RADIG: A Resolution-Adaptive Grid Reference Framework for Persistent Spatial Index

Sheng Zhou

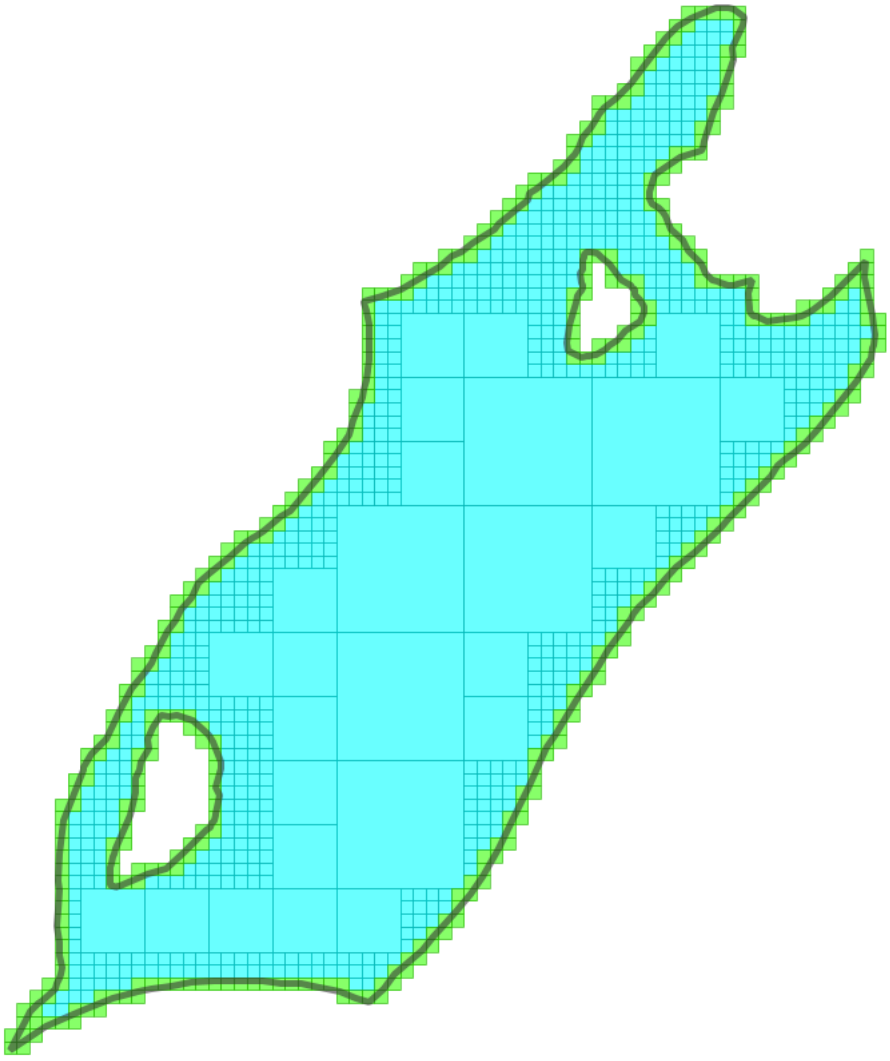
Data Science and Analytics

Chief Geospatial Office

Ordnance Survey

Ver.1.5

05/01/2025 13:15



1 Introduction

Significant progresses have been made in handling geospatial data in large-scale distributed/parallel environment such as Azure Databricks. Recently Apache Sedona (formerly GeoSpark) introduced SpatialRDD with customised data partitioner for spatial data and proper spatial indexing capability.

Although SpatialRDD facilitates efficient spatial queries and spatial joins, and the spatial index can be made persistent for re-use (development is ongoing for persistent spatial partition), it lacks the interoperability to enable indexed data being easily processed in other platforms, especially platforms without appropriate spatial data handling capabilities.

At Ordnance Survey, a method has been developed to use BNG (British Nation Grid) reference to support spatial queries. BNG is a series of regular grids at pre-defined resolutions with each grid cell assigned with a unique reference. To index a geometry, the grid references of all grid cells at a given resolution are assigned to the geometry as an index key set. Subsequently, whether two geometries are potentially related spatially (intersects, contains, etc) may be determined by examining whether they share any grid references (at the same resolution). When two geometries share grid references, further spatial relation test may be performed to refine the result and determine the exact spatial relation between the two geometries. This method has been integrated into the new Databricks Mosaic spatial extension to Spark as a new index system.

This BNG reference based method provides a solution to persistent index. However, it has several issues to be addressed. The fundamental problem is that the spatial resolution used to generate BNG reference for a dataset is fixed. It is often tricky to decide what resolution we should use. For larger objects, a fine grid resolution will result in too many references assigned to the object, which increases storage requirement and affects query performance as well. For smaller objects, a large grid resolution will result in references representing larger grid cells assigned to the object so that it will be retrieved by too queries it does not relate to.

With BNG reference, two datasets to be compared normally have to use grids of the same resolution, and it is meaningless to compare two references at different resolutions directly. Therefore, multiple grid reference sets may have to be built for different resolutions. For dataset with huge variation in geometric size of features, we will still have to use the same resolution for all features in the dataset. All these issues introduce extra storage and performance overheads into the scheme.

In this report, we present a new grid-based persistent-able spatial indexing framework which address all the above issues. It supports grid reference at variable resolution inside the same dataset, and even for the same geometry. It also enables efficient spatial joins and range queries on non-spatial database platform with standard SQL support. Furthermore, we propose a new style BNG reference system which offers more powerful functionality than the existing BNG references which is for simple identification at a fixed resolution.

2 Digit interleaving, Digit concatenation and Adaptive Grid Reference

Before the introduction of our indexing scheme, we will first give a brief review on some concepts related to spatial index. Also, for clarity we will restrict our discussion to two-dimension data although most of our results also apply to data at higher dimensions.

2.1 Partition and Spatial Index

One of the main purposes of building spatial index (and in fact any data index) is to reduce query time from linear time to logarithmic or even constant time. Basically indexing is to partition value space into sub-spaces containing data, and sub-spaces containing no data. There are two categories of partition strategy: object-based partition and space-based partition.

Object-based partition creates the partition according to the extent of data objects. Point quadtree, B-tree and its spatial extension R-tree fall into this category. On the other hand, space-based partition divides space into pre-defined sub-space and subsequently use data objects to determine whether a sub-space is with data or without data. Regular grids and region quadtrees are examples of space-based partition scheme.

An obvious advantage of object-based partition is that it is normally more efficient in terms of space utilisation, i.e. the ratio of the area of data objects over the area of sub-space labelled as data space covering these objects. However, it is for exactly the same reason that object-based partition is poorly interoperable. Indeed, although Spark Sedona offers R-tree index for spatial data in a partition, the customised partition scheme is based on regional quadtree in order to form spatially conformed partitions over different datasets.

2.2 Dimension Reduction, Digit Inter-Leaving and Digit Concatenation

For practical reasons such as physical storage access requirements, it is often desirable to map multi-dimensional data into one dimension. Where some dimension reduction methods (e.g. Hilbert code) involve complex computation, some others (e.g. Morton code) may be implemented via two simple techniques: bit interleaving and bit concatenation. Since the term "bit" normally refers to binary presentation, we will use **digit interleaving** and digit concatenation instead, which are more generic.

Give a 2D point (X, Y) with each ordinate value represented by *m* digits in the form of ($x_m x_{m-1} \ldots x_1$, $y_m y_{m-1} \ldots y_1$), the two ordinates may be coded into a single string of digits in two ways:

- Digit interleaving: $x_m y_m x_{m-1} y_{m-1} x_1 y_1$
- Digit concatenation: $x_m x_{m-1} \ldots x_1 y_m x_{m-1} \ldots y_1$

For example, for X = 1234 and Y = 5678, the digit interleaving form is: 15263748; the digit concatenation form is: 12345678.

If the one-dimensional digit strings are used as identifiers only, there is not any significant difference between the two forms apart from that the concatenation form is more readable. However, they are fundamentally different on all other aspects.

2.3 Hierarchical space partition and digit interleaving:

Given a spatial extent and a coordinate system to represent locations in the space, each digit place in an ordinate value may be viewed as a partition to the extent in the corresponding dimension. Consequently, a hierarchical partition is formed going from the uppermost digit to the lowermost digit. For the example of a 2D space extent of X = [0, 10000000) and Y = [0, 10000000) in the radix (base) 10 number system and assuming 3 decimal places, a location will be represented as ($x_6 x_5 x_4 x_3 x_2 x_1 x_0 . x_{-1} x_{-2} x_{-3}$, $y_6 y_5 y_4 y_3 y_2 y_1 y_0 . y_{-1} y_{-2} y_{-3}$) with 7 integral digits. For the pair of uppermost digits (x_6, y_6), they form a 10x10 grid partition on the whole 10000km extent with cell size of 1000km by 1000km. Each cell is

represented by a pair of digit values. For example, (2, 3) represents the 1000km by 1000km cell with corners (2000000 3000000) and (3000000 4000000). Subsequently, digits (x_5, y_5) form a further 10x10 partition on each 1000km by 1000km cell, and so on. For example, (24, 35) for the 100km by 100km cell with corners(2400000 3500000) and (2500000, 3600000). To pack a digit pair into a one-dimension string via digit interleaving, we have 23 for the 10^3km by 10^3km cell (2, 3) and 2345 for the 100km by 100km cell (24, 35).

A complete list of the interleaved code for our sample space partitioning scheme above is listed in the table below:

Table 1: Hierarchy by digit partition

Interleaved code	Order of Digits in ordinate	Cell size/resolution	Upper-Cell partitioned by the digit
x_6y_6	1	10^3km x 10^3km	10^4km x 10^4km (Extent)
$x_6y_6x_5y_5$	2	100km x 100km	10^3km x 10^3km
$x_6y_6x_5y_5x_4y_4$	3	10km x 10km	100km x 100km
$x_6y_6x_5y_5x_4y_4x_3y_3$	4	1km x 1km	10km x 10km
$x_6y_6x_5y_5x_4y_4x_3y_3x_2y_2$	5	100m x 100m	1km x 1km
$x_6y_6x_5y_5x_4y_4x_3y_3x_2y_2x_1y_1$	6	10m x 10m	100m x 100m
$x_6y_6x_5y_5x_4y_4x_3y_3x_2y_2x_1y_1x_0y_0$	7	1m x 1m	10m x 10m
$x_6y_6x_5y_5x_4y_4x_3y_3x_2y_2x_1y_1x_0y_0x_{-1}y_{-1}$	7+1	0.1m x 0.1m	1m x 1m
$x_6y_6x_5y_5x_4y_4x_3y_3x_2y_2x_1y_1x_0y_0x_{-1}y_{-1}x_{-2}y_{-2}$	7+2	0.01m x 0.01m	0.1m x 0.1m
$x_6y_6x_5y_5x_4y_4x_3y_3x_2y_2x_1y_1x_0y_0x_{-1}y_{-1}x_{-2}y_{-2}x_{-3}y_{-3}$	7+3	0.001m x 0.001m	0.01m x 0.01m

2.4 Spatial predicts with digit interleaving strings:

An important property of such digit interleaving codes is that the interleaved code of a cell is a **prefix** of the codes of any of its sub-cells at any level of the hierarchy down from the cell. In the example above, "23" is a substring of "2345". In comparison, a code by digit concatenation does not possess this property (e.g. 23 vs 2435).

In a more general form, a grid cell represented by interleaved code $x_my_mx_{m-1}y_{m-1}\ldots x_iy_i$ spatially contains any grid cells represented by $x_my_mx_{m-1}y_{m-1}\ldots x_iy_i$**. Here ** is a wildcard symbol which denotes to any number of extra digit pairs.

It is obvious this property of digit interleaving code may be used to perform spatial intersection and containment test by string operation and without the need of spatial functions (albeit proper spatial operators are required for refining results). Crucially, the two strings do not need to have same length, or in other words, do not need to be encoded at the same resolution.

Most programming languages provides methods to test whether a string starts with another string, for example, `startswith(expr, startExpr)` in Databricks SQL, `starts_with(expr, startExpr)` ins PostgresSQL 11, or Java String class methods `startsWith(String str)` and `startsWith(String str, int offset)`.

To test if a cell represented by digit interleaving string str1 contains a cell represented by str2, we may use (in Java/Scala):

```
str2.startsWith(str1)
```

In other words, if `str2` starts with `str1`, then cell-str1 contains cell-str2.

To test if the two cells are intersecting strictly (i.e., with common interior points), we may use (in Java/Scala):

```
str1.startsWith(str2) || str2.startsWith(str1)
```

In database systems, such operations will normally be converted to a range join query which may be optimised significantly under certain conditions.

Potentially, techniques such as regular expression may be used to explore the possibility of performing other spatial operations (such as proximity) using interleaved code string.

Is it possible to perform the same tasks using digit concatenated string? The answer is yes but at extra costs. Firstly, the two ordinate components have to be extracted from the string, then the two parts have to be compared separately. For example (in Java):

```
int len1 = str1.length()

String str1X = str1.substring(0, len1 / 2)

String str1Y = str1.substring(len1 / 2)

int len2 = str2.length()

String str2X = str2.substring(0, len2 / 2)

String str2Y = str2.substring(len2 / 2)
```

To test containment:

```
Str2X.startsWith(str1X) && str2Y.startsWith(str1Y)
```

To test intersection:

```
str1X.startsWith(str2X) && str1Y.startsWith(str2Y) ||
str2X.startsWith(str1X) && str2Y.startsWith(str1Y)
```

Compared to the same operation in digit interleaving form, this is clearly more complicated.

3 RADIG – Resolution Adaptive Digit Interleaving Grid for persistent spatial index

In this section, we will present RADIG, a framework for generating digit interleaving grid references for spatial objects at varying resolutions.

3.1 Number base and resolution granularity:

Given a coordinate (X, Y) in radix 10 number system with *m* integral digits and *p* decimal digits, we may use the digits directly for interleaving encoding, or we may convert the integral and decimal parts to number systems of other bases (e.g. binary, octal, or hexadecimal). To represent the same spatial extent, different number of digits will be required under different bases. Also, different bases correspond to different number of sub-partitions to a grid cell and result in different resolution granularity.

Table 2 Digits in ordinate string $x_3x_2x_1x_0$ and the value range they represent

Base	x_0	x_1	x_2	x_3
10	(0-9)x1	(0-9)x10	(0-9)x100	(0-9)x1000
2	(0-1)x1	(0-1)x2	(0-1)x4	(0-1)x8
8	(0-7)x1	(0-7)x8	(0-7)x64	(0-7)x512
16	(0-15)x1	(0-15)x16	(0-15)x256	(0-15)x4096

For base 10, the ratio between two consecutive resolutions is 10:1 and it is 16:1 for hexadecimal system. For most applications this ratio is too large and will lead to very low space utilisation. Our solution to this issue is to introduce a dual divisor system, that is, for one digit, we partition the cell twice. For example, for base 10, we first partition a cell by 2 into 4 sub-cells, subsequently we partition each sub-cell 5 times. This way we convert the 10:1 resolution ratio to 2:1 and 5:1, which provides finer resolution granularity. Of course, this method will double the length of encoded string.

The dual divisors must satisfy the following condition:

```
base = first_divisor x second_divisor
```

Therefore, for base 10, we have choices of 2x5 or 5x2; for base 16, we have 4x4, 2x8 or 8x2. Of course, for binary system, second divisor is not required, and it provides the finest resolution granularity.

3.2 Encoding convention for grid index reference and grid descriptor

A single divisor digit-interleaved coordinate string of a coordinate (X, Y) with *m* integral digits and *p* decimal digits has the following form:

$$Gd_1d_2d_3d_4x_my_mx_{m-1}y_{m-1}...x_1y_1x_{-1}y_{-1}...x_{-p}y_{-p}$$

- G: either X or Y. X indicate x ordinate is given priority in the encoding process and Y indicate y ordinate has the priority
- d_1: base – 1 (permitted values: 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
- d_2d_3: number of integer digits (in base d_1) used in encoded coordinate string. Permitted value: "01" to "99". We may use a single printable ascii (32-126) for this purpose to save one digit, but it is less readable than a plain string of numbers.
- d_4: first divisor – 1 (permitted values: 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). If d_4 equals to base, only a single divisor (the base) is used in this scheme instance.

Based on these conventions, we may describe a grid reference system by a **grid descriptor** which is simple tuple of **(G, B, I, D)**. Here G is either X or S, B the base (minus one) corresponds to d_1, I the number of integral digits to d_2d_3 and D the first divisor (minus one) to d_4 above respectively.

For example, if we are to encode a coordinate in base-10, maximum 7 integral digits, first divisor of 2 and priority for x ordinate, we will have the encoding descriptor of (X, 9, 7, 1) and prefix of: "X9071". It is (X, 9, 7, 9) and "X9079" if we use one divisor only.

The number of integral digits and the total number of digits required for the above base-10 coordinate are *m* and *m+n* respectively. For number system of other bases B, the values m_B and p_B may be calculated from the base-10 coordinate value range [0, C_{max}) and the number of decimal places in the values *p*.

$$Pow(B, m_B) \geq C_{max} \Rightarrow m_B \geq lg(C_{max}) / lg(B)$$

$$Pow(B, -p_B) \leq Pow(10, -p) \Rightarrow p_B \geq p / lg(B)$$

For example, in BNG, the maximum coordinate value is 1300000, which requires 7 digits in radix 10 number system, we have: $m_{10} \geq lg(130000)/lg(10) \Rightarrow m_{10} \geq 6.1139... \Rightarrow m_{10} = 7$. To cover this range in binary, we have: $m_2 \geq lg(130000)/lg(2) \Rightarrow m_2 \geq 20.31... \Rightarrow m_2 = 21$. If we have 3 decimal places in base-10 value, we have $p_{10} \geq 3 / lg(10) \Rightarrow p_{10} = 3$. In binary, we have: $p_2 \geq 3 / lg(2) \Rightarrow p_2 \geq 9.96578... \Rightarrow p_2 = 10$. That is, if we convert a BNG ordinate value with 7.3 base-10 digits, we require 21.10 base-2 digits.

3.3 Encoding with dual divisors

In previous section we introduced the concept of dual divisors for improved resolution granularity. Here we provide an example how this system works. In the previous example of ($x_6x_5x_4x_3x_2x_1x_0.x_{-1}x_{-2}x_{-3}$, $y_6y_5y_4y_3y_2y_1y_0.y_{-1}y_{-2}y_{-3}$), interleaving the uppermost digits x_6 and y_6 will generate string "x_6y_6" under a single-divisor (10) scheme. Now with a dual divisor scheme ($d_1 = 2$ and $d_2 = 5$), the digits will be encoded in the following manner:

$$x_{6\_1}y_{6\_1}x_{6\_2}y_{6\_2}$$

Where: $x_{6\_1} = [x_6 / d_2]$, $y_{6\_1} = [y_6 / d_2]$, $x_{6\_2} = x_6 \% d_2$, $y_{6\_2} = y_6 \% d_2$

Here […] refers to the integral part of a number and % is the modular operation. For example, given coordinate (1234567, 7654321), the first digits 1,7 will be encoded as "17" in a single divisor scheme and will be encoded into "0112" in the above dual-divisor scheme:

Int (1/5) = **0**; int(7/5) = **1**; 1%5 = **1**; 7%5 = **2**

An alternative method to encode with dual divisors is not to use the modular operation but use the digit directly. In this case, the above example becomes "0117". To interpret a string encoded this way, the string may be divided into 4-letter groups from left to right. The first two letters in a group can be ignored unless it is the final group with 2 letters only. For example, we now encode the above example for 2 integral digits(12, 76) under dual divisors 2 and 5, we have "01120121" from the first encoding method and "01170126" from the second method. If we stop at the first divisor of the second digit, we have "011201" from the first method, and "011701" from the second method (from which we may ignore the leading "01" and still retrieve the grid cell extent ((1000000, 7500000), (1500000, 8000000)). At present we use the first encoding method with modular operation, but the second method is equally valid. Furthermore, the second may be modified to include a flag to indicate the encoding ends at first divisor so that shorter references will be generated. This flag, however, may prevent us from performing string-based spatial queries as described in previous section. To mention in passing, it appears this is the method Mosaic used to support the dual-divisions, i.e. use two digits in the encoded string for each digit in ordinate while a singular digit is added to the end to indicate quadrant if dual division is applied. For example, NG8116916OSE is represented internally in Mosaic as 10108811691604, and 10108811591602 for NG8115916ONW.

It should be noted that for any ordinate value with less than maximum number of integral digits, leading "0"s should be padded to it prior to encoding. For example, BNG eastings only have 6 integral digits (e.g. "123456") and should always be padded with a leading 0 ("0123456") for encoding.

3.4 Resolution-Adaptive Digit-Interleaving Grid (RADIG) reference generation

So far our discussion have been based on fixed resolution. In this section we will describe how a set of grid references at different resolutions may be generated for a single geometry. We will first discuss the concept of space utilisation in greater details and introduce two new concepts: **effective area** and **effective length**.

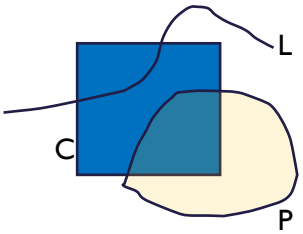


Figure 1: a grid cell C with an intersecting linestring L and polygon P

In figure 1, the grid cell C intersects with both linestring L and polygon P so the grid reference of C will be assigned to both L and P for indexing. It is easy to understand, if the area of intersection between C & P is larger, it is more likely P relates to a query that matches the grid reference. Consequently we define the effective area (**EA**) as the ratio of the area of intersection(C, P) and the area of C. Similarly, we define the effective length (**EL**) as the ratio of the length of intersection(C, L) and the length of the edge of C (i.e. resolution at current grid level).

Given a grid definition (G, B, I, D), we are able to generate a resolution range RR (as shown in table 1). For a geometry with maximum extent E (the larger of the width and height of its minimum bounding box), we find the resolution r in RR where r≥ E and use it as the initial grid size for indexing the geometry.

In case of strict-intersects (sharing interior points) at least one and at most 4 grid references will be returned; in case of generic-intersects (sharing boundary and/or interior points), at least one (when r > E) or four (r == E) references and at most 9 references are returned.

For each of the returned grid, we will calculate EA (or EL for linestring or polygon boundary). If EA is equal or above a predefined threshold, we return the grid reference for indexing; if EA is below the threshold, we will subdivide the cell into sub-cells using the divisor corresponding to the current cell size/resolution. For each sub-cell, we will repeat the comparison, discard those sub-cell not intersecting with the geometry and further test EA/EL for others until either the threshold is reach, or a predefined maximum number of subdivision levels, and/or, minimum cell size/resolution is reached.

The result of the above process is a set of grid references at different resolutions. This concludes our RADIG indexing framework.

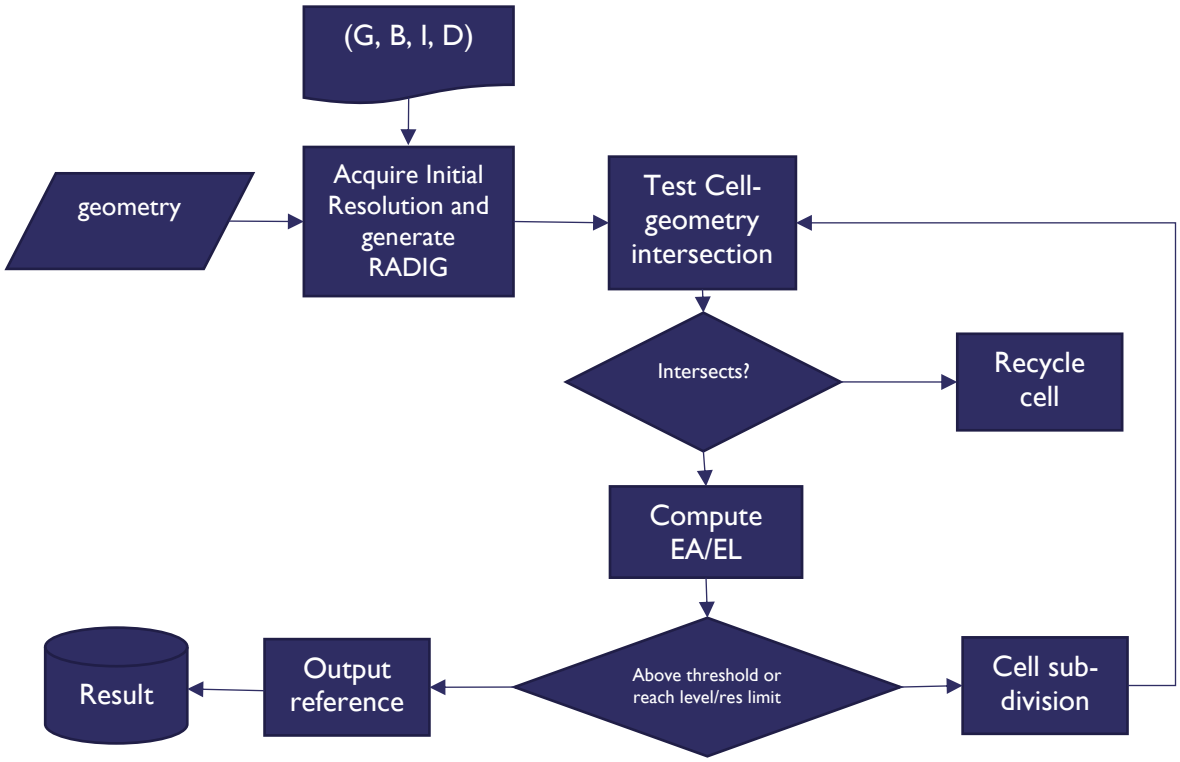


Figure 2: RADIG process flow chart

Note that potentially there are more appropriate definition of EA and EL based on statistical analysis and practical experiments. Also, additional controls may be added to decide the initial resolution.

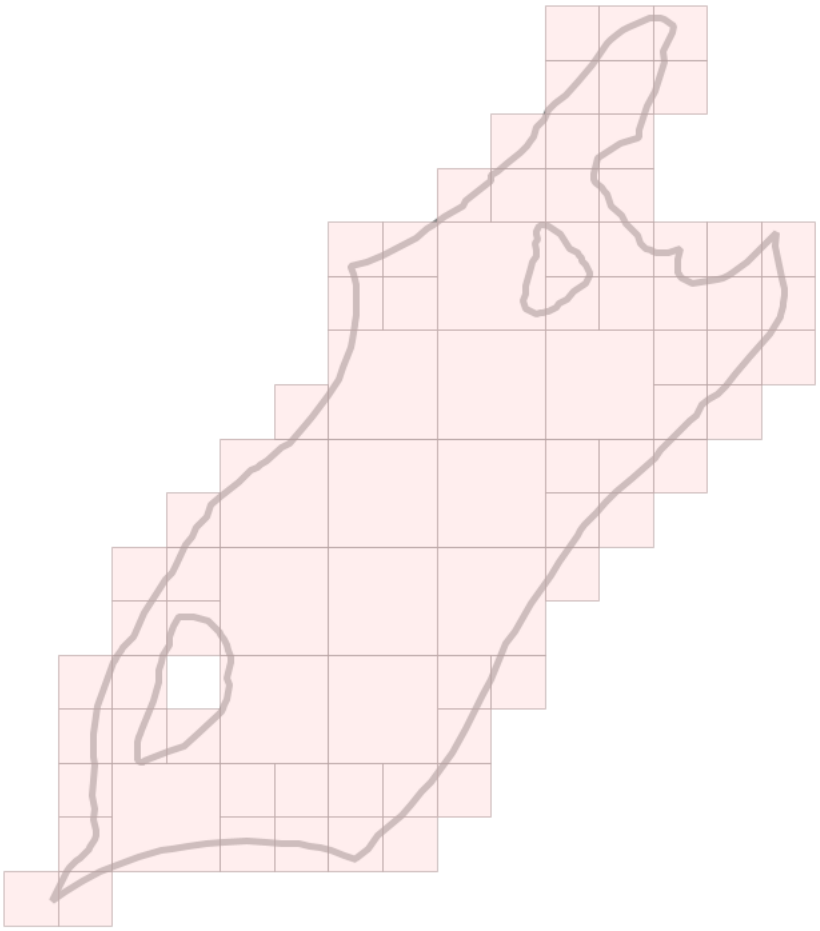


Figure 3: Sample RADIG output (Base-10, 2x5 divisors, max sub-div-level = 4, EA=0.8)

Compared to fix-resolution reference grids, RADIG with the same resolution value as its minimum resolution and no subdivision level control will always return the same, or in most cases, smaller reference set with the same degree of space utilisation. For the example in figure 3 where boundary intersecting and contained cells are separated, RADIG returns 78 cell references at two different resolutions, where a fix-resolution grid will return 117 references.

With a smaller minimum resolution, RADIG will be able to achieve a higher degree of space utilisation with the same amount or fewer grid references.

3.5 Re-Organised BNG as a special case of RADIG

BNG is an interesting reference system. For those familiar with applications of bit-interleaving in image process and geometry data handling, a BNG reference string looks like a specifically mapped prefix plus an interleaved string. In fact, it is a mix of mapped prefix plus digit concatenation. It is effectively a grid system

with 7 integral digits in ordinate values, with the first two digits from both northing and easting mapped to a two-letter prefix. The top grid extent is 10^7m and divisor at this level is 100, resulting in the familiar top prefix-represented 100km by 100km grid cells. From the 100km cell down, it is a digit-concatenation scheme with a single divisor of 10, and occasionally with a dual divisor 2x5 to generate grid reference with quadrant suffix (e.g. TQ35SW).

It is obviously BNG was initially designed as an identifier system for human interpretation to facilitate rapid location reference on paper map. It serves this purpose well and is also sufficient as and space identifier. However, had the BNG been designed in a different, digit-interleaving way, it could have far great extent of applications. Holding this view, we are here proposing a re-organised BNG reference system based on RADIG.

Using our grid descriptor, the BNG grid may be defined as (X, 9, 7, 9) for single divisor, or (X, 9, 7, 1) for a 2x5 dual divisors. We will then use RADIG APIs to generate a conventional RADIG reference, remove the RADIG prefix, convert the first 4 letters (8 for dual divisors) of the remaining string to the original two-letter BNG tile reference. Finally we re-encode the reference as the follows for single divisor:

$$X9079x_m y_m x_{m-1} y_{m-1} x_{m-2} y_{m-2}\ldots \Rightarrow$$

$$STTx_{m-2} y_{m-2}\ldots$$

And for dual divisors:

$$X9071x_{m\_1} y_{m\_1} x_{m\_2} y_{m\_2} x_{m-1\_1} y_{m-1\_1} x_{m-1\_2} y_{m-1\_2} x_{m-2\_1} y_{m-2\_1} x_{m-2\_2} y_{m-2\_2}\ldots \Rightarrow$$

$$DTTx_{m-2\_1} y_{m-2\_1} x_{m-2\_2} y_{m-2\_2}\ldots$$

Here **S** prefix stands for "single" and **D** for "dual" to indicate the reference is from a single or dual divisor. **TT** are the two-letter BNG top tile references. If we use only the single divisor, or only dual divisors, we may omit the S/D prefix. However, it will then be impossible to distinguish between RADIG-derived references and original BNG references.

4 RADIG Reference Matching and Performance Consideration

In previous section we mentioned that RADIG reference matching is a substring operation. While string comparison is commonly optimised in all DBMS, substring operation such as startsWith() is not necessarily the case. In this section we will discuss briefly how such optimisation may be achieved.

4.1 RADIG Match Trie for reference matching

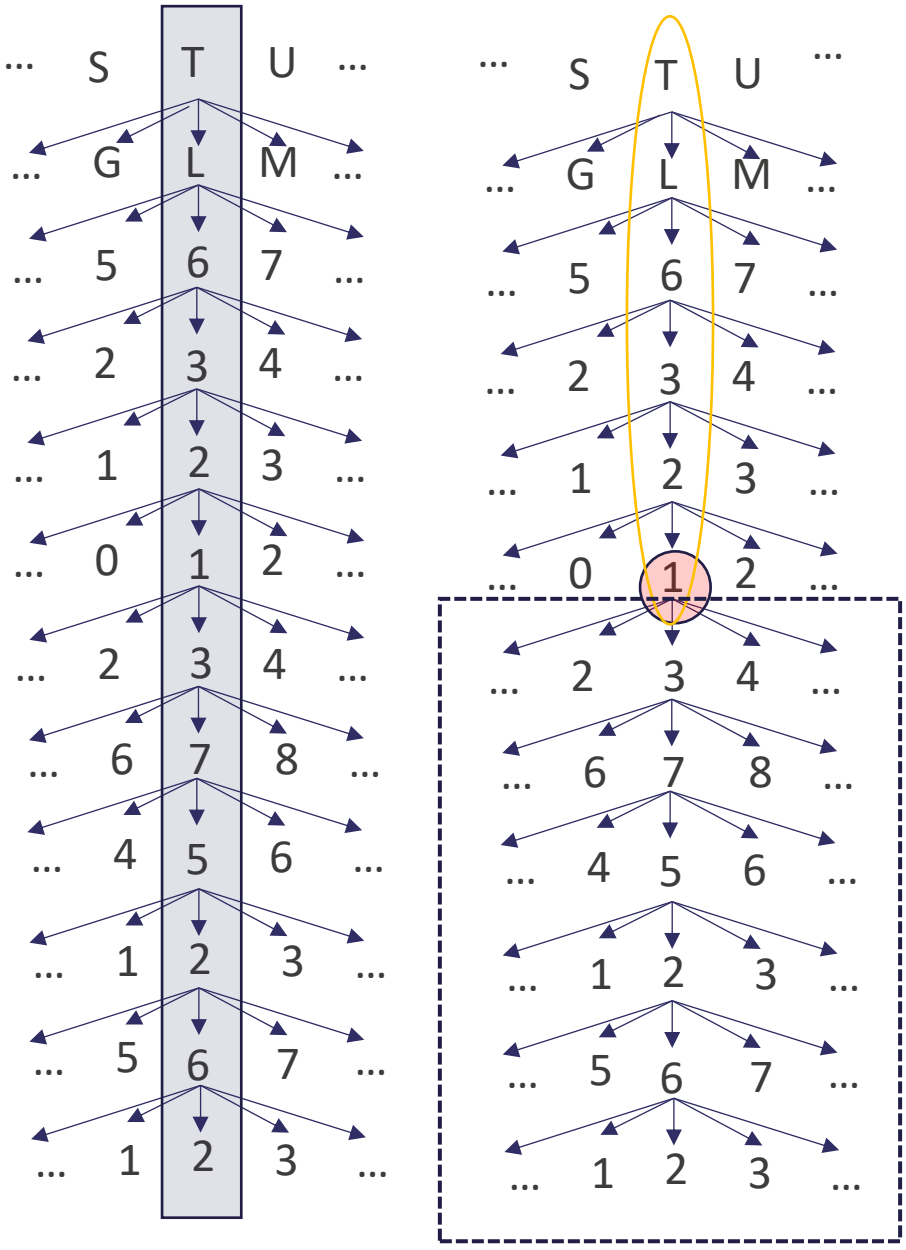


Figure 4: A Trie (prefix tree) and RADIG Reference matching

The natural data structure candidate for our prefix substring search requirement is Trie (a.k.a Prefix Tree). In figure 4 (left) the RADIG reference TL6321375262 is used as an example of how a reference string is represented in a Trie data structure. When a query to find strings starting with "TL6321" is executed (Figure 4 right), the search will drill down digit by digit until the matching node (red circle) is found. References in this node and all its children is the result set of the query.

To match two tries more efficiently, we have developed an algorithm to traverse two tries in a synchronised manner. When the traverse is completed, for all references in one trie the matching references in the other trie will be found.

4.2 Prefix matching in a distributed environment:

For distributed systems such as Databricks, it is more complicated to make the data structure perform well. The main issue is to find a suitable partition scheme to serve the purpose of prefix matching. The two datasets have to be partitioned under the same scheme and for each pair of corresponding partitions from the two datasets, RADIG Match Tries can then be built for matching.

Here we propose two approaches to address the partition issue.

4.2.1 Partition Key

The first approach is to add a second partition key for each feature in both datasets (e.g. a fixed 1km resolution RADIG reference). Subsequently, two datasets may be grouped by the partition key into units that may be matched by RADIG Match Trie. This method has been implemented with good results.

4.2.2 RADIG Partition Trie

The partition key approach will create partitions with large variation in the amount of features in each partition on dataset with non-uniform distribution. To address this issue, we propose data partition by RADIG Partition Trie.

Given the number of partitions to be generated and the number of records of the dataset to be partitioned, we may compute the preferred partition size. Subsequently, we use a Trie data structure similar to the RADIG Match Trie for prefix matching to count the number of references keys on each node. This trie has a predefined depth. Each Radig reference string is passed through the trie and the counter on each node a reference passes will be increased by one. A depth-first traversal will then be performed to determine the split points for partitions. Each node in the trie will be assigned with one or more partition IDs.

The crucial operation here is to trace back from split nodes to assign multiple partition IDs to the parent nodes.

After the partition trie is created, both object and query set references may be passed through the trie to acquire partition ID(s). The two datasets can then be partitioned and references in each pair of partitions may be compared with a trie built for each pair of partitions.

The partition trie for a dataset may be made persistent to avoid re-computation each time the dataset is used. Nevertheless, the other dataset to be joined has to be partitioned online and subsequent grouping operation will add extra overhead. Consequently, the overall performance may not necessarily be improved.

4.3 Prefix matching using RADIG reference substrings:

In above subsections the reference matching is carried out by the use of match trie. An alternative solution without any ad hoc partition schemes is to use reference substringing.

Given a string A (e.g. TL6321375262), we wish to test if it starts with string B (e.g. TL6321). Instead of using the often-not-optimised StartsWith function, we break string A into a set of substrings {TL, TL63,

TL6321, TL632137, TL63213752, TL6321375262}. Subsequently we check if any of string is in this set equals to string B.

In practice, we may pre-compute the substrings and store them along with the data and apply optimisation (Z Ordering) to the storage. This is a strategy similar to what Databricks Mosaic is using.

This method has also been implemented but the performance is not as good as the RADIG Match Trie approach.

5 Implementation and experiment

5.1 RADIG in Java/Scala

The RADIG framework described above has been implemented in Java. A set of BNG specific methods (Grid Descriptor (D, 10, 7, 2)) have been provided with Scala UDF interfaces for Databricks Spark applications.

5.2 Experiment results and evaluation

Below are examples under different base, levels of division and EA values. In particular, figure 5 shows the result with “intersects” and “contains” grid cells separated. Figures 7 and 8 are the grids with base 2 and 16 (4x4 divisors) respectively. In our opinion, base-2 grid offers the best resolution granularity (which will in turn result in better space utilisation). Therefore, it should be the scheme to use if alignment to existing grid systems such as BNG is not required.

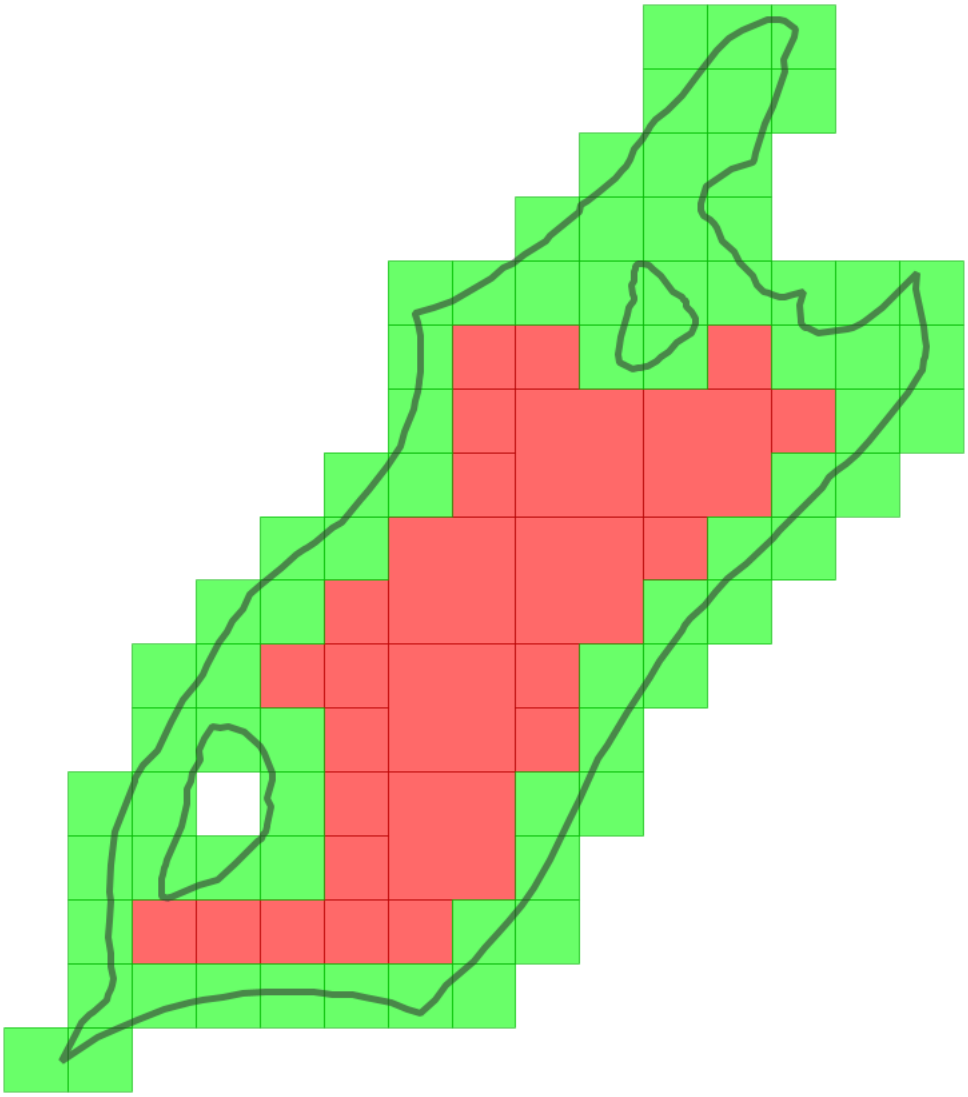


Figure 5: Sample RADIG output (Base 10, maximum division level = 4, divisor 2x5, EA Threshold = 0.8)

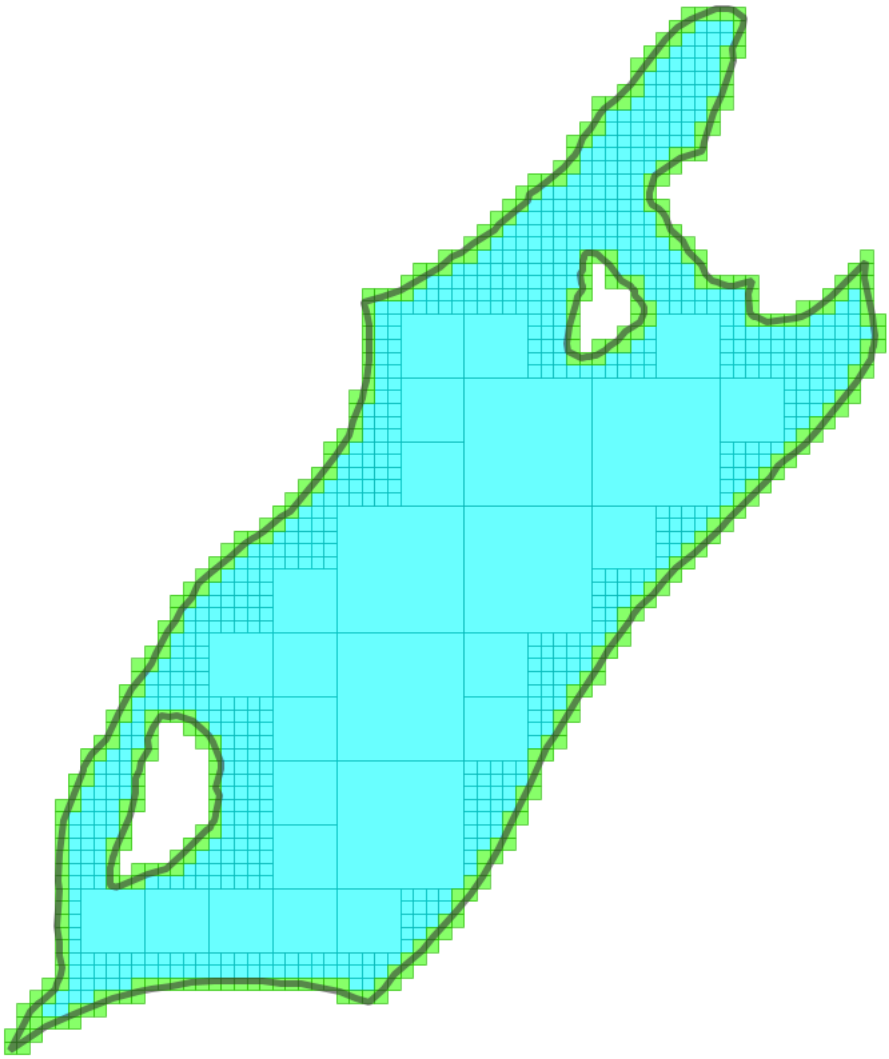


Figure 6: Base 10, divLevel = 5, 2x5 divisors

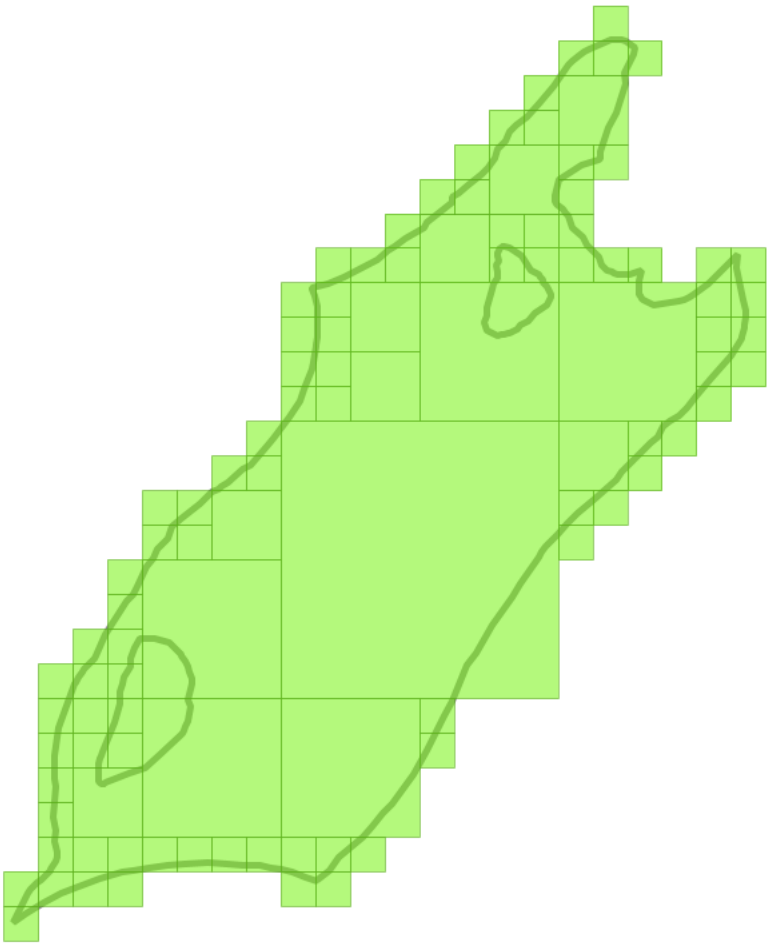


Figure 7: Base 2, divLevel 6, intersects only

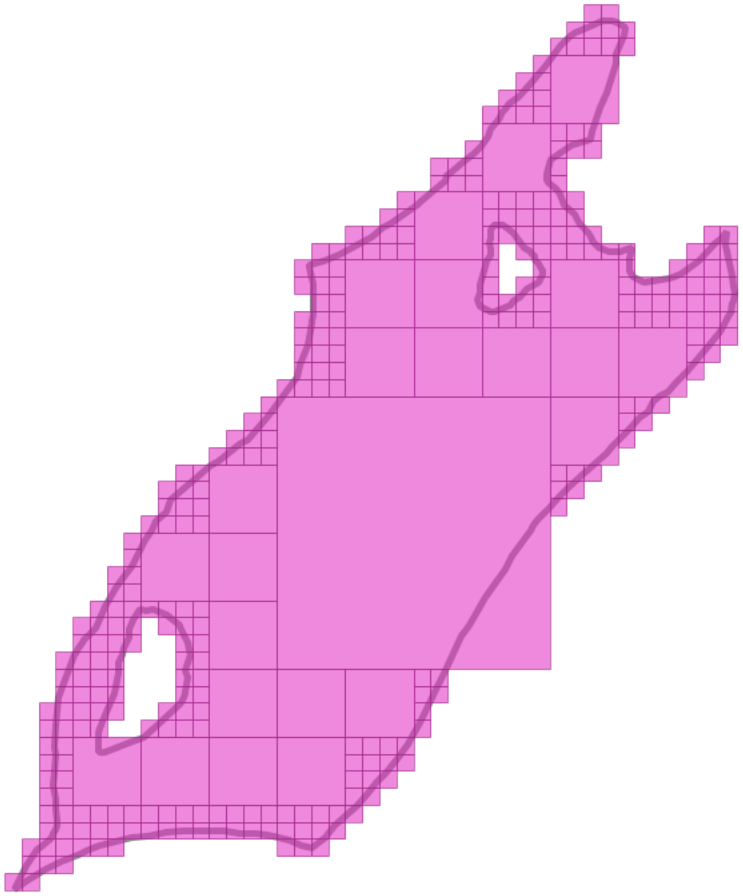


Figure 8: Base 16, divLevel 4, 4x4 divisors, intersects only

5.3 Comparison of space utilisation with Mosaic BNG:

Table 3 shows the index statics for landformArea features from a small sample area. The optimal resolution for BNG reference was calculated by Databricks Mosaic API, which result in 66238 references generated for all these features and the total area of the grid cells is 1652230.0.

The next three rows are results from adaptive grid reference generation of BNGRadig, the RADIG using BNG grid parameters. Minimum resolution is set to 5 to match Mosaic grid; divLevel is set to a very large value of 15 so that subdivision will always reach the minimum resolution if necessary; three EA values are used to demonstrate the impact of EA ratio settings.

As expected, the result show that a smaller EA will generate far fewer grid references compared to Mosaic but at a price of increased grid area and lower space utilisation. Increase in EA will result in increase in number of references (but always fewer than Mosaic BNG) and better space utilisation. An EA of 1.0 will

still generate fewer grid references with the same optimal space utilisation. It should be noted these are achieved in addition to the benefit of the ability to make spatial queries by substring tests.

Table 3: Sample Topobase landformArea data (30km x 30km) with 1190 features

	Parameter	Reference count	Total grid cell aera	Largest cell edge length
BNG (Mosaic)	Optimal Res = -6(5.0m)	66238	1652230.0	5
BNGRadig	DivLevel = 15, minRes = 5.0, ea = 0.5	38281	1725418.0	100
BNGRadig	DivLevel = 15, minRes = 5.0, ea = 0.8	50993	1653578.0	50
BNGRadig	DivLevel = 15, minRes=5.0, ea = 1.0	58932	1652328.0	50

5.4 Performance comparison between Mosaic and Radig Substringing:

Below is the spatial join between GB address points and buildings. Mosaic data were indexed with a fixed 100m resolution. Radig encoding were applied with two different set of parameters (1m fixed resolution for address points and adaptive resolution for buildings with a maximum of 1 and 2 levels of subdivisions respectively). Although substringing process generates a lot more extra rows, we were surprised to find out that the running time of Radig with reference substring is less than twice the time Mosaic performed on fixed grid resolution.

	Source Data	Mosaic (100m)	Radig (I) – one-way	Radig (II) – one-way
Addresses (row)	40,298,590	40,298,590	511,407,385 (1metre)	511,407,385 (1metre)
Buildings (row)	14,767,492	20,493,046	211,945,870(2div/1m)	89,216,889(1div/1m)
Time (ave. of 3 runs)		1.84min	3.55min	3.36min

Note that this result is from an ideal situation where one side of the join is a point set so we need to test containment from polygon to point only and both keys may be optimised. If we have to test both ways, the performance will deteriorate (to the level of around 10min).

5.5 RADIG Match Trie with Partition Key

The spatial join of the same dataset using partition key and RADIG match Trie takes about 4.5min (average of 3 runs, two-way matching). This has been very close to Mosaic's performance.

6 Summary

In this report we present RADIG, a generic framework for generating grid-based spatial index. It supports mixed grid references at different resolutions and coarse spatial query using string operations. Different number systems may be used for encoding. As an application of this framework, we present a RADIG system that aligns to the specification of BNG.

Compared to fixed-resolution grids such as BNG, RADIG offers smaller reference set, similar index space utilisation and most importantly, inter-operability among references at different resolutions. It will be especially suitable for indexing dataset with huge size variation among different features, or to work on two or more dataset with different typical size of features.

References: (to be added)