PostgreSQL and Compressed Documents

Aleksander Alekseev
a.alekseev@postgrespro.ru

A few words about me

- I live in Moscow, Russia;
- Develop software since 2007;
- Contribute to PostgreSQL since 2015;
- Work in Postgres Professional company;
- Interests: OSS, functional programming, electronics, SDR, distributed systems, blogging, podcasting;
- https://eax.me/ & http://devzen.ru/ ;

In this talk

- On data compression in general;
- Compressing JSONB;
- Indexing Protobuf;
- Ideas for new projects;
- Fun facts;

Fun fact!

I was informed that I’m giving this talk only yesterday.

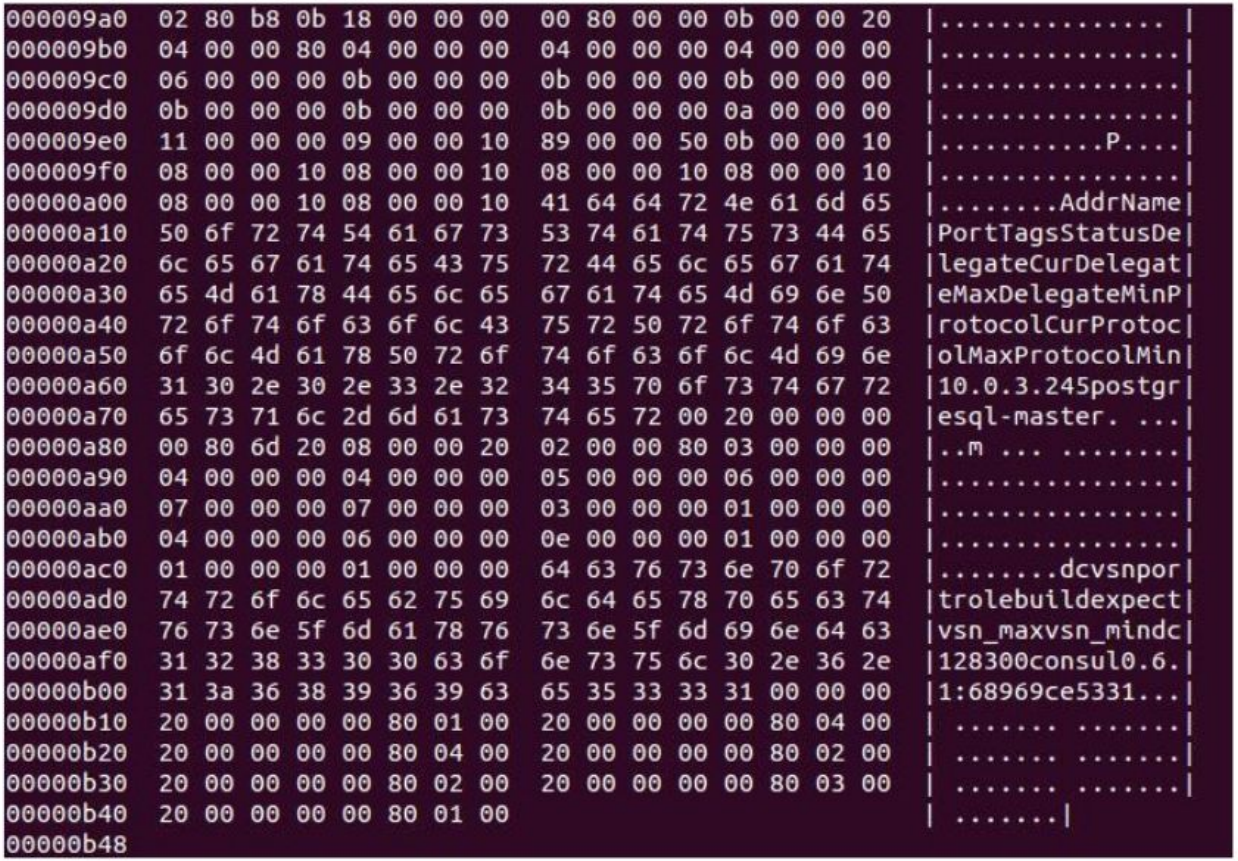
Sorry for raw slides :)

ZSON

ZSON

- An extension for transparent JSONB compression;
- A dictionary of common strings is created based on your data (re-learning is also supported);
- This dictionary is used to replace strings to 16-bit codes;
- Data is compressed in memory and on the disk;
- In some cases it gives 10% more TPS;
- Free and open source software (MIT license);
- 

How JSONB looks like



JSONB problems

- Redundancy;
- Disk space;
- Memory;
- => IO & TPS;

The idea

- Step 1: replace common strings to 16-bit codes;
- Step 2: compress using PGLZ as usual;

zson_learn

```
zson_learn(
    tables_and_columns text[][],
    max_examples int default 10000,
    min_length int default 2,
    max_length int default 128,
    min_count int default 2)
```

Example:

```
select zson_learn('{{"table1", "col1"}, {"table2", "col2"}}');
```

zson_extract_strings

```
CREATE FUNCTION zson_extract_strings(x jsonb)
    RETURNS text[] AS $$
DECLARE
    jtype text;
    jitem jsonb;
BEGIN
    jtype := jsonb_typeof(x);
    IF jtype = 'object' THEN
        RETURN array(select unnest(z) from (
                select array(select jsonb_object_keys(x)) as z
            union all (
                select zson_extract_strings(x -> k) as z from (
                    select jsonb_object_keys(x) as k
                ) as kk
            )
        ) as zz);
    ELSIF jtype = 'array' THEN
       RETURN ARRAY(select unnest(zson_extract_strings(t)) from
            (select jsonb_array_elements(x) as t) as tt);
    ELSIF jtype = 'string' THEN
        RETURN array[ x #>> array[] :: text[] ];
    ELSE -- 'number', 'boolean', 'bool'
        RETURN array[] :: text[];
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Other ZSON internals

```
CREATE FUNCTION zson_in(cstring)
    RETURNS zson
    AS 'MODULE_PATHNAME'
    LANGUAGE C STRICT IMMUTABLE;

CREATE FUNCTION zson_out(zson)
    RETURNS cstring
    AS 'MODULE_PATHNAME'
    LANGUAGE C STRICT IMMUTABLE;

CREATE TYPE zson (
    INTERNALLENGTH = -1,
    INPUT = zson_in,
    OUTPUT = zson_out,
    STORAGE = extended -- try to compress
);

CREATE FUNCTION jsonb_to_zson(jsonb)
    RETURNS zson
    AS 'MODULE_PATHNAME'
    LANGUAGE C STRICT IMMUTABLE;

CREATE FUNCTION zson_to_jsonb(zson)
    RETURNS jsonb
    AS 'MODULE_PATHNAME'
    LANGUAGE C STRICT IMMUTABLE;

CREATE CAST (jsonb AS zson) WITH FUNCTION jsonb_to_zson(jsonb) AS ASSIGNMENT;
CREATE CAST (zson AS jsonb) WITH FUNCTION zson_to_jsonb(zson) AS IMPLICIT;
```

Encoding

```
// VARHDRSZ
// zson_version [uint8]
// dict_version [uint32]
// decoded_size [uint32]
// hint [uint8 x PGLZ_HINT_SIZE]
// {
//skip_bytes [uint8]
//... skip_bytes bytes ...
//string_code [uint16], 0 = no_string
// } *
```

pg_protobuf

What it has to do with Star Wars?

Protocol Buffers

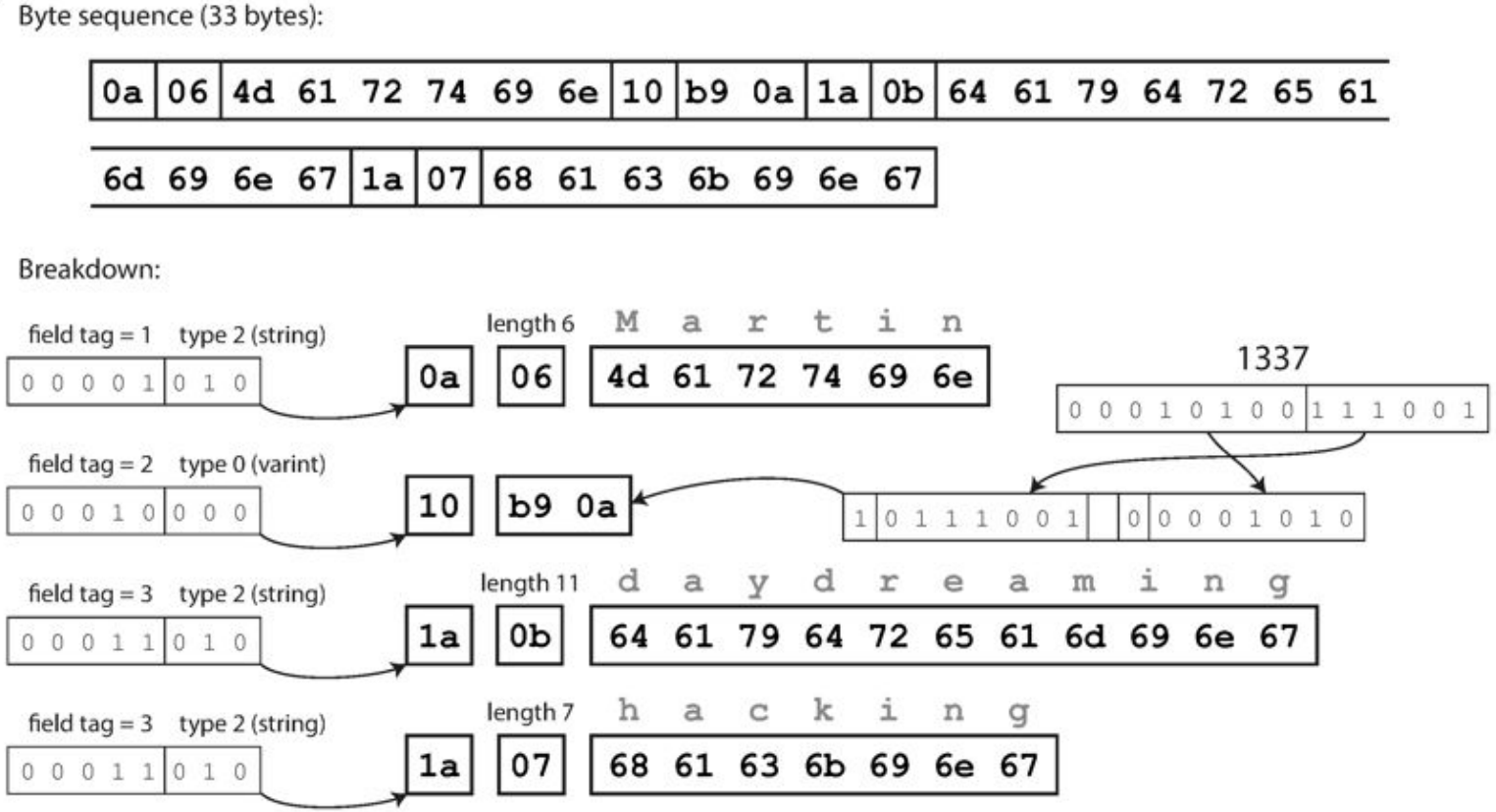
Protocol Buffers is a method of serializing structured data. It is useful in developing programs to communicate with each other over a wire or for storing data. The method involves an interface description language that describes the structure of some data and a program that generates source code from that description for generating or parsing a stream of bytes that represents the structured data.

-- Wikipedia

Person.proto

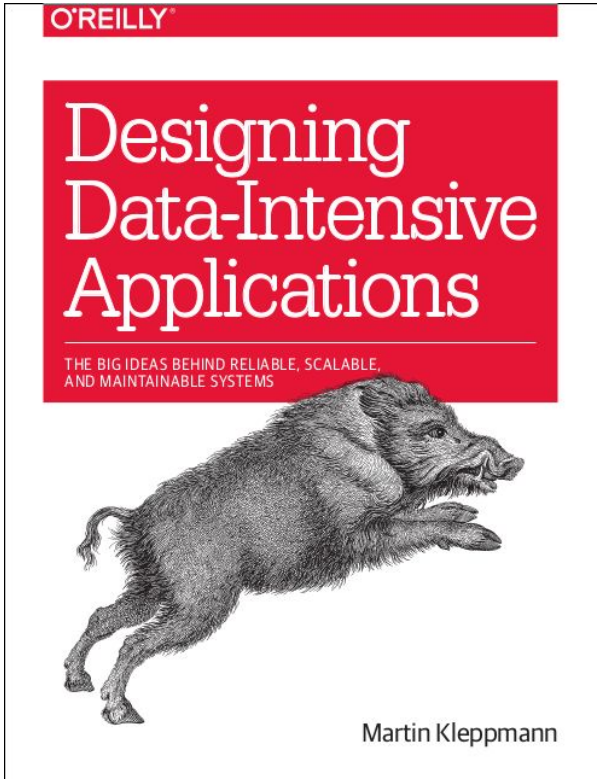
```
message Person {
    required string user_name        = 1;
    optional int64  favorite_number = 2;
    repeated string interests        = 3;
}
```

Protobuf


Byte sequence (33 bytes):
0a 06 4d 61 72 74 69 6e 10 b9 0a 1a 0b 64 61 79 64 72 65 61
6d 69 6e 67 1a 07 68 61 63 6b 69 6e 67
Breakdown:
field tag = 1 type 2 (string)
0 0 0 0 1 0 1 0
0a
length 6
06
M a r t i n
4d 61 72 74 69 6e
1337
0 0 0 1 0 1 0 0 1 1 1 0 0 1
field tag = 2 type 0 (varint)
0 0 0 1 0 0 0 0
10
b9 0a
1 0 1 1 1 0 0 1 0 0 0 0 1 0 1 0
field tag = 3 type 2 (string)
0 0 0 1 1 0 1 0
1a
length 11
0b
d a y d r e a m i n g
64 61 79 64 72 65 61 6d 69 6e 67
field tag = 3 type 2 (string)
0 0 0 1 1 0 1 0
1a
length 7
07
h a c k i n g
68 61 63 6b 69 6e 67

These two images were borrowed from

- http://shop.oreilly.com/product/0636920032175.do
- https://martin.kleppmann.com/



Fun fact!

- The attribute `required` was removed in Protobuf 3;
- All fields are optional now;

pg_protobuf

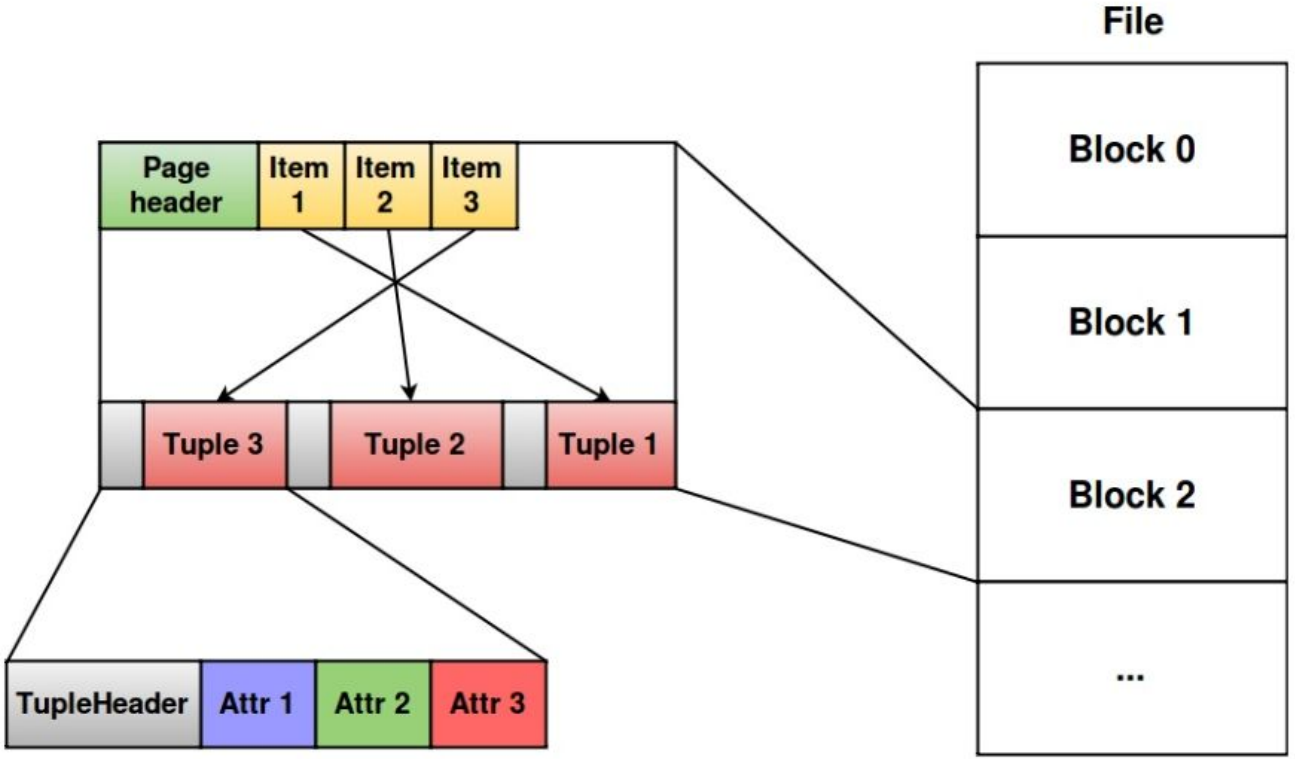
- Protobuf support for PostgreSQL;
- Like ZSON but even better;
- No shared dictionaries;
- No learning/re-learning steps;
- Requires changes in the application;
- Free and open source software (MIT license);
- 

pg_protobuf: example

```
create extension pg_protobuf;

create table heroes (x bytea);

create function hero_name(x bytea) returns text as $$
begin
return protobuf_get_string(x, 1);
end
$$ language 'plpgsql' immutable;

create index hero_name_idx on heroes using btree(hero_name(x));

select protobuf_decode(x) from heroes where hero_name(x) = 'foo';
```

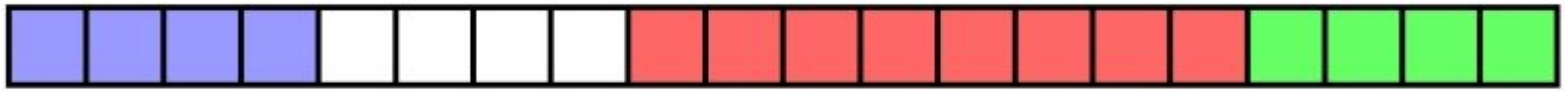
Fun facts!

Data layout

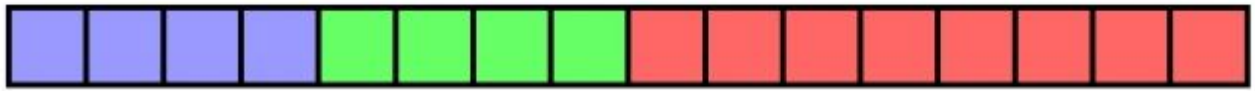

File
Block 0
Block 1
Block 2
...
Page header
Item 1
Item 2
Item 3
Tuple 3
Tuple 2
Tuple 1
TupleHeader
Attr 1
Attr 2
Attr 3

Order matters

```
create table bad (i1 int, b1 bigint, i1 int);
```



```
create table good (i1 int, i1 int, b1 bigint);
```



NULLs are free*

- Tuple header size: 23 bytes;
- With alignment: 24 bytes;
- Null mask is placed right after the header;
- Result: up to 8 nullable columns cost nothing;
- Also: buy one NULL, get 7 NULLs for free!

Alignment and B-tree

Index entries are 8-bytes aligned.

```
create table good (i1 int, i1 int, b1 bigint);

create index idx on good (i1);
```


```
create index idx_multi on good (i1, i1);
```


```
create index idx_big on good (b1);
```

Timetz vs timestamptz

- timetz: int64 (timestamp) + int32 (timezone);
- timestamptz: always an int64, UTC time;
- Result: time takes more space then date + time;

TOAST

- PGLZ: more or less same speed and ratio as ZLIB;
- Heuristic: if beginning of the attribute is compressed well, compress it;
- Works out-of-the-box for large string-like attributes;

Ideas

Ideas: ZSON

- Use PGLZ directly, don’t rely on PostgreSQL heuristics;
- Use more than one dictionary for different tables / columns;
- Same idea for TEXT / XML / whatever;

Ideas: pg_protobuf

- Add an ability to modify Protobuf data;
- Write a tool that will generate PL/pgSQL procedures for accessing Protobuf fields;
- Support unsigned types: uint, fixed32, fixed64;
- Support fields with [packet=true] attribute (in Protobuf 3 - by default);

Fun fact! There are no unsigned integer types in PostgreSQL.

Ideas: more extensions!

- pg_thrift, pg_avro, pg_capnproto, pg_messagepack, …;
- An extension with pluggable compression algorithms;

Links

- https://github.com/afiskon/zson
- https://github.com/afiskon/pg_protobuf
- https://github.com/google/protobuf
- https://eax.me/postgresql-extensions/ (in Russian)
- https://eax.me/cpp-protobuf/ (in Russian)
- https://afiskon.github.io/pgconf2017-talk.html

We are hiring!

- https://postgrespro.ru/jobs

Thank you for your attention!

- a.alekseev@postgrespro.ru
- https://afiskon.github.io/
- https://postgrespro.com/
- https://github.com/postgrespro/