time-series encoding in Greenplum

project: pgts

https://github.com/Sasasu/pgts

Greenplum Hackathon 2022

Why we need a time series encoding

the challenge

- a lot of data
- has very low entropy
- compute by sequential scan

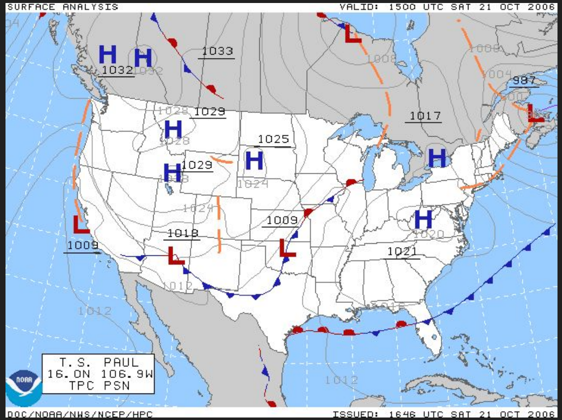
the goal

- high compression rate
- store data in time order
- need fast write speed

iot

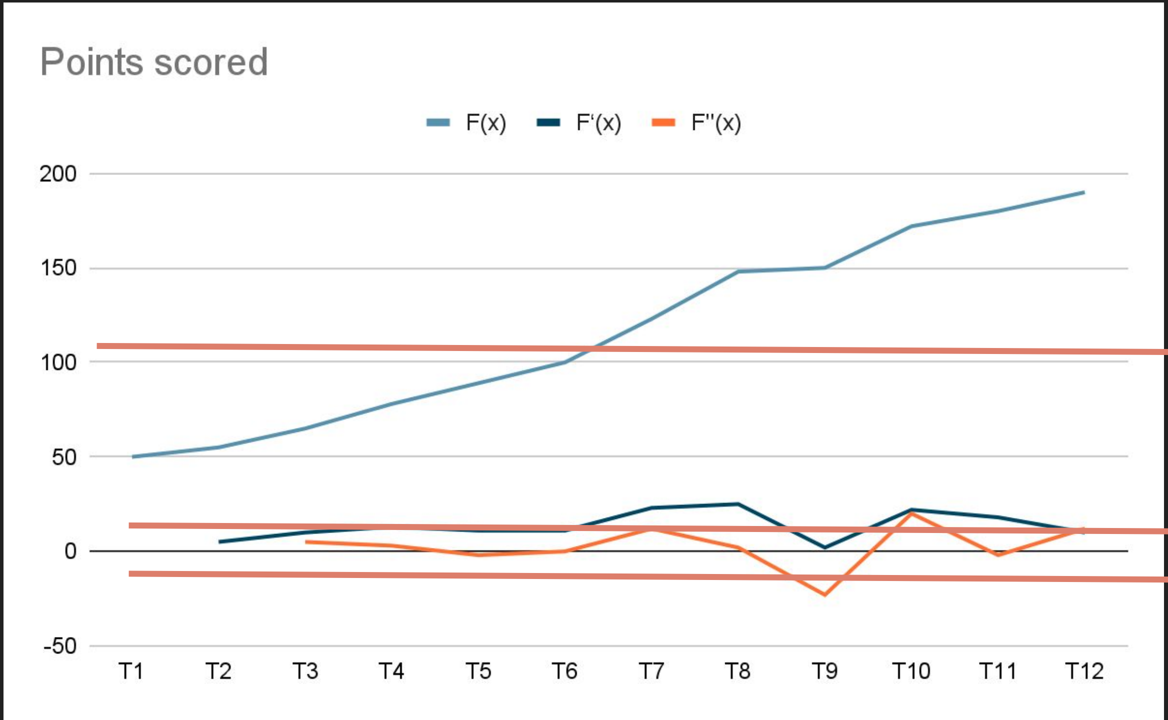
Picture from wikipedia

monitoring



gis or self-driving

How the time series encoding works



64bit precision

32bit precision

16bit precision

How the time series encoding works

But for F(x) = [1, 2, 3, 4, 5, ….] . F’(F’(x)) = [1, 0, …., 0]

If there are many ‘zero’ in the encoding output, add a ZSTD encoding with level 0

```
sa@127:postgres> select * from x;
-[ RECORD 1 ]-------------------------
hostname        | localhost
ctime           | \x28b52ffd608512b5000078218c9bc0b685fd30870200c0e1e400010073d3833a
mem_total       | \x28b52ffd60851285000048218c9b0000d2cf0700010079d38323
mem_used        | \x28b52ffda04ecf0200ec5d0ace88cafd215820d6196d3a8541bc1de238b8a223d886d6214e5664cc75b1
mem_actual_used | \x28b52ffda052cf02001c600aae8b8a0a225510605844e9b88128012e4d538e5fdba2c8c6b6a46aacf3c7
mem_actual_free | \x28b52ffda052cf020084630a4e9d0a4a2255106858e31c38e06ec1437077acc24af02bac1cb0045be946
swap_total      | \x28b52ffd6085127d000040218c9b00f07f660001007ad3032e
swap_used       | \x28b52ffd6085125d000020218c9b0001007ed30322
swap_page_in    | \x28b52ffd6085125d000020218c9b0001007ed30322
swap_page_out   | \x28b52ffd6085125d000020218c9b0001007ed30322
cpu_user        | \x28b52ffda044220200bc4b0b8e0b4d2b31431010a3b55a031c9a5e45d3b362b5d1453595da58f7327dd7
cpu_sys         | \x28b52ffda0106f01005d99097ecc10c8404a1018e4bc8b01c9695033fcc0052cf5b0af8a86e154f6158e
cpu_idle        | \x28b52ffda043660200ec3f0b7e00cdb72f3d101091670d1f29cda0685d0883cb4c40d56851963207d77c
load0           | \x28b52ffd60278b6d5b031e654679165f4012559d0788fd2dfc0ea0e4859843c3c8cd88f02a17251ae95b
load1           | \x28b52ffd60c28dd5fb014ae5755d4c204d9a3c08d8a02045099d97ca5b6c5dc60ee70896feafb162d0eb
load2           | \x28b52ffd60ed4cedff004af2542a377027691bc495f4ebfea315228720260bfd18a37d6b7eccde9e9e25
quantum         | \x28b52ffd6085126500002821 8c9b0f0001007dd3032d
disk_ro_rate    | \x28b52ffd608e12e5000088218c9b0031d779f1be0600018f1be18f0003100020d31743cd4c04
disk_wo_rate    | \x28b52ffd60c8b2419e05218c9b2b00000000000000f1ffffffffffff834265badb88b478a59a2f478c
disk_rb_rate    | \x28b52ffda018da01003d54006414218c9b007800600003d4f4c0001e4f040000f2c44c0003cb1130001e
disk_wb_rate    | \x28b52ffda09bce020064fa0d0e245dd267541020681d75217b871fd8d19217fcca69029ef1210735bbb9
net_rp_rate     | \x28b52ffd6010be81f805218c9b7801000000000000c2feffffffffffe181a44caf12db3e2fec662754
net_wp_rate     | \x28b52ffd60f8bdc1f705218c9b7601000000000000bffeffffffffffe1a9a54cef32db4630ec762955
net_rb_rate     | \x28b52ffda015cc02005cce0ccee75a23594010588fd31a98ce0cf8170504084eb63459c13ec743f4f41e
net_wb_rate     | \x28b52ffda01acc0200e4b70c8e0e9998533a1018d5a6314090094630af654110d138698a0d57cb5c77d1
cpu_iowait      | \x28b52ffd608e12cd000080218c9b002034280a0706aac42910400002006a03fa340322
SELECT 1
(END)
```

4bytes ZSTD header did not remove :(

In this case,
There are 39820 datapoints in one pgts tuple

Why using RLE is a bad idea

```
CREATE TABLE gpmetrics.gpcc_system_history (
    ctime timestamp(0) without time zone NOT NULL ENCODING (compresstype=rle_type,compresslevel=2,blocksize=
    hostname character varying(64) NOT NULL ENCODING (compresstype=rle_type,compresslevel=2,blocksize=32768)
    mem_total bigint NOT NULL ENCODING (compresstype=rle_type,compresslevel=2,blocksize=32768),
    mem_used bigint NOT NULL ENCODING (compresstype=rle_type,compresslevel=2,blocksize=32768),
    mem_actual_used bigint NOT NULL ENCODING (compresstype=rle_type,compresslevel=2,blocksize=32768),
```

ctime	0	0	2	2	3	3	4	4
hostname	foo	bar	foo	bar	foo	bar	foo	bar

- RLE can compress the ctime
- But not works on hostname and others
- Will have compression rate when there are many hostname (sizeof() > 32768)

How to use this encoding

```
create table x as
select
  hostname,
  ts.timestamp_encode( array_agg(   ctime                        order by ctime) ) as ctime,
  ts.u8_encode(        array_agg(   mem_total                    order by ctime) ) as mem_total,
  ts.u8_encode(        array_agg(   mem_used                     order by ctime) ) as mem_used,
  ts.u8_encode(        array_agg(   mem_actual_used              order by ctime) ) as mem_actual_used,
  ts.u8_encode(        array_agg(   mem_actual_free              order by ctime) ) as mem_actual_free,
  ts.u8_encode(        array_agg(   swap_total                   order by ctime) ) as swap_total,
  ts.u8_encode(        array_agg(   swap_used                    order by ctime) ) as swap_used,
  ts.u8_encode(        array_agg(   swap_page_in                 order by ctime) ) as swap_page_in,
  ts.u8_encode(        array_agg(   swap_page_out                order by ctime) ) as swap_page_out,
  ts.u8_encode(        array_agg(   (cpu_user*100000)::bigint    order by ctime) ) as cpu_user,
  ts.u8_encode(        array_agg(   (cpu_sys *100000)::bigint    order by ctime) ) as cpu_sys,
  ts.u8_encode(        array_agg(   (cpu_idle*100000)::bigint    order by ctime) ) as cpu_idle,
  ts.u8_encode(        array_agg(   (load0   *100   )::bigint    order by ctime) ) as load0,
  ts.u8_encode(        array_agg(   (load1   *100   )::bigint    order by ctime) ) as load1,
  ts.u8_encode(        array_agg(   (load2   *100   )::bigint    order by ctime) ) as load2,
  ts.u8_encode(        array_agg(   quantum                      order by ctime) ) as quantum,
  ts.u8_encode(        array_agg(   disk_ro_rate                 order by ctime) ) as disk_ro_rate,
  ts.u8_encode(        array_agg(   disk_wo_rate                 order by ctime) ) as disk_wo_rate,
  ts.u8_encode(        array_agg(   disk_rb_rate                 order by ctime) ) as disk_rb_rate,
  ts.u8_encode(        array_agg(   disk_wb_rate                 order by ctime) ) as disk_wb_rate,
  ts.u8_encode(        array_agg(   net_rp_rate                  order by ctime) ) as net_rp_rate,
  ts.u8_encode(        array_agg(   net_wp_rate                  order by ctime) ) as net_wp_rate,
  ts.u8_encode(        array_agg(   net_rb_rate                  order by ctime) ) as net_rb_rate,
  ts.u8_encode(        array_agg(   net_wb_rate                  order by ctime) ) as net_wb_rate,
  ts.u8_encode(        array_agg(  (cpu_iowait*100   )::bigint   order by ctime) ) as cpu_iowait
from gpmetrics.gpcc_system_history group by hostname;
```

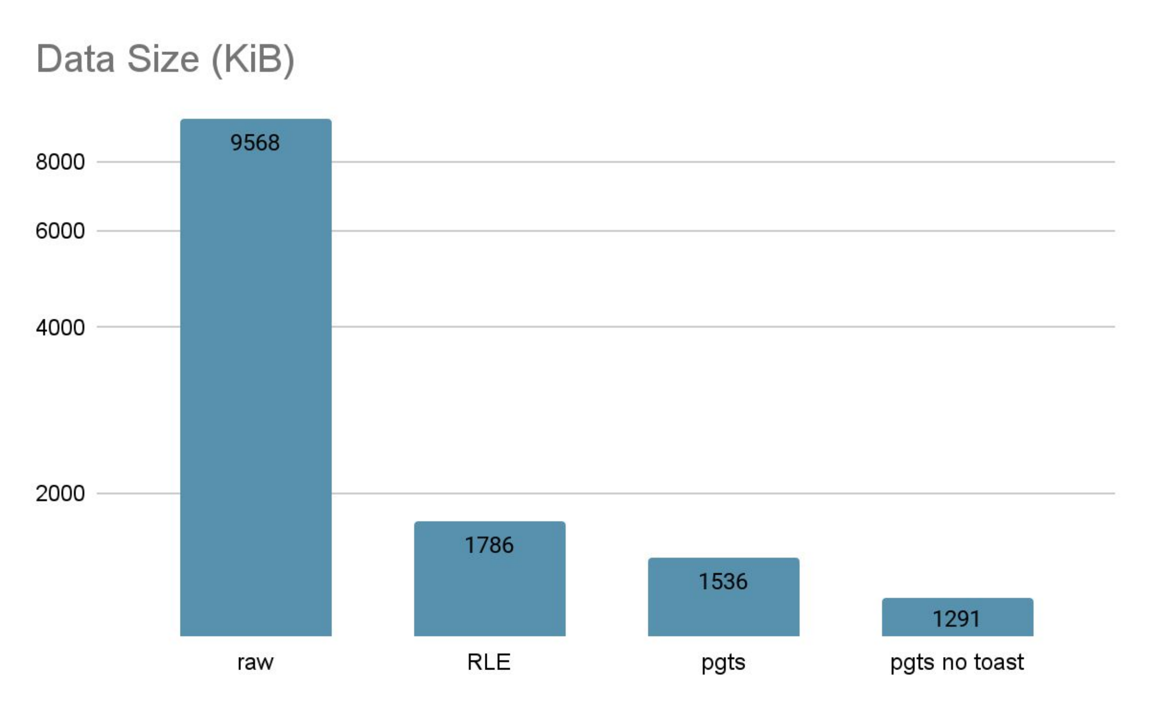
```
select
  hostname,
  unnest(ts.timestamp_decode(  ctime    )) as ctime,
  unnest(ts.u8_decode        (mem_total )) as mem_total
from
  x
group by
  hostname,
  ctime,
  mem_total
order by
  ctime;
```

query

insert

The compression rate in real word

Test data from real gpcc metric in test environment (1host * 17days)



- 7.41X smaller than raw data

- 1.38X smaller than RLE

In theory

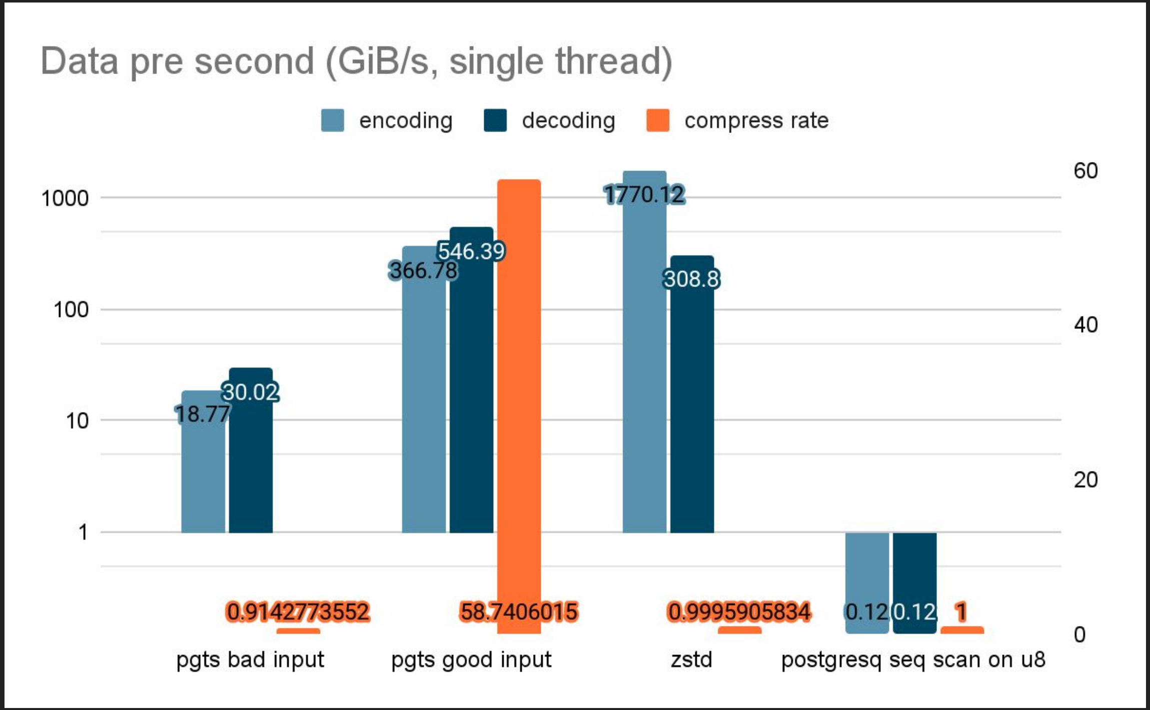
RLE can deal with small amount of hosts

pgts does not degrade compression rate when deal with a large number of hosts.

Need more test.

The performance

bad input = [rand(), rand(), …] good input = [1, 2, 3, ….]



pgts can process hundred gigabytes of data per second.

the data is compressed, and pgts can do iterator on compressed data, even apply a filter. But not implement currently.

The test data = 1GiB int64

The performance

```
sa@127:postgres> explain select
   hostname,
   unnest(ts.timestamp_decode(  ctime   )) as ctime,
   unnest(ts.u8_decode       (mem_total )) as mem_total
 from
   x
 group by
   hostname,
   ctime,
   mem_total
 order by
   ctime;
+-------------------------------------------------------------------------------------+
| QUERY PLAN                                                                          |
|-------------------------------------------------------------------------------------|
| Gather Motion 3:1  (slice1; segments: 3)  (cost=273.30..414.96 rows=10000 width=226) |
|   Merge Key: (unnest(ts.timestamp_decode(ctime)))                                   |
|   ->  Sort  (cost=273.30..281.63 rows=3333 width=226)                               |
|         Sort Key: (unnest(ts.timestamp_decode(ctime)))                              |
|         ->  ProjectSet  (cost=53.25..78.25 rows=3333 width=226)                     |
|               ->  HashAggregate  (cost=53.25..56.58 rows=333 width=210)             |
|                     Group Key: hostname, ctime, mem_total                           |
|                     ->  Seq Scan on x  (cost=0.00..45.00 rows=1100 width=210)       |
| Optimizer: Postgres query optimizer                                                 |
+-------------------------------------------------------------------------------------+
EXPLAIN
Time: 0.009s
```

Due to the UDF execute mode

pgts need a very complex SQL
using many group by and order by

The query performance using SQL is bad.

Future works

- More encoding algorithm. I didn’t implement the gorilla method for floating type
- Find a better method add this encoding type to gp or pg.
    - But data must be ordered, the data from same metric need store in the same place
- Add a memory layer to store recent data, so pgts can support realtime insert
- Support term index (gptext may enough), so can do GIS query

Thanks for listening