Boris Muratshin @zzeng 91.0 karma 57.6 rating
User

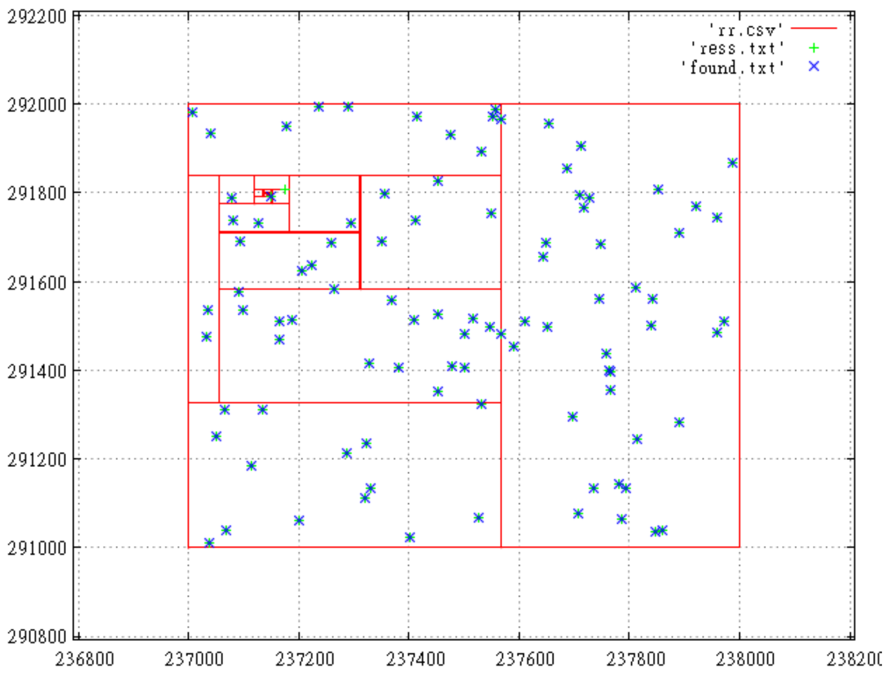
23 Publications 128 Comments 37 Favorites 34 Followers

Profile

January 18 at 07:40

Development → Z-order vs R-tree , continued

Geoinformation Services * algorithms *, the PostgreSQL *, the C *



The last time we came to the conclusion that for the effective operation of the spatial index based on the Z-order need to do 2 things:

- efficient algorithm to obtain sub-intervals
- a low-level job with the B-tree

That is exactly what we are going to do under the cut.

Let's start with the more interesting algorithm.

Splitting into sub-queries

We consider 2-dimensional algorithm (indexed points) because of its relative simplicity. However, the algorithm can easily be generalized to higher dimensions.

Now we are (for simplicity) we will use non-negative integer coordinates. Coordinates are limited to 32 bits, so the value of the index key can be stored in uint64

We need the following simple properties of z-numbering:

1. Suppose a marked rectangle in the plane. Then, among the lowest points of the rectangle z-number is the lower left corner of the rectangle (we call it "z"), and the largest - the upper right corner (it will be called "Z"). This property clearly follows from the construction of z-rooms.

2. Each rectangle can be divided into two uniquely rectangle (vertical or horizontal cut) so that all the z-number of the first rectangle is less than all of the second z-rooms. This follows from the self-similarity of Z-curve. The unit cell of four cells is divided in half, then in half two more cuts, the same thing is happening at all levels of the hierarchy.

As it is necessary to cut the extent to preserve the continuity of sub-intervals?

From the self-similar curve, which can be cut only by the lattice pitch at a power of two and to a node at the origin.

But what specific array of available 64 to choose? It's pretty simple. Being cut extent should occupy in the desired lattice more than one cell, there is simply nothing else to cut. On the other hand, the coordinates of any size can not exceed 2 cells, and at least one must be strictly 2 cells. If both dimensions being cut Extent occupies 2 cells will be cut along the coordinate, whose rank in the construction of Z-values older.

How to find a grill? This is also a snap. It is enough to compare the Z-values of the angles z and Z. begin to compare with the senior categories and find the category where their values are separated. That and the desired lattice.

How to make the cut? Here it is necessary to recall the method for constructing the Z-values, namely that x & y coordinates are alternated through the discharge. Consequently:

- Let z and the difference between the Z discharge started in m
- Will be cut along one coordinate, which had m, regardless of whether, x or y is, even in this case, x, however, for all y operates in the same way
- Because the extent (x0, y0, x1, y1) we get two: (x0, y0, x2-1, y1), (x2, y0, x1, y1)
- And in order to get enough x2 reset all bits of x coordinates under m, ie through a
- x2-1 turns zeroing m bits and assigning 1 x all junior ranks

So how do you look algorithm for generating sub-queries? Quite unpretentious:

1. We get all sub-queries, initially in the queue a single item - the desired extent
2. While the queue is not empty:

    1. We get the element with the top of the queue
    2. If this request is not working **stop criterion**
        1. We get the z and Z - values for its corner
        2. We compare z and Z - find rank m, for which we shall cut
        3. The above-described manner, the two subqueries
        4. We put them in a queue, first one with a large Z-values, then the second

Such a process ensures that we generate sub-queries, where the Z-value of the final (ie, those that worked stopping criterion) subqueries only increase, icing the back does not arise.

criterion stop

It is very important, if they are ignored, the generation of sub-queries will continue as long as everything is not cut into individual squares.

It is worth noting that the development of such a test we can not rely only on the parameters of the query, the square, the geometric properties of ... The real distribution of data points can be very uneven, for example, the city and empty spaces. The population of the areas we known in advance.

So we need to integrate with the search in the index tree, which, as we recall, B-tree.

What is a subquery in terms of the index? This set of data residing on a certain number of pages. A subquery is empty, and empty data slot, but nevertheless somewhere looks as to understand that there is no data, it is necessary to try to read them and come down from the top of the tree to the leaf page. It may happen that an empty query and looks beyond the tree.

Generally, sub-query data subtraction process consists of two phases -

- Sounding tree. Since we are looking for the root page of the key is less than or equal to the Z-value of the lower-left corner of the subquery (z) and so down to the leaf page. The output is a stack of pages and the leaf page "in the air". On this page, look for the element is greater than or equal to the claim.

- Read ahead to the end of the subquery. In PostgreSQL, leaf pages B-tree linked list, if it was not, in order to get the next page would have to climb up on a stack of pages, to then go down on her.

So, after the probe request is in our hands leaf page, which presumably begin our data. There are different situations:

1. Found item page over the top right corner of the subquery (Z). Those. No data.
2. Found a page element is less than Z, the last item of the page is less than Z. That is, subquery begins on this page, it ends somewhere further.

3. Found a page element is less than Z, the last page element Z. That is more than all subquery is located on this page.
4. Found a page element is less than Z, the last element of the page is equal to Z. That is, subquery begins on this page, but it may end in the next (few elements with some coordinates). And maybe much more if a lot of duplicates.

N1 option requires no action. For N2 natural to the next stop criterion (subqueries splitting) - will cut them up until we get options 3 or 4. With all the obvious option N3, N4 in the case of data pages can be a little, but because it is senseless to cut the subquery . a subsequent page (Zach) can only be equal to the key data with Z, after the cut we are in the same situation. To cope with this, it is enough just to consider the following (ing) all the data pages with the key equal to Z. They can not be, in general, N4 - this is quite an exotic case.

Working with the B-tree

The low-level job with the B-tree does not present any difficulty. But the need to create an extension . The general logic is - will register SRF function:

```
CREATE TYPE __ret_2d_lookup AS (c_tid TID, x integer, y integer); CREATE FUNCTION zcurve_2d_lookup(text, integer, integer, integer, i
nteger) RETURNS SETOF __ret_2d_lookup AS 'MODULE_PATHNAME' LANGUAGE C IMMUTABLE STRICT;
```

```
x integer, y integer);
```

```
CREATE TYPE __ret_2d_lookup AS (c_tid TID, x integer, y integer); CREATE FUNCTION zcurve_2d_lookup(text, integer, integer, integer, i
nteger) RETURNS SETOF __ret_2d_lookup AS 'MODULE_PATHNAME' LANGUAGE C IMMUTABLE STRICT;
```

```
integer, integer, integer)
```

```
CREATE TYPE __ret_2d_lookup AS (c_tid TID, x integer, y integer); CREATE FUNCTION zcurve_2d_lookup(text, integer, integer, integer, i
nteger) RETURNS SETOF __ret_2d_lookup AS 'MODULE_PATHNAME' LANGUAGE C IMMUTABLE STRICT;
```

Which receives the input of the name of the index and extent. A returns a set of elements, each of which is a pointer to the row table and coordinates.

Access to the property tree is as follows:

```
const char *relname; /* внешний параметр */ ... List *relname_list; RangeVar *relvar; Relation rel; ... relname_list = stringToQualif
iedNameList(relname); relvar = makeRangeVarFromNameList(relname_list); rel = indexOpen(rel); ... indexClose(rel);
```

Obtaining root page:

```
int access = BT_READ; Buffer buf; ... buf = _bt_getroot(rel, access);
```

In general, the search index is made like a normal search in the B-tree (see postgresql / src / backend / access / nbtree / nbtsearch.c). Changes associated with the key characteristics, perhaps you could do without it, even if it would be and a little slower.

Search within a page looks like this:

```
Page page; BTPageOpaque opaque; OffsetNumber low, high; int32 result, cmpval; Datum datum; bool isNull; ... page = BufferGetPage(bu
f); opaque = (BTPageOpaque) PageGetSpecialPointer(page); low = P_FIRSTDATAKEY(opaque); high = PageGetMaxOffsetNumber(page); ... тут и
дёт бинарный поиск ... /* для листовой страницы возвращаем найденное значение */ if (P_ISLEAF(opaque)) return low; /* для промежуточн
ой - предыдущий элемент */ return OffsetNumberPrev(low);
```

```
);
```

```
Page page; BTPageOpaque opaque; OffsetNumber low, high; int32 result, cmpval; Datum datum; bool isNull; ... page = BufferGetPage(bu
f); opaque = (BTPageOpaque) PageGetSpecialPointer(page); low = P_FIRSTDATAKEY(opaque); high = PageGetMaxOffsetNumber(page); ... тут и
дёт бинарный поиск ... /* для листовой страницы возвращаем найденное значение */ if (P_ISLEAF(opaque)) return low; /* для промежуточн
ой - предыдущий элемент */ return OffsetNumberPrev(low);
```

Getting a page element:

```
OffsetNumber offnum; Page page; Relation rel; TupleDesc itupdesc = RelationGetDescr(rel); IndexTuple itup; Datum datum; bool isNull;
 uint64 val; ... itup = (IndexTuple) PageGetItem(page, PageGetItemId(page, offnum)); datum = index_getattr(itup, 1, itupdesc, &isNul
l); val = DatumGetInt64(datum);
```

```
, PageGetItemId (page, offnum));
```

```
OffsetNumber offnum; Page page; Relation rel; TupleDesc itupdesc = RelationGetDescr(rel); IndexTuple itup; Datum datum; bool isNull;
 uint64 val; ... itup = (IndexTuple) PageGetItem(page, PageGetItemId(page, offnum)); datum = index_getattr(itup, 1, itupdesc, &isNul
l); val = DatumGetInt64(datum);
```

The final algorithm

1. We get all sub-queries, initially in the queue a single item - the desired extent
2. While the queue is not empty:
    1. We get the element with the top of the queue
    2. We carry out a probe request in the index for z (lower left corner). For reasons of economy, one can not make sensing each time, but only if the last obtained value (which is initially 0) is greater than or equal to z
    3. If found minimum value greater than Z (upper right corner), completing the processing of the sub-query, go to A2
    4. Check the last element of the leaf page B-tree, which stopped the probe request
    5. If it is greater than or equal to the Z, the elements of the page, select, filter their search for accessories extent at and remember the resulting point.
    6. If it is equal to Z, we read index forward to the complete exhaustion of the keys with a value equal to Z and also remember them
    7. Otherwise - the last value of the page is less than Z

        1. We compare z and Z - find rank m, for which we shall cut
        2. The above-described manner, the two subqueries
        3. We put them in a queue, first one with a large Z-values, then the second

Preliminary results

The title illustration paper presents real-partition request for subqueries and the resulting point. Showing comparison found R-tree with the results obtained by the above algorithm. Picture debug times and it is clear that one point is not enough.

But pictures - pictures and want to see a comparison of productivity. On our side will be the same table:

```
create table test_points (x integer,y integer); COPY test_points from '/home/.../data.csv'; create index zcurve_test_points on test_p
oints(zcurve_val_from_xy(x, y));
```

```
integer);
```

```
create table test_points (x integer,y integer); COPY test_points from '/home/.../data.csv'; create index zcurve_test_points on test_p
oints(zcurve_val_from_xy(x, y));
```

And the type of queries:

```
select count(1) from zcurve_2d_lookup('zcurve_test_points', 500000,500000,501000,501000);
```

Compare going to the R-tree as the standard de facto. Moreover, unlike the last article, we need "index only scan" on the R-tree as Our Z-Index does not apply to the table more.

```
create table test_pt as (select point(x,y) from test_points); create index test_pt_idx on test_pt using gist (point); vacuum test_pt;
```

```
x, y) from test_points);
```

```
create table test_pt as (select point(x,y) from test_points); create index test_pt_idx on test_pt using gist (point); vacuum test_pt;
```

```
point);
```

```
create table test_pt as (select point(x,y) from test_points); create index test_pt_idx on test_pt using gist (point); vacuum test_pt;
```

In such a data request:

```
explain (analyze, buffers) select * from test_pt where point <@ box( point(500000, 500000), point(510000, 510000));
```

```
from test_pt where
```

```
explain (analyze, buffers) select * from test_pt where point <@ box( point(500000, 500000), point(510000, 510000));
```

gives:

```
 QUERY PLAN ------------------------------------------------------------------------------------------- Index Only Scan using test_
pt_idx on test_pt (cost=0.42..539.42 rows=10000 width=16) (actual time=0.075..0.531 rows=968 loops=1) Index Cond: (point <@
'(510000,510000),(500000,500000)'::box) Heap Fetches: 0 Buffers: shared hit=20 Planning time: 0.088 ms Execution time: 0.648 ms (6 ro
ws)
```

```
cost = 0.42..539.42 rows =
```

```
 QUERY PLAN ------------------------------------------------------------------------------------------- Index Only Scan using test_
pt_idx on test_pt (cost=0.42..539.42 rows=10000 width=16) (actual time=0.075..0.531 rows=968 loops=1) Index Cond: (point <@
'(510000,510000),(500000,500000)'::box) Heap Fetches: 0 Buffers: shared hit=20 Planning time: 0.088 ms Execution time: 0.648 ms (6 ro
ws)
```

```
(510000,510000), (500000,500000)' :: box)
```

```
 QUERY PLAN ------------------------------------------------------------------------------------------- Index Only Scan using test_
pt_idx on test_pt (cost=0.42..539.42 rows=10000 width=16) (actual time=0.075..0.531 rows=968 loops=1) Index Cond: (point <@
'(510000,510000),(500000,500000)'::box) Heap Fetches: 0 Buffers: shared hit=20 Planning time: 0.088 ms Execution time: 0.648 ms (6 ro
ws)
```

required.

Actually the comparison:

Type of request	Type index	Time ms.	Shared reads	Shared hits
100x100 -1 Point	R-tree Z-curve	0.4 * 0.34 *	1.8 1.2	3.8 3.8
1000x1000 ~ 100 points	R-tree Z-curve	0.5 ... 7 ** 0.41 *	6.2 2.8	4.9 37
10000H10000 ~ 10000 points	R-tree Z-curve	4 ... 150 *** 6.6 ****	150 43.7	27 2900

* - Data obtained by averaging the lengths of 100,000 Series
** - Data obtained by averaging a series of different lengths, 10,000 vs 100,000
*** - Data obtained by averaging a series of different lengths, 1000 vs 10,000
**** - Data obtained by averaging the length of 10 000 series

Measurements were carried out on a modest virtual machine with two cores and 4 GB of RAM, so the times do not have an absolute value, and this is the number of pages read can trust.
Times are shown in the second run, in the heated servers and virtual machines. The number of read buffers - on the freshly-lifted server.

conclusions

- at least for small queries Z-index is faster R-tree
- and reading at the same time significantly fewer pages
- R-tree begins much earlier massively overshoot cache
- and wherein the Z-index itself four times smaller, so that the cache is more efficient for him

- Moreover, it is possible that mistakes are and past the disk cache host machine, otherwise it is difficult to explain such a difference

What's next

Considered a two-dimensional dot code is only intended to test the concept, in life it is little useful.

Even it is not enough 64-bit key for three-dimensional index (or butt) for useful resolution.

So what lies ahead will be:

- switching to a different key, numeric
- 3D option including the field of point
- is it possible to work with Hilbert curve
- full measurements
- 4D version - rectangles
- 6D version - rectangles on the field
- ...

PS: The sources are laid out here with a BSD license, is described in this article corresponds to the branch of "raw-2D"

PPS: The algorithm itself was developed in ancient times (2004-2005 biennium) in collaboration with Alexander Artyushin.

PPPS: Thank you so much guys from PostgresPro because spodvigli me to implement the algorithm in PostgreSQL.

the postgresql , index the spatial , tree-r , ZOrder , spi , DBMS

26 3,5k 64

Boris Muratshin @zzeng
User

karma 91.0 rating 57.6

Related publications

48 PostgreSQL 9.4 What's New?
40,4k 130 15

8 Web Map Service is screwed to the unsuspecting OpenSource database
4,9k 36 3

37 Screwed spatial index to unsuspecting OpenSource database
14,5k 107 15

Most popular

Development

Now **Day** **A week** **Month**

30 Why I do not like synthetic benchmarks
1,6k August 16

22 ROC has launched the first Orthodox messenger
35,4k 28,169

15 Visual C ++ for Linux Development: The practice of using for Windows developers
2,2k September 30

15 Digest of fresh materials from the world for last week frontend №248 (January 30 - February 5, 2017)
7,1k 58 4

13 Electronic ink Wirenboard 5 or draw bar codes on the Go
1,1k 12 0

Comments (9)

kashey January 18, 2017 at 08:00 # 0

A possible benchmark for 32-bit keys. In principle coordinate 16 bits - enough, especially if it is used as an index Z.

zzeng January 18, 2017 at 08:05 # 0

In any case, not right now.
I think that small queries will not change anything because read single page.
The difference due to the denser packing (if in general will) occur only when the number of points in the query exceeds the number of dots significantly on the page.

zzeng January 18, 2017 at 08:07 *(Comment has been changed)* # 0

16-bit - 1000 km - the resolution of 15 meters
not enough to be, though, looking for what purposes

kashey January 18, 2017 at 08:10 # +1

16 meters of the same?
A normal 32-bit Z code describes the square on the side of 300 meters when used «world-wide». Suffice it to filter most of the data.

zzeng January 18, 2017 at 08:29 # 0

rectangle 600H300 for block index of more than

fuCtor January 18, 2017 at 08:39 # 0

As a spatial R-Tree index is good, there are still various modifications under the special cases. But if we take the problem of determining the presence of objects in the field, the Z-Index will be faster O (1) for comparison operations.
Many years ago, did rasterizer on their own vector maps ispozovanie Z-Index to trim blank areas gave a great boost. It was no matter what the scale, everything was done on a bit in the memory operations.

TSR2013 January 18, 2017 at 10:39 # 0

There was a task to implement its own database with indexing and other buns. Long chose between different implementations of spatial indexes, eventually stopped at the R * Tree (take the implementation of https://libspatialindex.github.io/). Judging from the articles on the subject R * Tree definitely wins over the standard R-Tree

zzeng January 18, 2017 at 10:40 # 0

It's time to say what wins, in what circumstances and cause links

TSR2013 January 18, 2017 at 11:01 # +1

Generally it had a great article in pdf, but it could not immediately be found. Immediately found only in the description https://en.wikipedia.org/wiki/R*_tree . Briefly, the index is a more compact by reducing the number of intersections of the tree. Initially, my task was a strong bias for reading and active reading of R * Tree unequivocally defeated R-Tree. Perhaps with a large number of inserts and updates the situation will be slightly different

Featured publication

- Sport is dead! (based on the «Agile died" and other obituaries) 4
- RAS recognized homeopathy pseudoscience 55
- What is Traffic Arbitrage? 1
- Why I do not like synthetic benchmarks 16
- The story of the designer, endearing mathematics 1