TECHNICAL WHITE PAPER

COTS Embedded Database Solving Dynamic Points-of-Interest

Abstract

article looks at how commercial off the shelf (COTS) database solutions can be utilized to solve the dynamic point of interest problem.

Contents

1 INTRODUCTION

Navigation systems found in cars and handheld units provide an efficient and easy way to lookup points of interest in a city, find the best route to a user's destination and provide other regionally-based operations by efficiently managing map points. A typical query for these devices is for example locating the nearest Italian restaurant based on the current position of the vehicle. Proprietary data sets and indexing algorithms are used to solve these types of queries, but a major drawback with most of these devices is that they are 'read only' to prevent users from making changes and corrupting the dataset. Updates to datasets must be done in batch mode, and the complete dataset and indexes must be rebuilt on a regular basis because new businesses are constantly cropping up and new roads and buildings are being constructed. Vendors are therefore unable to offer customers localized datasets which could be sold at gas stations and other venues, nor can users make route calculations on the fly to avoid ad-hoc obstacles like accidents.

2 COMMERCIAL OFF THE SHELF (COTS) SOLUTION

COTS embedded databases are designed to manage changing datasets and indexes without the chance of data corruption, but they have a different problem - many don't support two dimensional indexes needed to efficiently manage points-of-interest. If we can find a solution where COTS engines could be used, navigation vendors would be able to design more robust devices and provide new services to their customers.

The problem stems from the fact that a point of interest must be indexed based on both its longitude and latitude value where neither of the two values is favored. By default a one dimensional index will favor one of the two values making a range query very inefficient, which is the main reason why vendors create their own proprietary solutions.

Longitude Values

Latitude Values

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)	(10,0)	(11,0)	(12,0)	(13,0)	(14,0)	(15,0)
1	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)	(10,1)	(11,1)	(12,1)	(13,1)	(14,1)	(15,1)
2	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	(10,2)	(11,2)	(12,2)	(13,2)	(14,2)	(15,2)
3	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)	(11,3)	(12,3)	(13,3)	(14,3)	(15,3)
4	(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	(10,4)	(11,4)	(12,4)	(13,4)	(14,4)	(15,4)
5	(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)	(10,5)	(11,5)	(12,5)	(13,5)	(14,5)	(15,5)
6	(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)	(7,6)	(8,6)	(9,6)	(10,6)	(11,6)	(12,6)	(13,6)	(14,6)	(15,6)
7	(0,7)	(1,7)	(2,7)	(3,7)	(4,7)	(5,7)	(6,7)	(7,7)	(8,7)	(9,7)	(10,7)	(11,7)	(12,7)	(13,7)	(14,7)	(15,7)
8	(0,8)	(1,8)	(2,8)	(3,8)	(4,8)	(5,8)	(6,8)	(7,8)	(8,8)	(9,8)	(10,8)	(11,8)	(12,8)	(13,8)	(14,8)	(15,8)
9	(0,9)	(1,9)	(2,9)	(3,9)	(4,9)	(5,9)	(6,9)	(7,9)	(8,9)	(9,9)	(10,9)	(11,9)	(12,9)	(13,9)	(14,9)	(15,9)
10	(0,10)	(1,10)	(2,10)	(3,10)	(4,10)	(5,10)	(6,10)	(7,10)	(8,10)	(9,10)	(10,10)	(11,10)	(12,10)	(13,10)	(14,10)	(15,10)
11	(0,11)	(1,11)	(2,11)	(3,11)	(4,11)	(5,11)	(6,11)	(7,11)	(8,11)	(9,11)	(10,11)	(11,11)	(12,11)	(13,11)	(14,11)	(15,11)
12	(0,12)	(1,12)	(2,12)	(3,12)	(4,12)	(5,12)	(6,12)	(7,12)	(8,12)	(9,12)	(10,12)	(11,12)	(12,12)	(13,12)	(14,12)	(15,12)
13	(0,13)	(1,13)	(2,13)	(3,13)	(4,13)	(5,13)	(6,13)	(7,13)	(8,13)	(9,13)	(10,13)	(11,13)	(12,13)	(13,13)	(14,13)	(15,13)
14	(0,14)	(1,14)	(2,14)	(3,14)	(4,14)	(5,14)	(6,14)	(7,14)	(8,14)	(9,14)	(10,14)	(11,14)	(12,14)	(13,14)	(14,14)	(15,14)
15	(0,15)	(1,15)	(2,15)	(3,15)	(4,15)	(5,15)	(6,15)	(7,15)	(8,15)	(9,15)	(10,15)	(11,15)	(12,15)	(13,15)	(14,15)	(15,15)

Figure 1

Figure 1 is a longitude, latitude grid of all points in a system where both the longitude and latitude are coded as 4 bit values. In real world applications these values would be 32 bit but for the simplicity of describing both the problem and solution, the maps in this article will be based on 4 bit data types.

A one dimensional index, such as a B-Tree, will only provide an efficient longitude/latitude index in one direction. In the figure, the longitude values are illustrated as the most significant information so the index will only have vertical efficiency. Think of the value in each cell as what the index sees, and the index is sorted from low to high.

If we add a region query based on the bounding box between (5, 5) to (9, 8) this would translate to a range scan of the index between the two values. Figure 2 illustrates exactly that.

Longitude Values (columns) / Latitude Values (rows)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)	(10,0)	(11,0)	(12,0)	(13,0)	(14,0)	(15,0)
1	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)	(10,1)	(11,1)	(12,1)	(13,1)	(14,1)	(15,1)
2	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	(10,2)	(11,2)	(12,2)	(13,2)	(14,2)	(15,2)
3	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)	(11,3)	(12,3)	(13,3)	(14,3)	(15,3)
4	(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	(10,4)	(11,4)	(12,4)	(13,4)	(14,4)	(15,4)
5	(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)	(10,5)	(11,5)	(12,5)	(13,5)	(14,5)	(15,5)
6	(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)	(7,6)	(8,6)	(9,6)	(10,6)	(11,6)	(12,6)	(13,6)	(14,6)	(15,6)
7	(0,7)	(1,7)	(2,7)	(3,7)	(4,7)	(5,7)	(6,7)	(7,7)	(8,7)	(9,7)	(10,7)	(11,7)	(12,7)	(13,7)	(14,7)	(15,7)
8	(0,8)	(1,8)	(2,8)	(3,8)	(4,8)	(5,8)	(6,8)	(7,8)	(8,8)	(9,8)	(10,8)	(11,8)	(12,8)	(13,8)	(14,8)	(15,8)
9	(0,9)	(1,9)	(2,9)	(3,9)	(4,9)	(5,9)	(6,9)	(7,9)	(8,9)	(9,9)	(10,9)	(11,9)	(12,9)	(13,9)	(14,9)	(15,9)
10	(0,10)	(1,10)	(2,10)	(3,10)	(4,10)	(5,10)	(6,10)	(7,10)	(8,10)	(9,10)	(10,10)	(11,10)	(12,10)	(13,10)	(14,10)	(15,10)
11	(0,11)	(1,11)	(2,11)	(3,11)	(4,11)	(5,11)	(6,11)	(7,11)	(8,11)	(9,11)	(10,11)	(11,11)	(12,11)	(13,11)	(14,11)	(15,11)
12	(0,12)	(1,12)	(2,12)	(3,12)	(4,12)	(5,12)	(6,12)	(7,12)	(8,12)	(9,12)	(10,12)	(11,12)	(12,12)	(13,12)	(14,12)	(15,12)
13	(0,13)	(1,13)	(2,13)	(3,13)	(4,13)	(5,13)	(6,13)	(7,13)	(8,13)	(9,13)	(10,13)	(11,13)	(12,13)	(13,13)	(14,13)	(15,13)
14	(0,14)	(1,14)	(2,14)	(3,14)	(4,14)	(5,14)	(6,14)	(7,14)	(8,14)	(9,14)	(10,14)	(11,14)	(12,14)	(13,14)	(14,14)	(15,14)
15	(0,15)	(1,15)	(2,15)	(3,15)	(4,15)	(5,15)	(6,15)	(7,15)	(8,15)	(9,15)	(10,15)	(11,15)	(12,15)	(13,15)	(14,15)	(15,15)

Figure 2

Not only are the yellow points returned to the application but also all the false gray points that are clearly outside the bounding box. This inefficiency is the major reason for navigational vendors implementing proprietary solutions based on two dimensional indexing algorithms like R-Tree or any of its cousins. By making the proprietary implementation read only, the vendors don't need to implement transactional safety or concurrency control to avoid corruption or allow multi-threaded application access to the data.

3 IMPLEMENTATION

This article will describe a solution to the two dimensional problem introducing a new data structure type called an R-Tree. This not only allows the vendors to efficiently query these points-of- interest but also allows them to dynamically add other one dimensional information to the query, like type (gas station, restaurant, etc), which can't be solved with a two dimensional index system.

If you look at the previous figure, the one dimensional index is inefficient since its most significant piece of information is taken from the Longitude values and the second most significant from the latitude, thus creating the undesired vertical efficiency. If you translate the points into the bit pattern used by the indexing system it will look like this:

Longitude Values

Latitude Values	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00000000	00010000	00100000	00110000	01000000	01010000	01100000	01110000	10000000	10010000	10100000	10110000	11000000	11010000	11100000	11110000
1	00000001	00010001	00100001	00110001	01000001	01010001	01100001	01110001	10000001	10010001	10100001	10110001	11000001	11010001	11100001	11110001
2	00000010	00010010	00100010	00110010	01000010	01010010	01100010	01110010	10000010	10010010	10100010	10110010	11000010	11010010	11100010	11110010
3	00000011	00010011	00100011	00110011	01000011	01010011	01100011	01110011	10000011	10010011	10100011	10110011	11000011	11010011	11100011	11110011
4	00000100	00010100	00100100	00110100	01000100	01010100	01100100	01110100	10000100	10010100	10100100	10110100	11000100	11010100	11100100	11110100
5	00000101	00010101	00100101	00110101	01000101	01010101	01100101	01110101	10000101	10010101	10100101	10110101	11000101	11010101	11100101	11110101
6	00000110	00010110	00100110	00110110	01000110	01010110	01100110	01110110	10000110	10010110	10100110	10110110	11000110	11010110	11100110	11110110
7	00000111	00010111	00100111	00110111	01000111	01010111	01100111	01110111	10000111	10010111	10100111	10110111	11000111	11010111	11100111	11110111
8	00001000	00011000	00101000	00111000	01001000	01011000	01101000	01111000	10001000	10011000	10101000	10111000	11001000	11011000	11101000	11111000
9	00001001	00011001	00101001	00111001	01001001	01011001	01101001	01111001	10001001	10011001	10101001	10111001	11001001	11011001	11101001	11111001
10	00001010	00011010	00101010	00111010	01001010	01011010	01101010	01111010	10001010	10011010	10101010	10111010	11001010	11011010	11101010	11111010
11	00001011	00011011	00101011	00111011	01001011	01011011	01101011	01111011	10001011	10011011	10101011	10111011	11001011	11011011	11101011	11111011
12	00001100	00011100	00101100	00111100	01001100	01011100	01101100	01111100	10001100	10011100	10101100	10111100	11001100	11011100	11101100	11111100
13	00001101	00011101	00101101	00111101	01001101	01011101	01101101	01111101	10001101	10011101	10101101	10111101	11001101	11011101	11101101	11111101
14	00001110	00011110	00101110	00111110	01001110	01011110	01101110	01111110	10001110	10011110	10101110	10111110	11001110	11011110	11101110	11111110
15	00001111	00011111	00101111	00111111	01001111	01011111	01101111	01111111	10001111	10011111	10101111	10111111	11001111	11011111	11101111	11111111

Figure 3

The four most significant bits ($x_4x_3x_2x_4$) stem from the Longitude value and the 4 second most significant bits ($y_4y_3y_2y_4$) stem from the Latitude. One solution to this without the use of geospatial data types is to introduce the Z-value[1], which is made up of interleaving the most significant bits from both dimensions, resulting in values on the form ($y_4x_4y_3x_3y_2x_2y_1x_1$). If our one dimensional index uses this bit pattern as its index values we'll be indexing something where the most significant information from both dimensions is taken into consideration. Figure 4 shows the pattern.

1 Morton, G. M. (1966), A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing, Technical Report, Ottawa, Canada: IBM Ltd.

Longitude Values

Latitude Values

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00000000 0	00000001 1	00000100 4	00000101 5	00010000 16	00010001 17	00010100 20	00010101 21	01000000 64	01000001 65	01000100 68	01000101 69	01010000 80	01010001 81	01010100 84	01010101 85
1	00000010 2	00000011 3	00000110 6	00000111 7	00010010 18	00010011 19	00010110 22	00010111 23	01000010 66	01000011 67	01000110 70	01000111 71	01010010 82	01010011 83	01010110 86	01010111 87
2	00001000 8	00001001 9	00001100 12	00001101 13	00011000 24	00011001 25	00011100 28	00011101 29	01001000 72	01001001 73	01001100 76	01001101 77	01011000 88	01011001 89	01011100 92	01011101 93
3	00001010 10	00001011 11	00001110 14	00001111 15	00011010 26	00011011 27	00011110 30	00011111 31	01001010 74	01001011 75	01001110 78	01001111 79	01011010 90	01011011 91	01011110 94	01011111 95
4	00100000 32	00100001 33	00100100 36	00100101 37	00110000 48	00110001 49	00110100 52	00110101 53	01100000 96	01100001 97	01100100 100	01100101 101	01110000 112	01110001 113	01110100 116	01110101 117
5	00100010 34	00100011 35	00100110 38	00100111 39	00110010 50	00110011 51	00110110 54	00110111 55	01100010 98	01100011 99	01100110 102	01100111 103	01110010 114	01110011 115	01110110 118	01110111 119
6	00101000 40	00101001 41	00101100 44	00101101 45	00111000 56	00111001 57	00111100 60	00111101 61	01101000 104	01101001 105	01101100 108	01101101 109	01111000 120	01111001 121	01111100 124	01111101 125
7	00101010 42	00101011 43	00101110 46	00101111 47	00111010 58	00111011 59	00111110 62	00111111 63	01101010 106	01101011 107	01101110 110	01101111 111	01111010 122	01111011 123	01111110 126	01111111 127
8	10000000 128	10000001 129	10000100 132	10000101 133	10010000 144	10010001 145	10010100 148	10010101 149	11000000 192	11000001 193	11000100 196	11000101 197	11010000 208	11010001 209	11010100 212	11010101 213
9	10000010 130	10000011 131	10000110 134	10000111 135	10010010 146	10010011 147	10010110 150	10010111 151	11000010 194	11000011 195	11000110 198	11000111 199	11010010 210	11010011 211	11010110 214	11010111 215
10	10001000 136	10001001 137	10001100 140	10001101 141	10011000 152	10011001 153	10011100 156	10011101 157	11001000 200	11001001 201	11001100 204	11001101 205	11011000 216	11011001 217	11011100 220	11011101 221
11	10001010 138	10001011 139	10001110 142	10001111 143	10011010 154	10011011 155	10011110 158	10011111 159	11001010 202	11001011 203	11001110 206	11001111 207	11011010 218	11011011 219	11011110 222	11011111 223
12	10100000 160	10100001 161	10100100 164	10100101 165	10110000 176	10110001 177	10110100 180	10110101 181	11100000 224	11100001 225	11100100 228	11100101 229	11110000 240	11110001 241	11110100 244	11110101 245
13	10100010 162	10100011 163	10100110 166	10100111 167	10110010 178	10110011 179	10110110 182	10110111 183	11100010 226	11100011 227	11100110 230	11100111 231	11110010 242	11110011 243	11110110 246	11110111 247
14	10101000 168	10101001 169	10101100 172	10101101 173	10111000 184	10111001 185	10111100 188	10111101 189	11101000 232	11101001 233	11101100 236	11101101 237	11111000 248	11111001 249	11111100 252	11111101 253
15	10101010 170	10101011 171	10101110 174	10101111 175	10111010 186	10111011 187	10111110 190	10111111 191	11101010 234	11101011 235	11101110 238	11101111 239	11111010 250	11111011 251	11111110 254	11111111 255

Figure 4

For illustration purposes each cell has both the binary and decimal representation of the Z-Values. Instead of having vertical efficiency we've now managed to introduce square like efficiency, but we still can't take any random region and expect an efficient result. If our bounding box happened to be from (0, 0) to (3, 3) or (6, 6) to (7, 7) we'd be extremely efficient since any query will be mapped to a range query and all points in our region have contiguous increasing value. E.g. the (0, 0) to (3, 3) bounding box resulting in a range query from 0 to 15.

Let's look at the range query from (5, 5) to (9, 8) that we were initially working on, How would that look?

Longitude Values

Latitude Values	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	00000000 0	00000001 1	00000100 4	00000101 5	00010000 16	00010001 17	00010100 20	00010101 21	01000000 64	01000001 65	01000100 68	01000101 69	01010000 80	01010001 81	01010100 84	01010101 85
1	00000010 2	00000011 3	00000110 6	00000111 7	00010010 18	00010011 19	00010110 22	00010111 23	01000010 66	01000011 67	01000110 70	01000111 71	01010010 82	01010011 83	01010110 86	01010111 87
2	00001000 8	00001001 9	00001100 12	00001101 13	00011000 24	00011001 25	00011100 28	00011101 29	01001000 72	01001001 73	01001100 76	01001101 77	01011000 88	01011001 89	01011100 92	01011101 93
3	00001010 10	00001011 11	00001110 14	00001111 15	00011010 26	00011011 27	00011110 30	00011111 31	01001010 74	01001011 75	01001110 78	01001111 79	01011010 90	01011011 91	01011110 94	01011111 95
4	00100000 32	00100001 33	00100100 36	00100101 37	00110000 48	00110001 49	00110100 52	00110101 53	01100000 96	01100001 97	01100100 100	01100101 101	01110000 112	01110001 113	01110100 116	01110101 117
5	00100010 34	00100011 35	00100110 38	00100111 39	00110010 50	00110011 51	00110110 54	00110111 55	01100010 98	01100011 99	01100110 102	01100111 103	01110010 114	01110011 115	01110110 118	01110111 119
6	00101000 40	00101001 41	00101100 44	00101101 45	00111000 56	00111001 57	00111100 60	00111101 61	01101000 104	01101001 105	01101100 108	01101101 109	01111000 120	01111001 121	01111100 124	01111101 125
7	00101010 42	00101011 43	00101110 46	00101111 47	00111010 58	00111011 59	00111110 62	00111111 63	01101010 106	01101011 107	01101110 110	01101111 111	01111010 122	01111011 123	01111110 126	01111111 127
8	10000000 128	10000001 129	10000100 132	10000101 133	10010000 144	10010001 145	10010100 148	10010101 149	11000000 192	11000001 193	11000100 196	11000101 197	11010000 208	11010001 209	11010100 212	11010101 213
9	10000010 130	10000011 131	10000110 134	10000111 135	10010010 146	10010011 147	10010110 150	10010111 151	11000010 194	11000011 195	11000110 198	11000111 199	11010010 210	11010011 211	11010110 214	11010111 215
10	10001000 136	10001001 137	10001100 140	10001101 141	10011000 152	10011001 153	10011100 156	10011101 157	11001000 200	11001001 201	11001100 204	11001101 205	11011000 216	11011001 217	11011100 220	11011101 221
11	10001010 138	10001011 139	10001110 142	10001111 143	10011010 154	10011011 155	10011110 158	10011111 159	11001010 202	11001011 203	11001110 206	11001111 207	11011010 218	11011011 219	11011110 222	11011111 223
12	10100000 160	10100001 161	10100100 164	10100101 165	10110000 176	10110001 177	10110100 180	10110101 181	11100000 224	11100001 225	11100100 228	11100101 229	11110000 240	11110001 241	11110100 244	11110101 245
13	10100010 162	10100011 163	10100110 166	10100111 167	10110010 178	10110011 179	10110110 182	10110111 183	11100010 226	11100011 227	11100110 230	11100111 231	11110010 242	11110011 243	11110110 246	11110111 247
14	10101000 168	10101001 169	10101100 172	10101101 173	10111000 184	10111001 185	10111100 188	10111101 189	11101000 232	11101001 233	11101100 236	11101101 237	11111000 248	11111001 249	11111100 252	11111101 253
15	10101010 170	10101011 171	10101110 174	10101111 175	10111010 186	10111011 187	10111110 190	10111111 191	11101010 234	11101011 235	11101110 238	11101111 239	11111010 250	11111011 251	11111110 254	11111111 255

Figure 5

A straight forward range query of this region would return a whole lot of false points, (the grey points), which make this just as inefficient as the previous discussion. So the question is, can we optimize our query? The answer is yes. Since our points are now organized in squares we can take the region and break it down into multiple squares of contiguous increasing index values. You'll see from the rest of this discussion that the proposed algorithm for calculating these squares will result in our yellow region being broken into 5 smaller squares, scanned separately, optimizing out most of the false points.

00110011 51	00110110 54	00110111 55	01100010 98	01100011 99
00111001 57	00111100 60	00111101 61	01101000 104	01101001 105
00111011 59	00111110 62	00111111 63	01101010 106	01101011 107
10010001 145	10010100 148	10010101 149	11000000 192	11000001 193

We'll start out by doing a range query from 51 to 193, which are our bounding box values. As we scan we'll decide to calculate a region division if 3 false points are reported. We allow up to 3 false points to avoid too many division calculations. In our case the first division calculation will take place when the last reported point is above 63 and lower than 98 given that there are more than 3 points to report in this interval.

To illustrate the division calculation we've broken out the bounding box values in binary form:

```
51  = 00110011 = (0101,0101), and
193 = 11000001 = (1001,1000)
```

The first thing we do is look at the Z value and determine the first significant bit that differs based on the values; this will determine whether we're looking at a vertical or a horizontal division. With the above numbers the identified bit is z_8 which translates to the y_4 bit. Y bits will result in a horizontal split, x bits vertical.

Since we've identified a vertical split we know that the upper x boundary can be inherited from the 193 value and the lower x boundary from the 51. The two points we want to identify in this division (we'll call them LitMax and BigMin) are 107 and 145 which are the highest number in the upper square and lowest number in the bottom square divided by the red line. What we don't know is the y value just above and below this division line. LitMax's y value can be calculated as 'all common most significant bits' from the bounding box y values followed by a 0 and then 1's, and the BigMin's y value would be 'all common most significant bits' followed by 1 and then 0's.

So in our case we take the y's from 51 and 193 and look at the bit patterns:

```
51’s y = 0101
193’2 y = 1000
```

This gives us the LitMax y value to be 0111 (no common bits), and BigMin's y value 1000 resulting in the LitMax point being (1001,0111) and BigMin's point being (0101,1000) interleaved resulting in 01101011 = 107 and 10010001 = 145.

Now that the calculation is done, our initial bounding box (5,5) to (9,8) is split in two - (5,5) to (9,7) and (5,8) to (9,8) . Since our last reported point was between 63 and 98 (less than LitMax) there is a chance of finding more valid points in the first region so we do a recursive call into the division algorithm with the new bounding box values, 51 to 107.

```
51  = 00110011 = (0101,0101), and
107 = 01101011 = (1001,0111)
```

Identifying that x_4 is the first significant bit that changes, means a vertical split where we inherit the y values from the bounding box extent. The LitMax and BigMin values are now computed based on the x values in the same way as our first calculation, resulting in the LitMax point being (0111,0111) and BigMin point being (1000,0101) interleaved 00111111 = 63 and 01100010 = 98 dividing along the blue line. We still know that the last valid point was between 63 and 98, so we can start a regular scan from 98 and 107 which is our new bounding box. The new scan will result in new splits but at one point we'll return to the last reported value being larger than 107 and we'll start a regular scan from the BigMin calculated in our first split.

With our original bounding box we'll end up with 4 splits and 4 scans with a maximum of (4-1) * 3 false points reported.

There are another couple of observations to make: first the z value is a loss less representation of the x and y, so by storing z there is no need to store the x and y value. Another observation is that z is now just indexed as a one dimensional value; pre-pending it with any other one dimensional value would allow for even more specialized queries. Say you pre-pend it with type information than you can efficiently index any gas stations, any restaurants or any other type that you may discover at runtime without needing to ship the software with a predefined set of types.

RDM does all of this behind the scenes when you use the geospatial datatypes it supports. Any optimizations necessary are done automatically by the database engine and thus the user does not need to worry about implementing any of their own code for the above design.

4 CONCLUSION

To conclude the above discussion allows commercial off the shelf databases to dynamically and efficiently manage points-of-interest data. It also supports on-device changes of data without the possibility of corruption. RDM handles all of this internally with the use of its geospatial data types and r-tree implementation. RDM takes advantage of the algorithmic benefits listed above but also allows for the ease of use for the developer with the geospatial datatypes and bounding boxes implementation. More information can be found at: https://docs.raima.com/rdm/14_1/r-tree.html.

5 COMPLETE SOURCE

Please download Raima's RDM technology for a running example of the problem described in this article, https://raima.com/download-table/.

6 REFERENCES

Z-order (curve). (2008, July 26). In *Wikipedia, The Free Encyclopedia*. Retrieved 19:44, September 10, 2008, from http://en.wikipedia.org/w/index.php?title=Z-order_(curve)&oldid=228028274

Morton, G. M. (1966), *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing,* Technical Report, Ottawa, Canada: IBM Ltd.

Tropf, H. & Herzog, H. (1981), "Multidimensional Range Search in Dynamically Balanced Trees," *Angewandte Informatik* 2: 71–77.

Want to know more?

Please call us to discuss your database needs or email us at info@raima.com. You may also visit our website for the latest news, product downloads and documentation: www.raima.com.

Headquarter: 720 Third Avenue Suite 1100, Seattle, WA 98104, USA T: +1 206 748 5300

Europe: Stubbings House, Henley Road, Maidenhead, UK SL6 6QL T: +44 1628 826 800