In-core compression:
how to shrink your database size in several times

Aleksander Alekseev
Anastasia Lubennikova

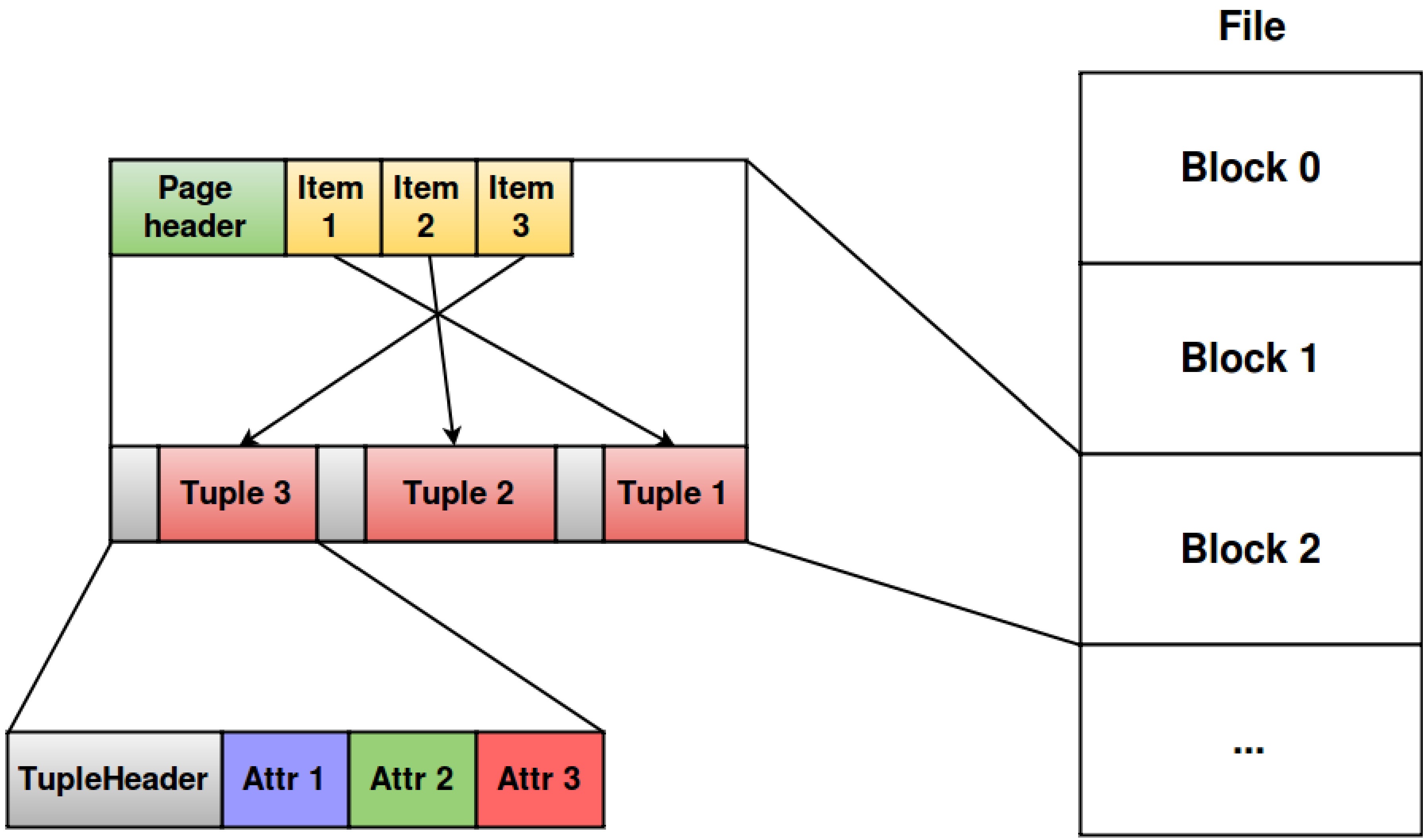
Agenda

- What does Postgres store?
  - A couple of words about storage internals
- Check list for your schema
  - A set of tricks to optimize database size
- In-core block level compression
  - Out-of-box feature of Postgres Pro EE
- ZSON
  - Extension for transparent JSONB compression

What this talk doesn't cover

- MVCC bloat
  - Tune autovacuum properly
  - Drop unused indexes
  - Use pg_repack
  - Try pg_squeeze
- Catalog bloat
  - Create less temporary tables
- WAL-log size
  - Enable wal_compression
- FS level compression
  - ZFS, btrfs, etc

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

Empty tables are not that empty

- Imagine we have no data

```
create table tbl();

insert into tbl select from generate_series(0,1e07);

select pg_size_pretty(pg_relation_size('tbl'));

pg_size_pretty
---------------
???
```

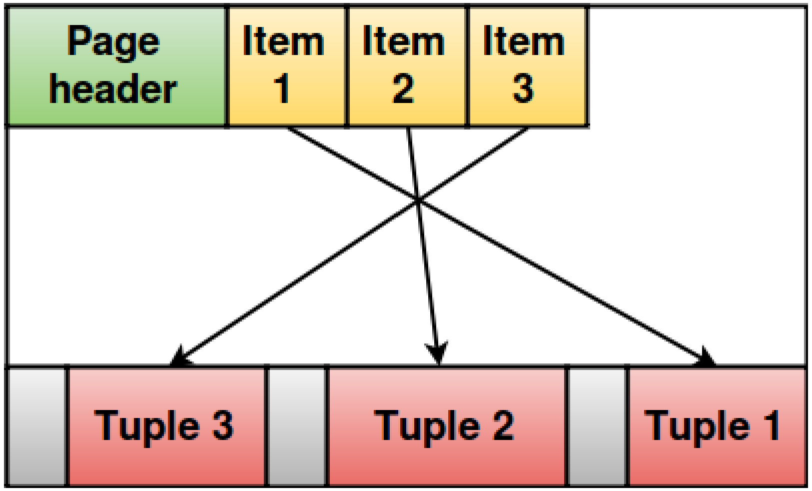
Empty tables are not that empty

- Imagine we have no data

```
create table tbl();

insert into tbl select from generate_series(0,1e07);

select pg_size_pretty(pg_relation_size('tbl'));

pg_size_pretty
---------------
268 MB
```

Meta information

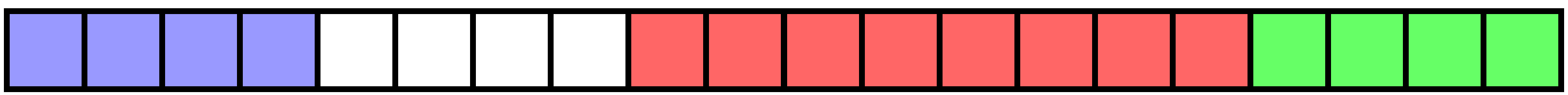
```
db=# select * from heap_page_items(get_raw_page('tbl',0));
-[ RECORD 1 ]-------------------
lp          | 1
lp_off      | 8160
lp_flags    | 1
lp_len      | 32
t_xmin      | 720
t_xmax      | 0
t_field3    | 0
t_ctid      | (0,1)
t_infomask2 | 2
t_infomask  | 2048
t_hoff      | 24
t_bits      |
t_oid       |
t_data      |
```



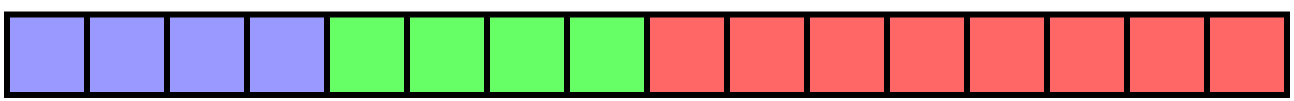
Order matters

- Attributes must be aligned inside the row

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



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



Safe up to **20%** of space.

NULLs for free*

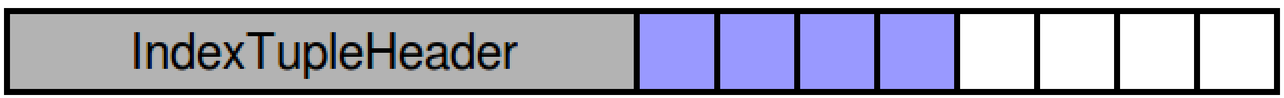
- Tuple header size: 23 bytes
- With alignment: 24 bytes
- Null mask is placed right after a header
- Result: up to 8 nullable columns cost nothing
- Also: buy one NULL, get 7 NULLs for free! (plus alignment)

* not actually free

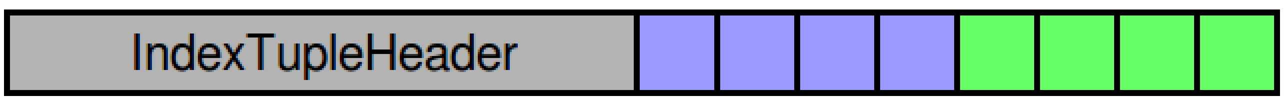
Alignment and B-tree

All index entries are 8 bytes aligned

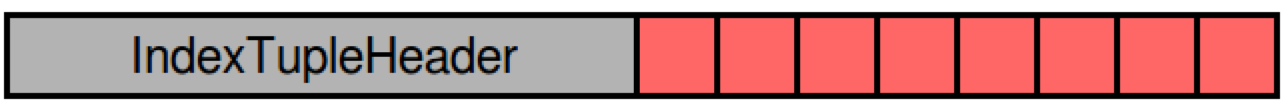
```
create table good (i1 int, i1 int, b1 bigint);

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




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




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



Alignment and B-tree

- It cannot be smaller, but it can keep more data
- Covering indexes* may come in handy here
  - CREATE INDEX tbl_pkey (i1) INCLUDE (i2)

- + It enables index-only scan for READ queries
- – It disables HOT updates for WRITE queries

*Already in PostgresPro, hopefully will be in PostgreSQL 10

Use proper data types

```
CREATE TABLE b AS
SELECT 'a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11'::bytea;

select lp_len, t_data from heap_page_items(get_raw_page('b',0));

lp_len | t_data
-------+-------------------------------------------------------------
    61 |
\x4b6130656562633939392d396330622d346566382d626236642d366262396264
33830613131
```

```
CREATE TABLE u AS
SELECT 'a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11'::uuid;

select lp_len, t_data from heap_page_items(get_raw_page('u',0));

lp_len | t_data
-------+------------------------------------------
    40 | \xa0eebc999c0b4ef8bb6d6bb9bd380a11
```

Timetz vs timestamptz

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

TOAST

- Splitting of oversized attributes with an optional compression
  - PGLZ: more or less same (speed, ratio) as ZLIB
  - Heuristic: if beginning of the attribute is compressed well then compress it
  - Works out of the box for large string-like attributes

Know your data and your database

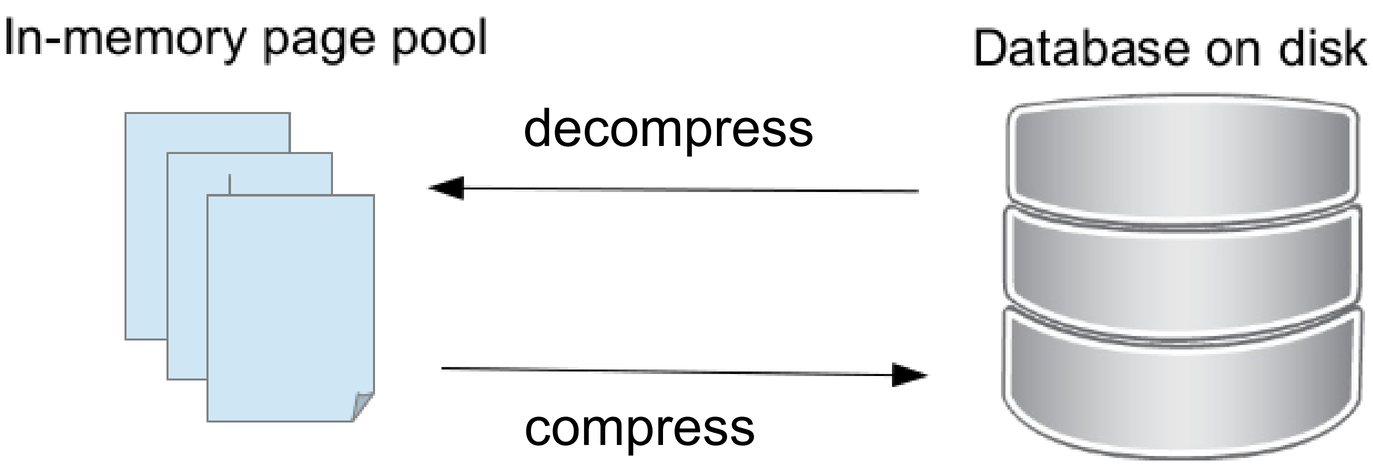
- Use proper data types
- Reorder columns to avoid padding
- Pack data into bigger chunks to trigger TOAST

Know your data and your database

- Use proper data types
- Reorder columns to avoid padding
- Pack data into bigger chunks to trigger TOAST

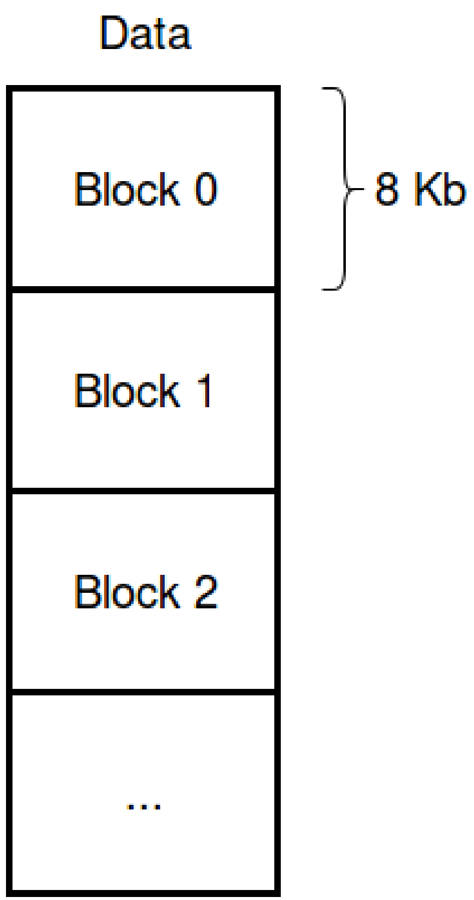
CFS

- CFS — «compressed file system»
  - Out of box (PostgresPro Enterprise Edition)

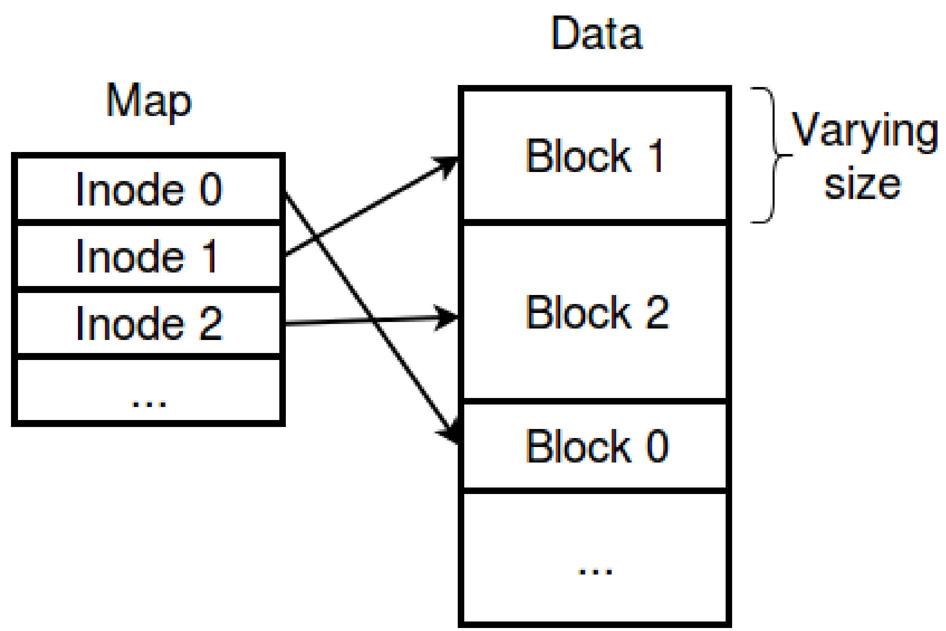


Layout changes

- Postgres layout



- CFS layout



CFS usage

```
CREATE TABLESPACE cfs LOCATION
'/home/tblspc/cfs' with (compression=true);

SET default_tablespace=cfs;

CREATE TABLE tbl (x int);

INSERT INTO tbl VALUES (generate_series(1, 1000000));

UPDATE tbl set x=x+1;

SELECT cfs_start_gc(4); /* 4 – number of workers */
```

Pgbench performance

- pgbench -s 1000 -i
  - **2 times slower**
  - 98 sec → 214 sec
- database size
  - **18 times smaller**
  - 15334 MB → 827 MB
- pgbench -c 10 -j 10 -t 10000
  - **5% better**
  - 3904 TPS → 4126 TPS

Always doubt benchmarks

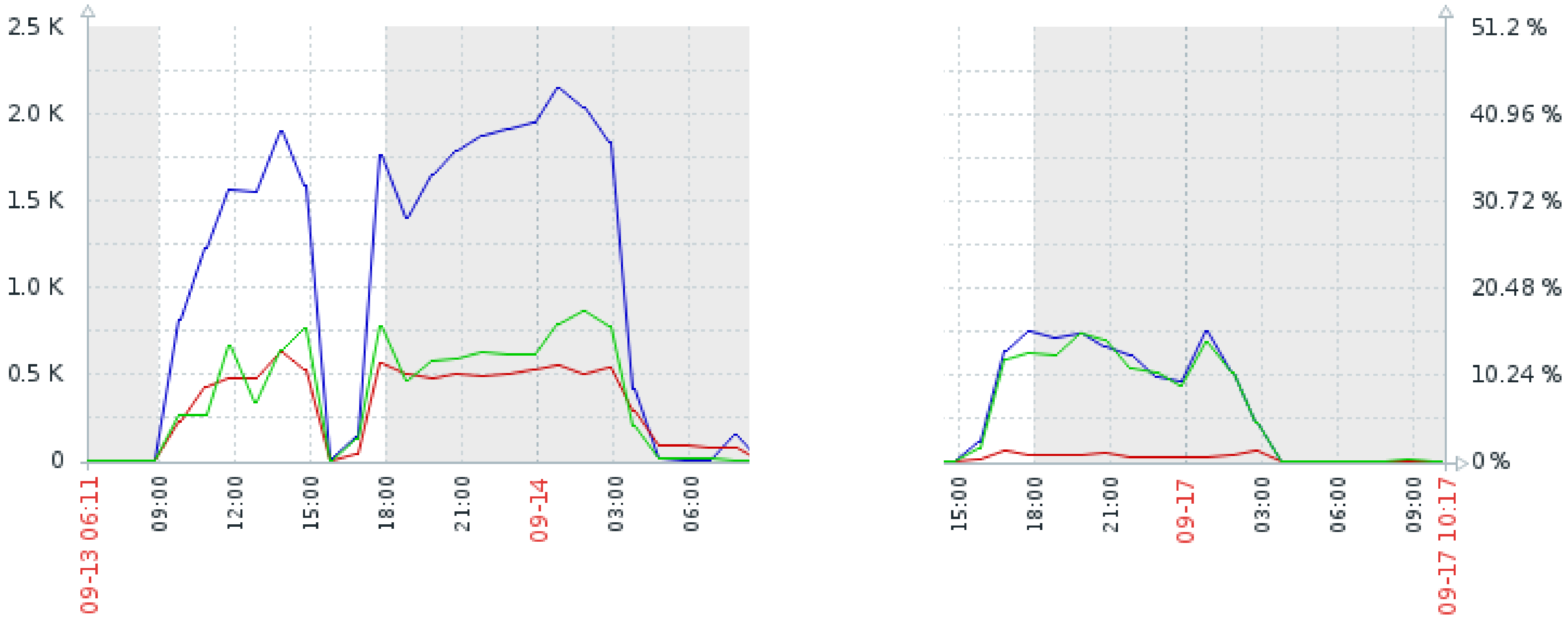
```
db=# select * from pgbench_accounts;
-[ RECORD1 ]------------------------------------
aid      | 1
bid      | 1
abalance | 0
filler   |

db=# \d pgbench_accounts
              Table "public.pgbench_accounts"
  Column  |      Type       | Collation | Nullable | Default
----------+-----------------+-----------+----------+---------
 aid      | integer         |           | not null |
 bid      | integer         |           |          |
 abalance | integer         |           |          |
 filler   | character(84)   |           |          |


db=# select pg_column_size(filler) from pgbench_accounts;
 pg_column_size
----------------
             85
```

Comparison of compression algoritms

pgbench -i -s 1000

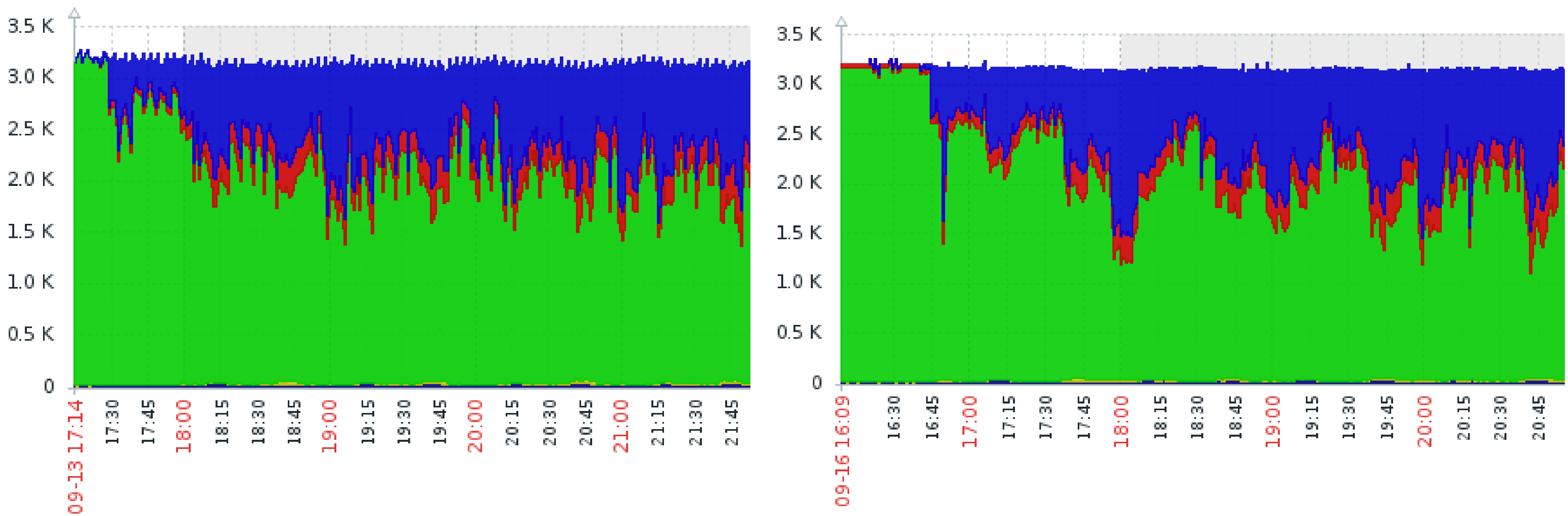
Configuration	Size (Gb)	Time (sec)
no compression	15.31	92
snappy	5.18	99
lz4	4.12	91
postgres internal lz	3.89	214
lzfse	2.80	1099
zlib (best speed)	2.43	191
zlib (default level)	2.37	284
zstd	**1.69**	**125**

CFS: I/O usage



2.5 K
2.0 K
1.5 K
1.0 K
0.5 K
0
09-13 06:11
09:00
12:00
15:00
18:00
21:00
09-14
03:00
06:00
51.2 %
40.96 %
30.72 %
20.48 %
10.24 %
0 %
15:00
18:00
21:00
09-17
03:00
06:00
09:00
09-17 10:17

CPU usage



- CPU time spent by normal programs and daemons
- CPU time spent by nice(1)d programs
- CPU time spent by the kernel in system activities
- CPU time spent Idle CPU time
- CPU time spent waiting for I/O operations
- CPU time spent handling interrupts
- CPU time spent handling batched interrupts

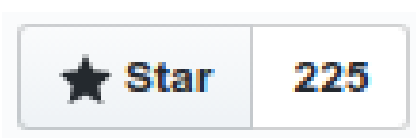
CFS pros

- Good compression rate:
  - All information on the page is compressed including headers
- Better locality:
  - CFS always writes new pages sequentially
- Minimal changes in Postgres core:
  - CFS works at the lowest level
- Flexibility:
  - Easy to use various compression algorithms

CFS cons

- Shared buffers utilization:
  - Buffer cache keeps pages uncompressed
- Inefficient WAL and replication:
  - Replica has to perform compression and GC itself
- Fragmentation
  - CFS needs its own garbage collector

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
- 

- https://github.com/postgrespro/zson

How JSONB looks like

```
000009a0  02 80 b8 0b 18 00 00 00  00 80 00 00 0b 00 00 20  |............... |
000009b0  04 00 00 80 04 00 00 00  04 00 00 00 04 00 00 00  |................|
000009c0  06 00 00 00 0b 00 00 00  0b 00 00 00 0b 00 00 00  |................|
000009d0  0b 00 00 00 0b 00 00 00  0b 00 00 00 0a 00 00 00  |................|
000009e0  11 00 00 00 09 00 00 10  89 00 00 50 0b 00 00 10  |...........P....|
000009f0  08 00 00 10 08 00 00 10  08 00 00 10 08 00 00 10  |................|
00000a00  08 00 00 10 08 00 00 10  41 64 64 72 4e 61 6d 65  |........AddrName|
00000a10  50 6f 72 74 54 61 67 73  53 74 61 74 75 73 44 65  |PortTagsStatusDe|
00000a20  6c 65 67 61 74 65 43 75  72 44 65 6c 65 67 61 74  |legateCurDelegat|
00000a30  65 4d 61 78 44 65 6c 65  67 61 74 65 4d 69 6e 50  |eMaxDelegateMinP|
00000a40  72 6f 74 6f 63 6f 6c 43  75 72 50 72 6f 74 6f 63  |rotocolCurProtoc|
00000a50  6f 6c 4d 61 78 50 72 6f  74 6f 63 6f 6c 4d 69 6e  |olMaxProtocolMin|
00000a60  31 30 2e 30 2e 33 2e 32  34 35 70 6f 73 74 67 72  |10.0.3.245postgr|
00000a70  65 73 71 6c 2d 6d 61 73  74 65 72 00 20 00 00 00  |esql-master. ...|
00000a80  00 80 6d 20 08 00 00 20  02 00 00 80 03 00 00 00  |..m ... ........|
00000a90  04 00 00 00 04 00 00 00  05 00 00 00 06 00 00 00  |................|
00000aa0  07 00 00 00 07 00 00 00  03 00 00 00 01 00 00 00  |................|
00000ab0  04 00 00 00 06 00 00 00  0e 00 00 00 01 00 00 00  |................|
00000ac0  01 00 00 00 01 00 00 00  64 63 76 73 6e 70 6f 72  |........dcvsnpor|
00000ad0  74 72 6f 6c 65 62 75 69  6c 64 65 78 70 65 63 74  |trolebuildexpect|
00000ae0  76 73 6e 5f 6d 61 78 76  73 6e 5f 6d 69 6e 64 63  |vsn_maxvsn_mindc|
00000af0  31 32 38 33 30 30 63 6f  6e 73 75 6c 30 2e 36 2e  |128300consul0.6.|
00000b00  31 3a 36 38 39 36 39 63  65 35 33 33 31 00 00 00  |1:68969ce5331...|
00000b10  20 00 00 00 00 80 01 00  20 00 00 00 00 80 04 00  | ....... .......|
00000b20  20 00 00 00 00 80 04 00  20 00 00 00 00 80 02 00  | ....... .......|
00000b30  20 00 00 00 00 80 02 00  20 00 00 00 00 80 03 00  | ....... .......|
00000b40  20 00 00 00 00 80 01 00                           | .......|
00000b48
```

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
// } *
```

Thank you for your attention!
Any questions?

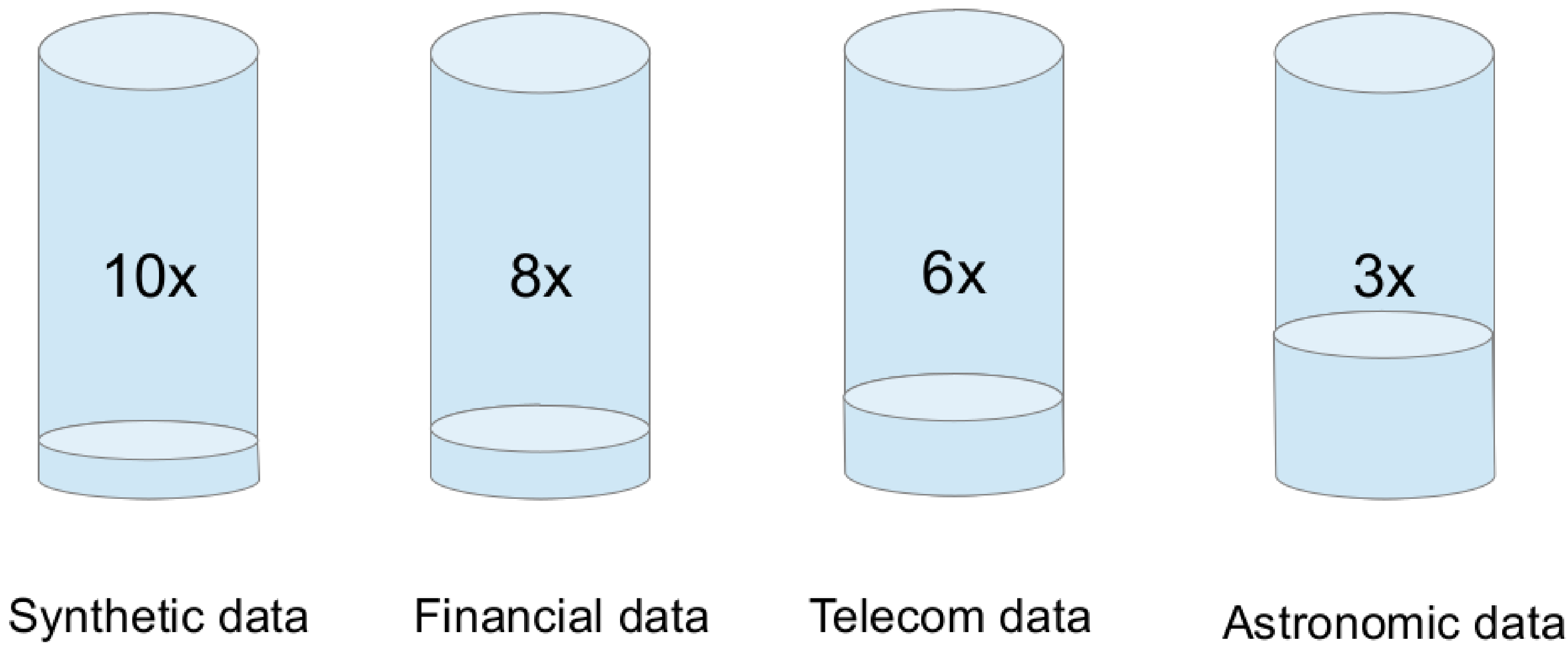
- https://postgrespro.com/
- a.lubennikova@postgrespro.ru
- a.alekseev@postgrespro.ru

Bonus Slides!

CFS parameters

- **cfs_gc_workers = 1**
  - Number of background workers performing CFS garbage collection
- **cfs_gc_threashold = 50%**
  - Percent of garbage in the file after which defragmentation begins
- **cfs_gc_period = 5 seconds**
  - Interval between CFS garbage collection iterations
- **cfs_gc_delay = 0 milliseconds**
  - Delay between files defragmentation

CFS: Compression ratio



In-memory dictionary

```
#define DICT_MAX_WORDS (1 << 16)

typedef struct {
    uint16 code;
    bool check_next; // next word starts with the same nbytes bytes
    size_t nbytes; // number of bytes (not letters) except \0
    char* word;
} Word;

typedef struct {
    int32 dict_id;
    uint32 nwords;
    Word words[DICT_MAX_WORDS]; // sorted by .word, word -> code
    uint16 code_to_word[DICT_MAX_WORDS]; // code -> word index
} Dict;

typedef struct DictListItem {
    Dict* pdict;
    union {
        time_t last_clean_sec; // for first list item
        time_t last_used_sec; // for rest list items
    };
    struct DictListItem* next;
} DictListItem;
```