back a 4 dimensional box over 3D space and time, containing much more data than what he is interested in. With the Polytope extraction technique, he will instead only get back the specific points he is interested in in the datacube without any need for post-processing, as shown in Fig. 8. Note that compared to the 4D box the user would currently get back, with Polytope, we experience a data reduction of more than 99.99%.

In both cases, the requested shapes are not axis-aligned and are therefore also not well approximated by bounding boxes. We thus see a significant data reduction when using the Polytope extraction technique compared to the traditional bounding box approach. Importantly, we observe that I/O is reduced when using the Polytope extraction

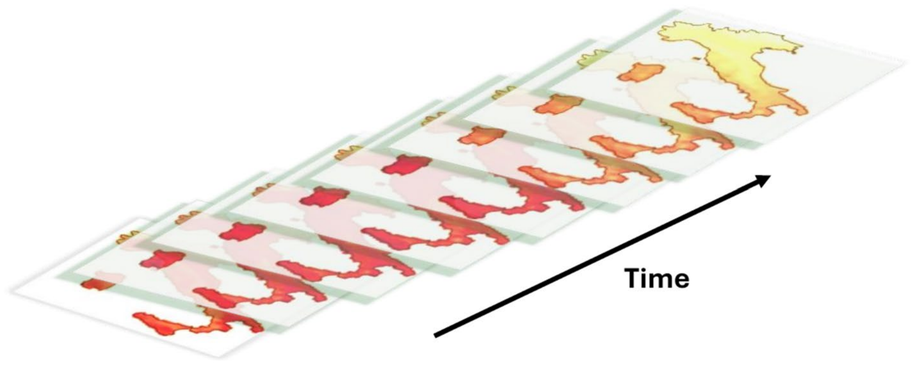

**Fig. 7** Timeseries example, where the coloured points are all the points that were extracted using the Polytope algorithm

algorithm. Moreover users do not need to perform any post-processing on their data anymore in order to get their requested shape.

### *Healthcare*

Similarly to the weather forecasting industry, the healthcare industry faces complex data handling challenges. Already in 2019, [49] estimated hospitals to generate tens of petabytes of data a year. As discussed in the previous example, working on this amount of data is extremely difficult and a tool like the Polytope algorithm could significantly help alleviate much of the difficulty involved.

A particular example of how Polytope can be used in the healthcare field is provided below.

> **Magnetic Resonance Imaging (MRI) Blood Vessel Detection** A clinically relevant application of MRI is the detection and characterization of plaque formation in (potential) stroke patients. This requires high-resolution scans using multiple MRI contrast weighting to comprehensively characterize the size and composition of plaque components. Using current extraction techniques, a clinician would have to download multiple entire MRI scan and then manually extract and compare the relevant data of interest from each of those scans. With the Polytope extraction technique however, it is possible to directly extract the required multi-contrast blood vessel data without further delay or expensive post-processing work, as is shown in Fig. 9 for a single high resolution black-blood vessel wall MRI dataset [50, 51].

## Performance and scalability

Polytope has been designed to considerably decrease the computational cost of extracting non-orthogonal data from petabyte-sized hypercubes. In this section, we justify this claim by first analysing the performance of the Polytope algorithm and then investigating the data reductions achieved on practical use cases when using the Polytope algorithm instead of traditional extraction methods. The Polytope algorithm implemented here is written in Python and operates without parallel processing. All experiments are run on a macOS-12.2.1-arm64-arm-64bit operating system with a CPython v3.9.19 Python interpreter. We conclude this section by discussing these results and their significance.

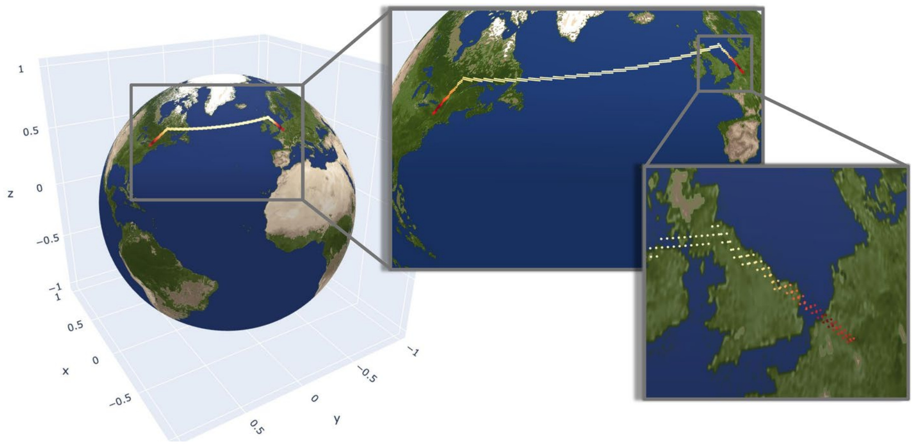

**Fig. 8** Flight path example, where the coloured points are all the points that were extracted using the Polytope algorithm

## Performance

There are many factors impacting the performance of the Polytope algorithm. To characterise it better, we identified some of the key features affecting how long the Polytope algorithm takes to extract points from datacubes: the number of extracted points, the dimension of the input shape and its geometry or how the input shape was constructed by the user. Note that, whilst there are many other important aspects which affect the algorithm's efficiency, such as the system's memory contingency, these are more system dependent and we thus do not study them here.

Consider two time quantities, the total slicing time and the algorithm run time. The slicing time is the total accumulated time spent just slicing, without constructing the index tree. The total algorithm run time however is the time it takes to perform all of Algorithm 1, including both the slicing time as well the time spent constructing the whole index tree. In Fig. 10, we have plotted both of these time quantities in different settings. In each subplot, we have varied one of the previously identified features and kept all others constant. This lets us gain an understanding of how each individual feature influences the performance of the Polytope algorithm. Note also that, although the algorithm influences data access patterns, we have not included the time taken in I/O to fetch the data from storage as this depends on the storage medium we use and is not strictly part of the Polytope algorithm.

In Fig. 10a, we first plot both the slicing time as well as the total algorithm run time for request shapes of different dimensions. We observe that the dimension of the shape does not significantly impact the algorithm's performance. This is due to the fact that, even when slicing higher-dimensional shapes, the Polytope algorithm spends most of its time slicing lower-dimensional polytopes. In fact, note that the number of polytopes to

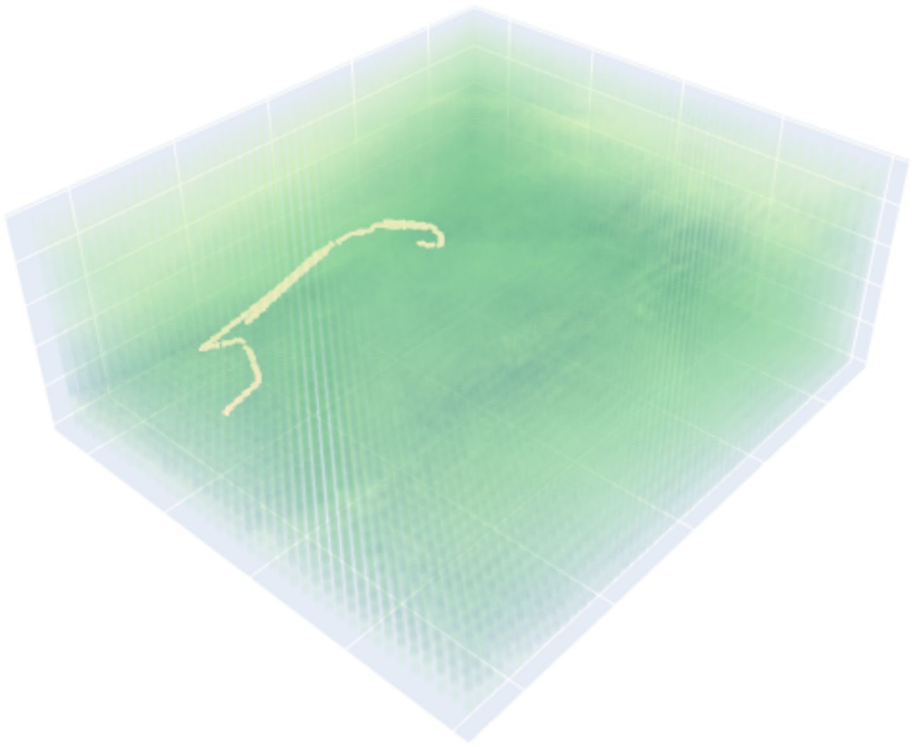

**Fig. 9** MRI scan example, where the white line is an extracted blood vessel, reaching from the cavernous segment of the internal carotid artery to the end of the middle cerebral artery

process at each step in the algorithm quickly grows every time we slice to a lower dimension. For example, imagine we slice a 4D box which contains two indices on each dimension. We first have to perform 2 4D slices. This then gives us 2 3D boxes, which we now have to slice. Each of these 3D boxes still has two indices on each dimension along which we need to slice. For each 3D box, we thus have to perform 2 3D slices. Considering there are 2 3D boxes, this implies we have to perform 4 3D slices in total. If we continue this logic, at the end of the algorithm, we will have performed 2 4D slices, 4 3D slices, 8 2D slices and 16 1D slices. This illustrates why the lower-dimensional slices do in fact take up most of the algorithm's slicing time.

Furthermore, we note in Fig. 10a that the slicing time is much lower than the total algorithm run time. This is because Polytope is currently using the XArray [10] library for datacube implementation, and relies on XArray to look up discrete axis indices on the datacube. This is a step we believe still needs to be optimised by developing more efficient datacube look-up mechanisms and alternative datacube implementations. Meanwhile, in Figs. 10b-10d, we use the slicing time rather than the total algorithm run time to estimate Polytope's performance, thus excluding this dependency.

Figure 10b shows the behaviour of the slicing time in more detail. In particular, we notice that like the total algorithm run time in Fig. 10a, the slicing time does not depend on the dimension of the input shape. We also observe that the slicing time grows linearly with the number of datacube points the algorithm finds in the input shape. As discussed

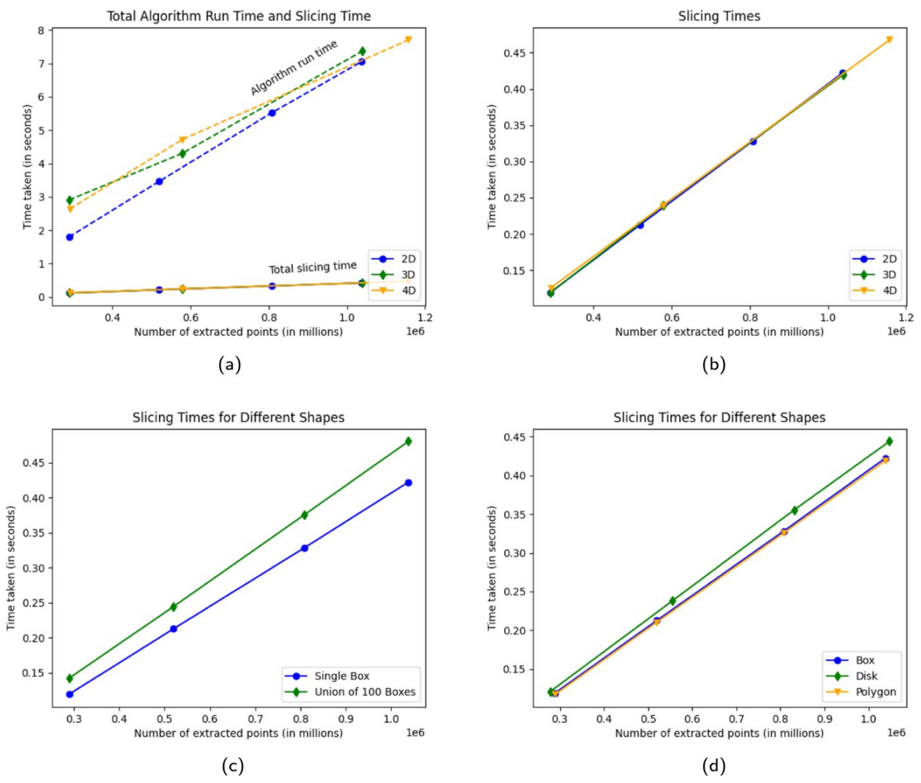

**Fig. 10** Performance plots of the time taken to run the Polytope algorithm in function of the number of points extracted by the algorithm. In (**a**), we plot both the total algorithm run time (dashed lines) and the slicing time (solid lines) for different numbers of dimensions. In (**b**), we plot the slicing times for different numbers of dimensions. In (**c**) and (**d**), we plot the slicing times for different shape types. In (**c**), we focus on the effect of constructive geometry operations, whereas in (**d**), we focus on the various primitive shapes implemented in the high-level Polytope interface. Algorithm timings on an Apple M1 Pro chip (3.2 GHz, 8 cores) and 16GB DDR5

before, this is due to the fact that most of the slicing time is spent performing 1D slices. Indeed, increasing the number of points contained in the shape is effectively equivalent to increasing the number of 1D slices to perform to find those points. As it is those slices that make up most of the slicing time anyway, it is thus natural that the performance of the algorithm grows linearly with the number of points contained in the shape.

In Figs. 10c and 10d, we now investigate the impact of the input shape's geometry and how it was constructed by the user.

In Fig. 10c, we first study how constructing a shape by taking a union of smaller sub-shapes affects the performance of the algorithm compared to directly specifying the input shape as one single object in 2 dimensions. We see that the performance when the shape is constructed using unions is worse than when the shape is specified as one single object. This is due to the fact that when we request shapes as unions of sub-shapes, we first slice each sub-shape individually in the algorithm before combining the results of these steps into one single output. Because we first slice each sub-shape individually, we in fact slice along all the sub-shapes edges. As the sub-shapes touch along their edges, we thus end up slicing along the edges several times, which increases the slicing time compared to when this does not happen in the non-union shape case. This is relevant where the input geometry has been produced via triangulation or mesh generation.

In Fig. 10d, we finally analyse how Polytope's different 2 dimensional high-level interface primitive shapes: the box, disk and polygon shapes, influence the algorithm's performance. Here, we observe that the box and polygon shapes perform similarly while the disk shape has a slightly worse performance than the other two shapes. Because the polygon shape we inputted is actually a square, we can conclude from this observation that the algorithm's performance is mostly impacted by the number of vertices of the shape, as well as how "non-orthogonal" it is. In particular, if the shape has many edges cutting across some axes, then it is less likely that there will be duplicates when we compute the intersection points in the slicing step. We will thus then have to perform more slicing later on in the algorithm. The same logic holds if there are more vertices in the shape, which also increases the number of intersection points computed in the slicing step.

## Bound on number of slices

As we just discussed, the time quantity of interest to us in evaluating the performance of the Polytope algorithm is the slicing time. The slicing time largely depends on a related quantity which is the number of slices performed during the algorithm. In this subsection, we quickly determine a theoretical upper bound to this related quantity.

Suppose we query an $m$ dimensional shape using Polytope and let $n_i$ be the maximal number of discrete indices stored along each of the $i = 1, \ldots, m$ axes of the datacube which are contained within the requested shape. Since we do not know a priori exactly how convex or non-box-like the requested shape is, we have to assume the worst-case scenario that the shape is in fact a box.

To find the datacube points within that worst-case box shape, the Polytope algorithm now first has to slice $n_1$ times along the first axis dimension. This produces $n_1$ $(m-1)$ -dimensional box shapes. We then have to slice each of these lower-dimensional box shapes $n_2$ times along the second dimension. This creates an additional $n_1 \times n_2$ slices.

If we continue this process up to the last 1D slices, we finally see that the number of slices performed during the Polytope algorithm, $N_{slices}$ is bounded by

$$\begin{aligned} N_{slices} &\leq n_1 + n_1 \times n_2 + \cdots + n_1 \times \cdots \times n_m \\ &= \sum_{i=1}^{m} \prod_{j=1}^{i} n_j \,. \end{aligned}$$

We see that, as expected, this upper bound is dominated by the number $\prod_{i=1}^{m} n_i$ of 1D slices, which will take up most of the slicing time. As we saw in the previous subsection however, 1D slices are relatively inexpensive to perform and in all of the examples in Fig. 10, the slicing time remains under a second.

## Data reductions

Although the Polytope algorithm represents an additional step to perform before extracting the data and it might thus at first glance seem like it has a much higher time complexity than traditional extraction approaches, it is important to remember the true purpose of the Polytope algorithm. Polytope is a tool which computes the precise bytes of data a user wants to access. Using this tool therefore implies that users only extract exactly the data points they need, which significantly reduces the number of points to be read from the I/O system compared to the alternative "bounding box" extraction techniques. This leads to faster data transfer times between systems, thus making this approach more time efficient when considered as a whole.

In this section, we analyse various data reduction statistics, which are indicators of the efficiency gains that can be achieved through Polytope on various datasets. The datasets used for the meteorological extractions are the NWP MARS archive datasets produced by ECMWF on a octahedral O1280 grid [42, 45]. For the extractions of the flight path from Paris to New York shown in Fig. 8, the point timeseries over a 14-day forecast range, and the vertical profile of a point extruded over 20 atmospheric pressure levels, the data is interpolated to a 0.25/0.25 regular latitude-longitude grid. For the box around Germany and country extractions, the data is interpolated to a higher-resolution 0.125/0.125 regular latitude-longitude grid instead. For the medical data extraction, the datasets shown are high-resolution black-blood vessel wall MRI images collected in [51]. The exact data reduction statistics achieved for the examples mentioned in the previous section in Figs. 7, 8 and 9 are shown in Table 1.

In Table 1, the first 3 columns show the number of bytes retrieved when using different extraction techniques. In particular, note the clear distinction between the first two columns which differentiate the bounding box approach described earlier from the state of practice extraction methods taken in the fields of meteorology and healthcare respectively. We refer to state of practice approaches here as full dataset file extractions, which are even less optimal than the bounding box approach. Indeed, it is important to note that one of the widely used extraction approaches in the field of meteorology for example is to extract whole fields, which are 2D arrays of latitude and longitude around the whole globe, from datacubes. Similarly, in the field of healthcare, MRI scans are currently stored as 3D images, which are completely extracted before any clipping operation is performed. The bounding box approach is thus already a clear improvement compared to these approaches. As we see in Table 1 however, Polytope performs even better than the bounding box approach. This can be clearly observed in the fifth column, where we provide the reduction factor of the data retrieved when using the Polytope algorithm compared to the bounding box approach. The sixth column shows the total reduction of the data retrieved when using the Polytope algorithm compared to the state of practice extraction methods taken in the meteorology and healthcare fields. The final two columns then show the two slicing and total algorithm run times discussed above for each of our example shapes.

Along the rows, we differentiate between different types of shapes. On the first 3 rows, we first test the Polytope algorithm on shapes that are defined orthogonally along their axis and which could be directly extracted using the bounding box approach. For these 3 rows, as we see in the fifth column, using the Polytope algorithm instead of the bounding box approach does not reduce the size of the retrieved data further. Note however that running the Polytope algorithm in these three examples does not take significant time. In the latter 4 rows, we then experiment using the Polytope algorithm to retrieve more complicated non-orthogonal or axis-aligned shapes. Already for country shapes in 2D, we see that there is a significant data reduction when using the Polytope algorithm compared to the bounding box approach, with a reduction factor of up to 6 times in some cases. When considering higher dimensional shapes, and especially “path"-like shapes such as flight paths, we experience an even higher reduction factor. Indeed, in the 4D case of the flight path from Paris to New York mentioned above, about 350 times less data is returned to the users when using the Polytope algorithm instead of the bounding box approach. Again, note here that in most examples, the Polytope algorithm takes below a second to run whilst reducing the retrieved data size by a factor of at least 1000 compared to the state of practice approaches.

Importantly here, note that Polytope is able to perform the exact same orthogonal extractions as the bounding box approach in minimal time, whilst significantly outperforming the bounding box approach when extracting more complicated shapes. This suggests that the Polytope algorithm performs at least as well as the bounding box approach and thus makes it a strong competitor to this approach.

## Discussion

Note that in subsection 5.1 above, in Fig. 10a, we did not include the time spent extracting data from the datacube. This is because this data extraction time is very dependent on the storage medium on which the algorithm is run. We expect that the cost of performing the Polytope algorithm will be significantly less than the savings made by retrieving less data. In particular, we suspect that hardware that supports high performance random-read, such as flash based devices for example, will benefit massively from the Polytope algorithm. To take advantage of sequential data access, where possible, calls to storage are batched together into byte ranges so that the access pattern aligns better with the data storage layout. This optimisation step allows the Polytope algorithm to perform well for both single point extraction, as well as area extractions, without incurring significant additional costs compared to block extractions.

Compared to state of practice methods, the Polytope algorithm is an additional step in the extraction process which takes time to run. However, as we saw in this section, and in Fig. 10 especially, the Polytope algorithm is efficient and scalable, being able to locate more than a million points in less than half a second. Moreover, as already mentioned,

**Table 1** Polytope data reduction and performance

| Example Shape | Data retrieved with state of practice approach | Data retrieved with bounding box approach | Data retrieved with Polytope algorithm | Reduction factor compared to state of practice approach | Reduction factor compared to bounding box approach | Slicing time | Total algorithm run time |
|---|---|---|---|---|---|---|---|
| Box around germany | 50.4 MB | 44 KB | 44 KB | 1173 × | 1 × | 2.3e-3 s | 0.03 s |
| Timeseries of London over 14 days | 5.5 GB | 896 B | 896 B | 6591049 × | 1 × | 1.4e-4 s | 0.13 s |
| Vertical profile over 20 layers - Rome | 1 GB | 800 B | 800 B | 1342177 × | 1 × | 4.6e-5 s | 0.02 s |
| Country shape of france | 50.4 MB | 67.7 KB | 32.3 KB | 1598 × | 2 × | 0.03 s | 0.94 s |
| Country shape of norway | 50.4 MB | 171.4 KB | 29.9 KB | 1726 × | 6 × | 0.06 s | 1.97 s |
| Flight path from Paris to New York | 7.9 GB | 247.3 MB | 4.9 KB | 1690561 × | 51681 × | 0.07 s | 0.18 s |
| MRI blood vessel | 1 GB | 1.5 MB | 4.5 KB | 233017 × | 341 × | 0.10 s | 0.35 s |

Algorithm timings on an Apple M1 Pro chip (3.2 GHz, 8 cores) and 16GB DDR5. Note that for completeness, both slicing and total algorithm run time are included, although the slicing time is more indicative of the Polytope algorithm performance, as discussed in subsection 5.1

when discussing performance of the algorithm, it is especially important to also consider its wider role in the total extraction process. As we saw in Table 1, the Polytope algorithm allows users to extract much less data than they would have done using a more traditional approach. As reading and returning data is usually a costly operation, it implies that, when incorporated in a complete extraction pipeline, the Polytope algorithm will make data extraction more efficient than the current state of practice. The slightly more expensive slicing mechanism inside the Polytope algorithm will thus be outweighted by the actual performance improvement of the whole data extraction pipeline.

To quantify the true performance and benefits of the Polytope algorithm, we have performed a more in-depth analysis of its behaviour within a complete data extraction framework in [52]. In this work, we investigated the performance and scalability of the Polytope algorithm on the FDB meteorological data stores [53]. We found that for domain-specific feature shapes of interest, such as point timeseries, an end-to-end system using the Polytope algorithm extracts data about 1 to 2 orders of magnitude faster than traditional data extraction methods. In particular, the analysis shows that point timeseries across the whole forecast length are extracted in a fraction of a second, whereas extracting the equivalent full atmospheric fields from the FDB takes a few seconds. When scaled to higher dimensions, this gap only increases with a Polytope extraction for a point timeseries over all forecast ensemble members taking around 4 seconds, whilst accessing the corresponding full atmospheric fields from the FDB takes nearly 2 minutes. On top of faster extraction times, we also observe in the paper that Polytope reduces byte access to a small fraction of the full atmospheric fields stored on the FDB. For a point timeseries extraction for example, Polytope only reads a few hundred bytes, whereas the equivalent full fields accumulate to around a gigabyte of data. Compared to full FDB field extractions, we therefore conclude that using Polytope will likely lead to a significantly reduced resource usage of the overall system. The exact savings will depend on the access patterns that are requested by the users and on the system hardware specifications. A more detailed analysis is beyond the scope of this publication.

## Conclusion

In this paper, we introduced a new data extraction algorithm called Polytope, which has the capability of extracting arbitrary geometrical shapes from a datacube. This new technique allows users to directly compute the precise bytes of interest to them before requesting these bytes from a datacube. This approach leads to many benefits, both for the users and the data providers. For data providers, much less I/O is needed whereas for users, the need for further post-processing after extraction is alleviated. We described the structure of this novel extraction algorithm and explained in more detail some of its key features. We then showed a few use cases of the Polytope extraction technique before finally analysing the performance of this method. Future steps include performing a more in-depth analysis of the algorithm performance, as well as a rigorous discussion of Polytope's use cases in different scientific fields.

## Appendix A: Polytope algorithm example

Take a pyramid with vertices [[0, 0, 0], [0, 1, 0], [1, 1, 0], [1, 0, 0], [0.5, 0.5, 1]]. The datacube we slice against is a regular 3-dimensional point cloud with points at the intersections of the lines $x = (0, 0.25, 0.5, 0.75, 1)$, $y = (0, 0.25, 0.5, 0.75, 1)$ and $z = (0, 0.25, 0.5, 0.75, 1)$.

The first axis we slice along is the $z$-axis. The extents of the pyramid along the $z$-axis is (0, 1). Between $z = 0$ and $z = 1$, we have datacube points at $z = (0, 0.25, 0.5, 0.75, 1)$. Our algorithm thus creates index nodes $(z = 0), (z = 0.25), (z = 0.5), (z = 0.75)$ and $(z = 1)$ in the index tree and then slices the pyramid along each of those $z$ values.

The first resulting 2-dimensional square of the intersection between the pyramid and the plane $z = 0$ is the square in $x - y$ space with vertices [[0, 0], [0, 1], [1, 1], [1, 0]]. We store this reduced polytope in the corresponding tree node $(z = 0)$. We then slice along the $y$-axis. The extents of the reduced 2-dimensional square along the $y$-axis is (0, 1). Between $y = 0$ and $y = 1$, we have datacube points at $y = (0, 0.25, 0.5, 0.75, 1)$. Our algorithm thus creates sub-index nodes $(y = 0), (y = 0.25), (y = 0.5), (y = 0.75)$ and $(y = 1)$ as children of the node $(z = 0)$ and then slices the square along each of those $y$ values.

The first resulting line segment of the intersection between the square and the line $y = 0$ is the line segment in $x$ space with vertices [0, 1]. We store this reduced polytope in the corresponding tree node $(z = 0, y = 0)$. The final and last axis to slice is the $x$-axis here. The extents of the line segment along the $x$-axis is (0, 1). Between $x = 0$ and $x = 1$, we have datacube points at $x = (0, 0.25, 0.5, 0.75, 1)$. We thus create the leaf nodes of this sub-branch $(x = 0), (x = 0.25), (x = 0.5), (x = 0.75)$ and $(x = 1)$.

We proceed similarly for the other $y$ sub-index nodes $(y = 0.25), (y = 0.5), (y = 0.75)$ and $(y = 1)$, where we also have leaf nodes $(x = 0), (x = 0.25), (x = 0.5), (x = 0.75)$ and $(x = 1)$ for each of those.

This finally gives the complete first branch of the result tree as shown in Fig. 11.

The second resulting 2-dimensional square of the intersection between the pyramid and the plane $z = 0.25$ is the square in $x - y$ space with vertices [[0.125, 0.125], [0.125, 0 .875], [0.875, 0.875], [0.875, 0.125]]. We store this reduced polytope in the corresponding tree node $(z = 0.25)$. We then slice along the $y$-axis. The extents of the reduced 2-dimensional square along the $y$-axis is (0.125, 0.875). Between $y = 0.125$ and $y = 0.875$, we have datacube points at $y = (0.25, 0.5, 0.75)$. Our algorithm thus creates sub-index nodes $(y = 0.25), (y = 0.5)$ and $(y = 0.75)$ as children of the node $(z = 0.25)$ and then slices the square along each of those $y$ values. Slicing the square along each of these $y$ values, we create the leaf nodes of each of those sub-branches $(x = 0.25), (x = 0.5)$ and $(x = 0.75)$. We then get the second branch of the result tree as shown in Fig. 12.

The third resulting 2-dimensional square of the intersection between the pyramid and the plane $z = 0.5$ is the square in $x - y$ space with vertices [[0.25, 0.25], [0.25, 0.75], [0. 75, 0.75], [0.75, 0.25]]. We store this reduced polytope in the corresponding tree node $(z = 0.5)$. We then slice along the $y$-axis. The extents of the reduced 2-dimensional square along the $y$-axis is (0.25, 0.75). Between $y = 0.25$ and $y = 0.75$, we have datacube points at $y = (0.25, 0.5, 0.75)$. Our algorithm thus creates sub-index nodes $(y = 0.25), (y = 0.5)$ and $(y = 0.75)$ as children of the node $(z = 0.5)$ and then slices the square along each of those $y$ values. Slicing the square along each of these $y$ values, we create the leaf nodes of each of those sub-branches $(x = 0.25), (x = 0.5)$ and $(x = 0.75)$. We then get the third branch of the result tree as shown in Fig. 13.

The fourth resulting 2-dimensional square of the intersection between the pyramid and the plane $z = 0.75$ is the square in $x - y$ space with vertices [[0.375, 0.375], [0.375, 0.62 5], [0.625, 0.625], [0.625, 0.375]]. We store this reduced polytope in the corresponding tree node $(z = 0.75)$. We then slice along the $y$-axis. The extents of the reduced 2-dimen-

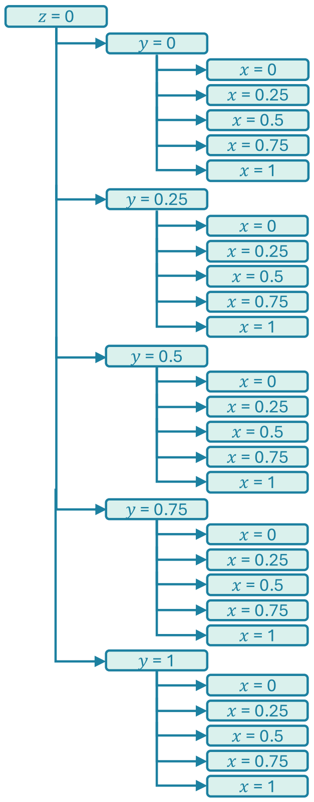

**Fig. 11** First tree branch

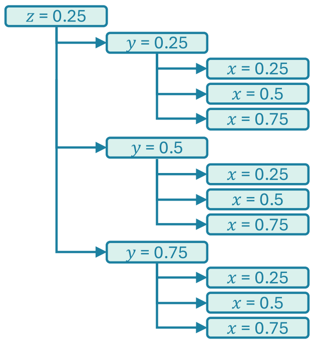

**Fig. 12** Second tree branch

sional square along the $y$-axis is (0.375, 0.625). Between $y = 0.375$ and $y = 0.625$, we have datacube points at $y = (0.5)$. Our algorithm thus creates the sub-index node $(y = 0.5)$ as a child of the node $(z = 0.75)$ and then slices the square along this $y$ value. Slicing the square along this $y$ value, we create the leaf node of this sub-branch $(x = 0.5)$. We then get the fourth branch of the result tree as shown in Fig. 14.

Finally, the fifth resulting 2-dimensional square of the intersection between the pyramid and the plane $z = 1$ is the point in $x - y$ space [0.5, 0.5]. We store this reduced polytope in the corresponding tree node $(z = 1)$. We then slice along the $y$-axis. The extents of the point along the $y$-axis is (0.5, 0.5). Between $y = 0.5$ and $y = 0.5$, we have datacube points at $y = (0.5)$. Our algorithm thus creates the sub-index node $(y = 0.5)$ as a child of the node $(z = 1)$ and then slices the square along this $y$ value. Slicing the square along

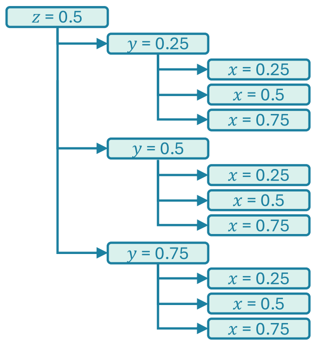

**Fig. 13** Third tree branch

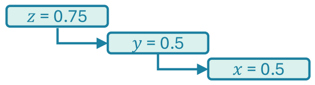

**Fig. 14** Fourth tree branch

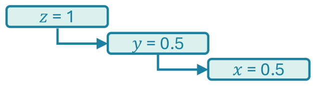

**Fig. 15** Fifth tree branch

this $y$ value, we create the leaf node of this sub-branch $(x = 0.5)$. We then get the fourth and final branch of the result tree as shown in Fig. 15.

At the end of the algorithm, we have now found all datacube points contained in the original pyramid, which are stored in our final result tree, as expected.

## Abbreviations

| | |
|---|---|
| ECMWF | European centre for medium range weather forecasts |
| ESEE | ECMWF software enginE |
| DestinE | Destination earth |
| DTE | Digital twin engine |
| OGC | Open geospatial consortium |
| EDR | Environmental data retrieval |
| NWP | Numerical weather prediction |
| MARS | Meteorological archival and retrieval system |
| MRI | Magnetic resonance imaging |

**Acknowledgements**
The authors would like to thank Matthijs de Buck and the Nuffield Department of Clinical Neurosciences at the University of Oxford for providing data for the MRI Blood Vessel Detection use case. The authors are also very grateful to ECMWF, the European Union's Destination Earth initiative and multiple European Commission projects and programmes which have helped fund this work.

## Author contributions

ML, JH, SS, ED and TQ contributed to the initial design and concept of the algorithm. ML and JH contributed to the implementation of the algorithm. ML conducted the performance and scalability analysis. ML wrote the main manuscript text. JH, TQ and MS reviewed the manuscript.

**Funding**
The work presented in this paper has been co-funded in the context of the European Union's Destination Earth Initiative and relates to tasks entrusted by the European Union to the European Centre for Medium-Range Weather Forecasts implementing part of this Initiative with funding by the European Union. Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

## Data availability

No datasets were generated or analysed during the current study.

## Declarations

**Competing interests**

The authors declare no competing interests.

Received: 21 December 2024 / Accepted: 8 October 2025
Published online: 01 November 2025

**References**

1. Yaqoob I, Hashem IAT, Gani A, Mokhtar S, Ahmed E, Anuar NB, et al. Big data: from beginning to future. Int J Inf Manage. 2016;36(6):1231–47.
2. Bauer P, Quintino T, Wedi N, Bonanni A, Chrust M, Deconinck W, Diamantakis M, Düben P, English S, Flemming J et al. The ECMWF Scalability Programme: Progress and Plans. ECMWF 2020. https://www.ecmwf.int/node/19380
3. Data: a small four-letter word which has grown exponentially to such a big value. https://www.deloitte.com/cy/en/Industries/technology/perspectives/data-grown-big-value.html. Accessed on 3 Oct 2024
4. Baumann P, Misev D, Merticariu V, Huu BP. Array databases: concepts, standards, implementations. J Big Data. 2021;8:1–61.
5. Baumann P, Misev D, Merticariu V, Huu, BP. Datacubes: Towards space/time analysis-ready data. Service-Oriented Mapping: Changing Paradigm in Map Production and Geoinformation Management, 2019;269–299
6. Killough B. Overview of the open data cube initiative. In: IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium, 2018;pp. 8629–8632 . IEEE
7. Baumann P. Datacube standards and their contribution to analysis-ready data. In: IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium, 2018;pp. 2051–2053 . IEEE
8. Wang Z, Chu Y, Tan K-L, Agrawal D, Abbadi AE, Xu X. Scalable data cube analysis over big data. arXiv preprint arXiv:1311.5663 2013.
9. Higham DJ, Higham NJ. MATLAB Guide. vol. 150. SIAM 2016.
10. Hoyer S, Hamman J. Xarray: ND labeled arrays and datasets in Python. J Open Res Softw. 2017;5(1):10.
11. Xtensor Stack: Xtensor Documentation. In *Xtensor* (Version 0.24.6). Retrieved from Read the Docs: https://xtensor.readthedocs.io/en/latest/. Accessed on 10 Feb 2023
12. Melton J, Simon AR. SQL: 1999: understanding relational language components. Elsevier 2001.
13. Snodgrass R. The temporal query language TQuel. ACM Trans Database Syst. 1987;12(2):247–98.
14. Obe RO, Hsu LS. PostgreSQL: up and running: a practical guide to the advanced open source database. O'Reilly Media: Inc; 2017.
15. Baumann P, Furtado P, Ritsch R, Widmann N. The RasDaMan approach to multidimensional database management. In: Proceedings of the 1997 ACM Symposium on Applied Computing, 1997;pp. 166–173
16. Fiore S, D'Anca A, Elia D, Palazzo C, Williams D, Foster I, Aloisio G. Ophidia: a full software stack for scientific data analytics. In: 2014 International Conference on High Performance Computing & Simulation (HPCS), 2014;pp. 343–350 . IEEE
17. Gosink L, Shalf J, Stockinger K, Wu K, Bethel W. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In: 18th International Conference on Scientific and Statistical Database Management (SSDBM'06), 2006;pp. 149–158 . IEEE
18. Taylor KE. Ezget: A library of fortran subroutines to facilitate data retrieval. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) 1996.
19. Gray J, Chaudhuri S, Bosworth A, Layman A, Reichart D, Venkatrao M, et al. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Min Knowl Discov. 1997;1:29–53.
20. Frihida A, Marceau DJ, Thériault M. Extracting and visualizing individual space-time paths: an integration of GIS and KDD in transport demand modeling. Cartogr Geogr Inf Sci. 2004;31(1):19–28.
21. Baumann P. Management of multidimensional discrete data. VLDB J. 1994;3(4):401–44.
22. Leuridan M, Warde A, Hawkes J, Varndell J, Tsrunchev P, Figala D. ecmwf/polytope: 1.0.21. https://doi.org/10.5281/zenodo.14537049 .
23. Geenen T, Wedi N, Milinski S, Hadade I, Reuter B, Smart S, et al. Digital twins, the journey of an operational weather system into the heart of Destination Earth. Procedia Computer Science. 2024;240:99–108.
24. Wolfe P. Finding the nearest point in a polytope. Math Program. 1976;11:128–49.
25. Thompson AC. Convex sets. In: Meyers, R.A. (ed.) Encyclopedia of Physical Science and Technology (Third Edition), Third edition edn., 2003;pp. 717–737. Academic Press, New York. https://doi.org/10.1016/B0-12-227410-5/00146-0. https://www.sciencedirect.com/science/article/pii/B0122274105001460
26. Chazelle B, Palios L. Decomposition algorithms in geometry. In: Algebraic Geometry and Its Applications: Collections of Papers from Shreeram S. Abhyankar's 60th Birthday Conference, 1994;pp. 419–447 . Springer
27. Owen SJ. A survey of unstructured mesh generation technology. IMR. 1998;239:267.
28. Guttman A. R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984;pp. 47–57
29. Obe R, Hsu L. PostGIS in action. Simon and Schuster 2021.
30. Stolte C, Tang D, Hanrahan P. Multiscale visualization using data cubes. IEEE Trans Vis Comput Graph. 2003;9(2):176–87.
31. Purss MB, Peterson PR, Strobl P, Dow C, Sabeur ZA, Gibb RG, et al. Datacubes: a discrete global grid systems perspective. Cartographica. 2019;54(1):63–71.
32. Harzheim E. Ordered sets. vol. 7. Springer Science & Business Media 2005.
33. Otoo EJ, Wang H, Nimako G. Multidimensional Sparse Array Storage for Data Analytics. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016;pp. 1520–1529 . IEEE
34. Makris A, Tserpes K, Spiliopoulos G, Anagnostopoulos D. Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data. In: EDBT/ICDT Workshops 2019.

35. Su Y, Agrawal G. Supporting user-defined subsetting and aggregation over parallel NetCDF datasets. In: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), 2012;pp. 212–219 . IEEE
36. Su Y, Agrawal G, Woodring J. Indexing and parallel query processing support for visualizing climate datasets. In: 2012 41st International Conference on Parallel Processing, 2012;pp. 249–258 . IEEE
37. Barber CB, Dobkin DP, Huhdanpaa H. The quickhull algorithm for convex hulls. ACM Trans Math Softw. 1996;22(4):469–83.
38. Burgoyne M, Blodgett D, Heazel C, Little C. OGC API-Environmental Data Retrieval Standard. Open Geospatial Consortium Inc., Wayland, MA, USA, OpenGIS® Implementation Specification OGC
39. Ricci A. A constructive geometry for computer graphics. Comput J. 1973;16(2):157–60.
40. Martin RR, Stephenson P. Sweeping of three-dimensional objects. Computer-Aided Design. 1990;22(4):223–34.
41. ECMWF: Key facts and figures. https://www.ecmwf.int/en/about/media-centre/key-facts-and-figures. Accessed on 29 Sep 2024
42. ECMWF: MARS Catalogue. https://apps.ecmwf.int/mars-catalogue/. Accessed on 29 Sep 2024
43. Wedi N, Quintino T, Modigliani U, Baousis V, Geenen T, Sandu I, et al. Destination Earth: Digital Twins of the Earth System. Copernicus Meetings: Technical report; 2022.
44. Wedi N, Bauer P, Sandu I, Hoffmann J, Sheridan S, Cereceda R, et al. Destination Earth: High-Performance Computing for Weather and Climate. Computing in Science & Engineering. 2022;24(6):29–37.
45. Raoult B. Architecture of the new MARS server. https://www.ecmwf.int/sites/default/files/elibrary/1997/11839-architecture-new-mars-server.pdf. Accessed on 11 Feb 2023
46. Guo Y, Punnasiri K, Tong S. Effects of temperature on mortality in Chiang Mai city, Thailand: a time series study. Environ Health. 2012;11:1–9.
47. Chen R, Yin P, Wang L, Liu C, Niu Y, Wang W, Jiang Y, Liu Y, Liu J, Qi J et al. Association between ambient temperature and mortality risk and burden: time series study in 272 main Chinese cities. BMJ 2018;**363**
48. Ordonez C. Data set preprocessing and transformation in a database system. Intell Data Anal. 2011;15(4):613–31.
49. 4 ways data is improving healthcare. https://www.weforum.org/agenda/2019/12/four-ways-data-is-improving-healthcare. Accessed on 24 Apr 2023
50. Viessmann O, Li L, Benjamin P, Jezzard P. T2-weighted intracranial vessel wall imaging at 7 tesla using a DANTE-prepared variable flip angle turbo spin echo readout (DANTE-SPACE). Magn Reson Med. 2017;77(2):655–63.
51. Buck MH, Hess AT, Jezzard P. Simulation-based optimization and experimental comparison of intracranial T2-weighted DANTE-SPACE vessel wall imaging at 3T and 7T. Magn Reson Med. 2024;92(5):2112–26.
52. Leuridan M, Bradley C, Hawkes J, Quintino T, Schultz M. Performance analysis of an efficient algorithm for feature extraction from large scale meteorological data stores. In: Proceedings of the Platform for Advanced Scientific Computing Conference, 2025;pp. 1–9
53. Smart SD, Quintino T, Raoult B. A high-performance distributed object-store for exascale numerical weather prediction and climate. In: Proceedings of the Platform for Advanced Scientific Computing Conference, 2019;pp. 1–11

## Publisher's Note