Delta Frame of Reference Compression. Custom development and its usage for PostgreSQL

Evgeny Voropaev

About Reporter

Evgeny Voropaev,
Ph. D. in Physico-Mathematical Sciences,
PostgreSQL developer

Hobbies: mountaineering, electronics, algorithms

My Work with PostgreSQL

Refactor some repetitive SLRU code (REL_19_STABLE)

Compress prune/freeze records with Delta Frame of Reference algorithm (PGSQL-HACKERS)

64-bit transaction identifier (PGSQL-HACKERS)

Abstracts

The Delta Frame of Reference algorithm (DFOR) is a widely used solution for the lossless compression of integer sequences. DFOR, which is classified as differential compression method, provides a high compression ratio even for short data sequences, where statistical and dictionary-based compression methods can be ineffective. While nowadays the plenty of DFOR implementations are available, custom tailoring and further development of the algorithm are often required when we are eager to achieve a greater compression ratio, especially when dealing with data specificity, original use cases, or the unique environment of an idiosyncratic system like PostgreSQL.

This article presents the core concepts of DFOR alongside the author's original enhancements to the standard algorithm. The current implementation already supports both sorted and unsorted sequences, provides positive and negative deltas, utilizes exceptions to store long deltas separately, thereby increasing 'pack density'. However, a few features that could improve the algorithm further are still awaiting implementation.

The paper also describes both completed and planned applications of DFOR within PostgreSQL. Regarding accomplished milestones, DFOR has already been integrated into WAL prune/freeze records to compress offset sequences, resulting in a practical compression ratio of more than 5. The next integration step is applying DFOR to GIN-index, where compression of tuple identifier sequences looks promising and could theoretically achieve compression ratio of 16.

The code architecture of the DFOR unit provides header templates comprising macros and function patterns. This allows developers to easily create an instantiation of the DFOR supporting an integer item of any size that is a multiple of 8 bits. The bit-packing utility, heavily utilised by DFOR as a workhorse, is implemented as a distinct unit and can be used by PostgreSQL independently.

In conclusion, the presented DFOR solution is now largely implemented, and its application within PostgreSQL already demonstrates significant compression levels. Future work in this realm will focus on improving the algorithm's compression ratio and expanding coverage to other areas of the PostgreSQL codebase where the compression of integer sequences is profitable.

DFOR in action

Input data: the sequence of integers

We have the sequence of integers.

DFOR in action

Sequence	uint16
1	0x0001
2	0x0002
3	0x0003
130	0x0082
257	0x0101
258	0x0102

2 byte * 6 =
96 bits

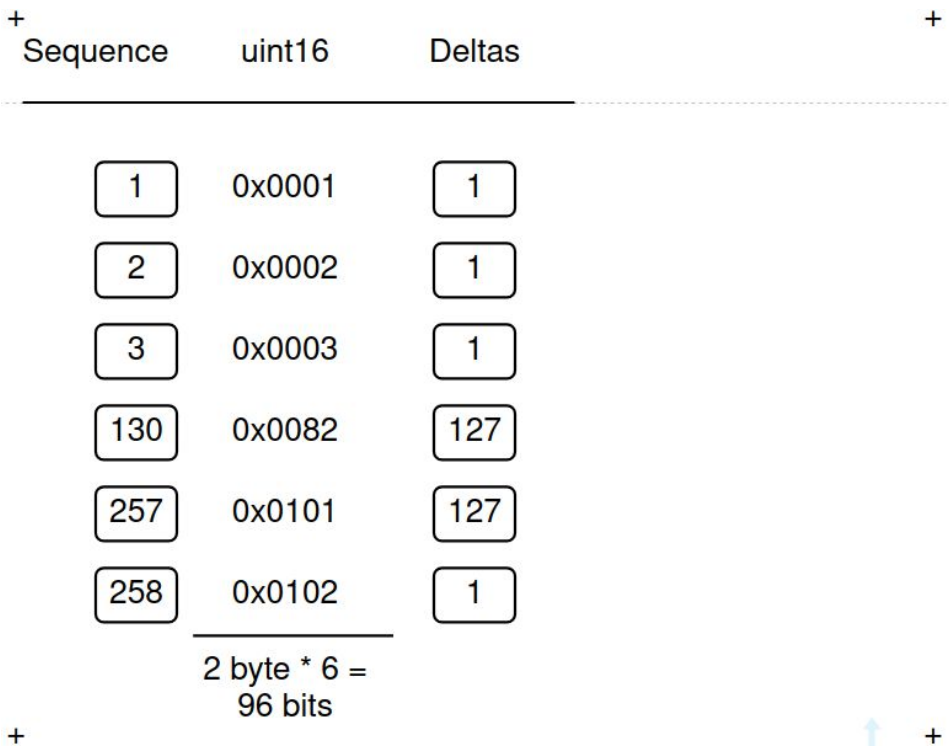
We have the sequence of integers.
By default the sequences can be presented in PC as an array of 2-byte words. This results in 96-bits pack.

Let’s try to compress it.

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

DFOR in action



Input data: the sequence of integers

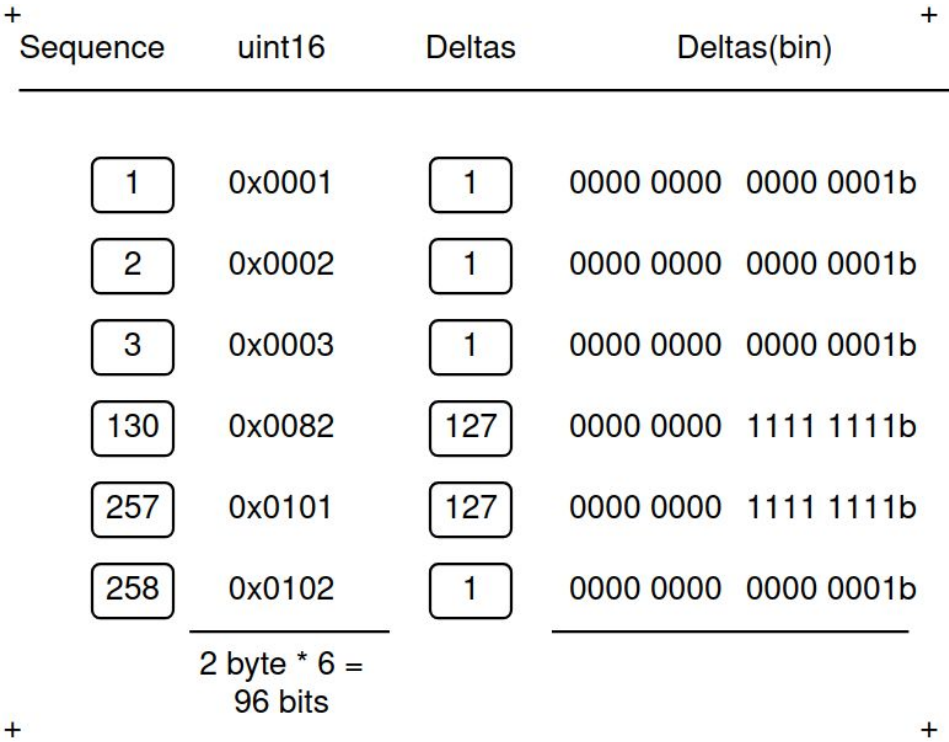
Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

Step 1. Calculate deltas

We have the sequence of integers.
By default the sequences can be presented in PC as an array of 2-byte words. This results in 96-bits pack.

Let's try to compress it. First of all, let's take deltas.
Their values are lesser than original items.

DFOR in action



Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

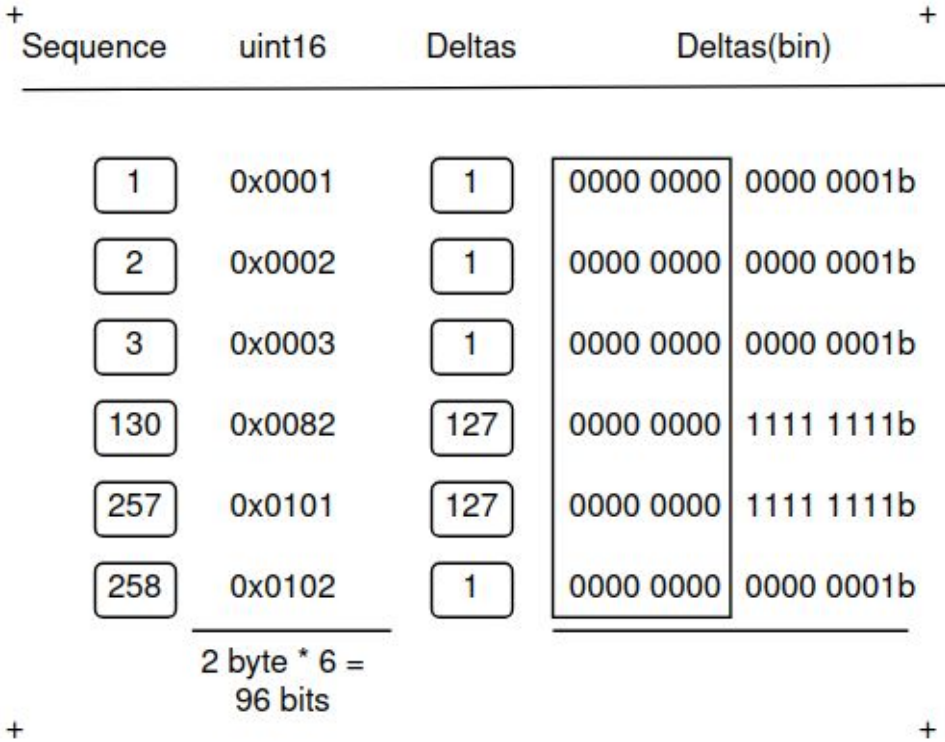
Step 1. Calculate deltas.

We have the sequence of integers.
By default the sequences can be presented in PC as an array of 2-byte words. This results in 96-bits pack.

Let's try to compress it. First of all, let's take deltas. Their values are lesser than original items.

Scrutinising the binary representation we can see a lot of leading zeroes.

DFOR in action



Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

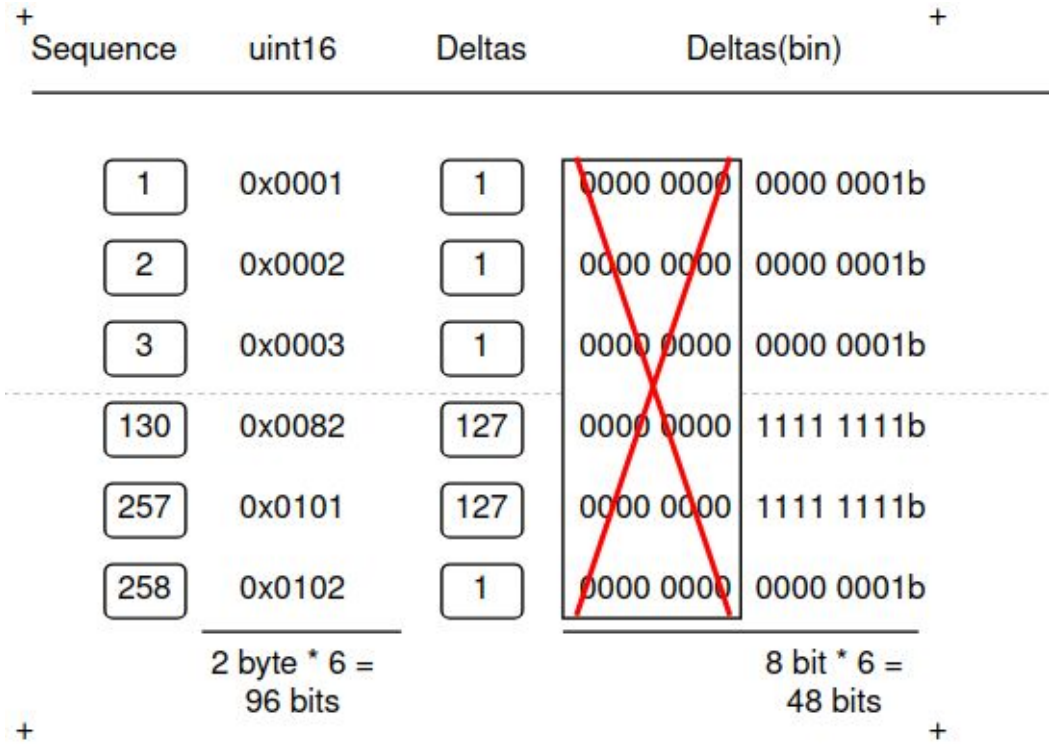
Step 1. Calculate deltas.

We have the sequence of integers.
By default the sequences can be presented in PC as an array of 2-byte words. This results in 96-bits pack.

Let's try to compress it. First of all, let's take deltas. Their values are lesser than original items.

Scrutinising the binary representation we can see a lot of leading zeros. Remove them, since they does not bring any information.

DFOR in action



Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

Step 1. Calculate deltas.

Step 2. Remove leading zeros.

We have the sequence of integers.
By default the sequences can be presented in PC as an array of 2-byte words. This results in 96-bits pack.

Let's try to compress it. First of all, let's take deltas. Their values are lesser than original items.

Scrutinising the binary representation we can see a lot of leading zeros. Remove them, since they does not bring any information. The pack is 48-bit size now.

DFOR in action

Sequence	uint16	Deltas	Deltas(bin)
1	0x0001	1	0000 0001b
2	0x0002	1	0000 0001b
3	0x0003	1	0000 0001b
130	0x0082	127	1111 1111b
257	0x0101	127	1111 1111b
258	0x0102	1	0000 0001b
	2 byte * 6 = 96 bits		8 bit * 6 = 48 bits

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

Step 1. Calculate deltas.

Step 2. Remove leading zeros.

We have the sequence of integers.
By default the sequences can be presented in PC as an array of 2-byte words. This results in 96-bits pack.

Let's try to compress it. First of all, let's take deltas. Their values are lesser than original items.

Scrutinising the binary representation we can see a lot of leading zeros. Remove them, since they does not bring any information. The pack is 48-bit size now.

DFOR in action

Sequence	uint16	Deltas	Deltas(bin)
1	0x0001	1	0000 000 1b
2	0x0002	1	0000 000 1b
3	0x0003	1	0000 000 1b
130	0x0082	127	1111 111 1b
257	0x0101	127	1111 111 1b
258	0x0102	1	0000 000 1b
	2 byte * 6 = 96 bits		

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

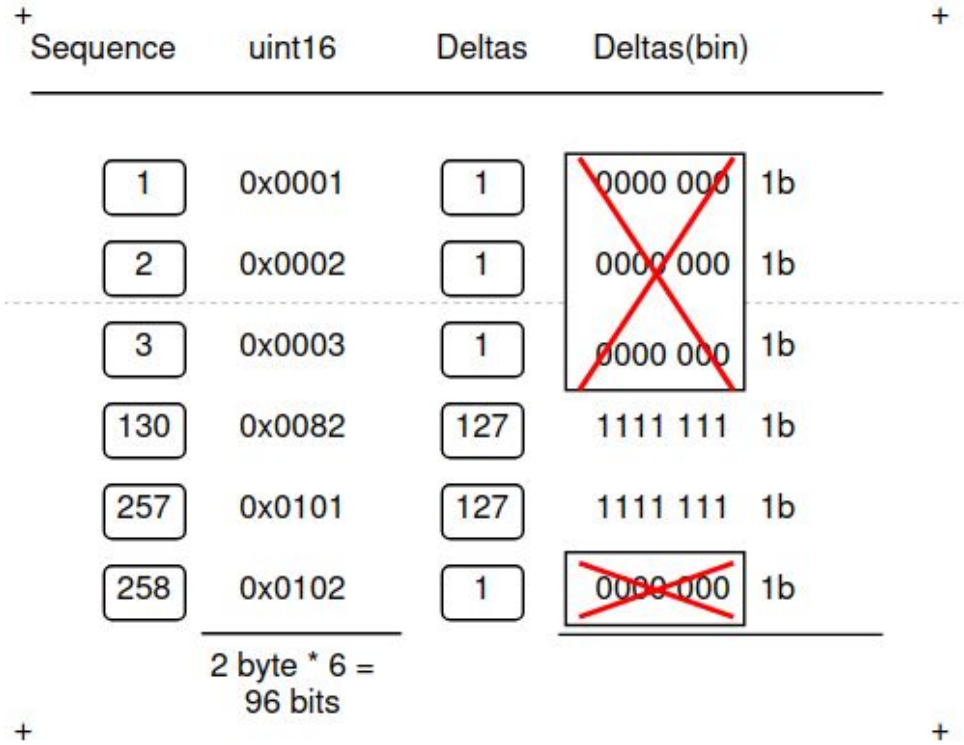
Step 1. Calculate deltas.

Step 2. Remove leading zeros.

Let's go further. We see that for some deltas the width of 1 bit is quite enough and these deltas still have leading zeros.

We separate each delta into two parts: lower, having 1-bit width, and higher, having 7-bit width.

DFOR in action



Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

Step 1. Calculate deltas.

Step 2. Remove leading zeros.

Let's go further. We see that for some deltas the width of 1 bit is quite enough and these deltas still have leading zeros.

We separate each delta into two parts: lower, having 1-bit width, and higher, having 7-bit width.
Remove insignificant leading zeros from deltas.

DFOR in action

Sequence	uint16	Deltas	Deltas(bin)	
			Exception	Delta
1	0x0001	1		1b
2	0x0002	1		1b
3	0x0003	1		1b
130	0x0082	127	1111 111	1b
257	0x0101	127	1111 111	1b
258	0x0102	1		1b
	2 byte * 6 = 96 bits			

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

Step 1. Calculate deltas.

Step 2. Remove leading zeros.

Step 3. Withdraw exceptions.

Let's go further. We see that for some deltas the width of 1 bit is quite enough and these deltas still have leading zeros.

We separate each delta into two parts: lower, having 1-bit width, and higher, having 7-bit width.
Remove insignificant leading zeros from deltas.

Remained 'Long' deltas are exceptions.

DFOR in action

Sequence	uint16	Deltas	Deltas(bin)			
			Exception	Delta	Pos	PosDelta
1	0x0001	1		1b	0	
2	0x0002	1		1b	1	
3	0x0003	1		1b	2	
130	0x0082	127	1111 111	1b	3	11b
257	0x0101	127	1111 111	1b	4	01b
258	0x0102	1		1b	5	

2 byte * 6 = 96 bits

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

Step 1. Calculate deltas.

Step 2. Remove leading zeros.

Step 3. Withdraw exceptions.

Let's go further. We see that for some deltas the width of 1 bit is quite enough and these deltas still have leading zeros.

We separate each delta into two parts: lower, having 1-bit width, and higher, having 7-bit width. Remove insignificant leading zeros from deltas.

The remained 'Long' deltas are exceptions. The algorithm defines their positions and also codes them using the delta technique.

DFOR in action

Sequence	uint16	Deltas	Deltas(bin)	Exception	Delta	Pos	PosDelta
1	0x0001	1			1b	0	
2	0x0002	1			1b	1	
3	0x0003	1			1b	2	
130	0x0082	127		1111 111	1b	3	11b
257	0x0101	127		1111 111	1b	4	01b
258	0x0102	1			1b	5	
	2 byte * 6 = 96 bits						

Pack: 111111 1101 1111111 1111111

24 bits Deltas PosΔ Exceptions

Pack deltas, exceptions and position deltas bit-by-bit.

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

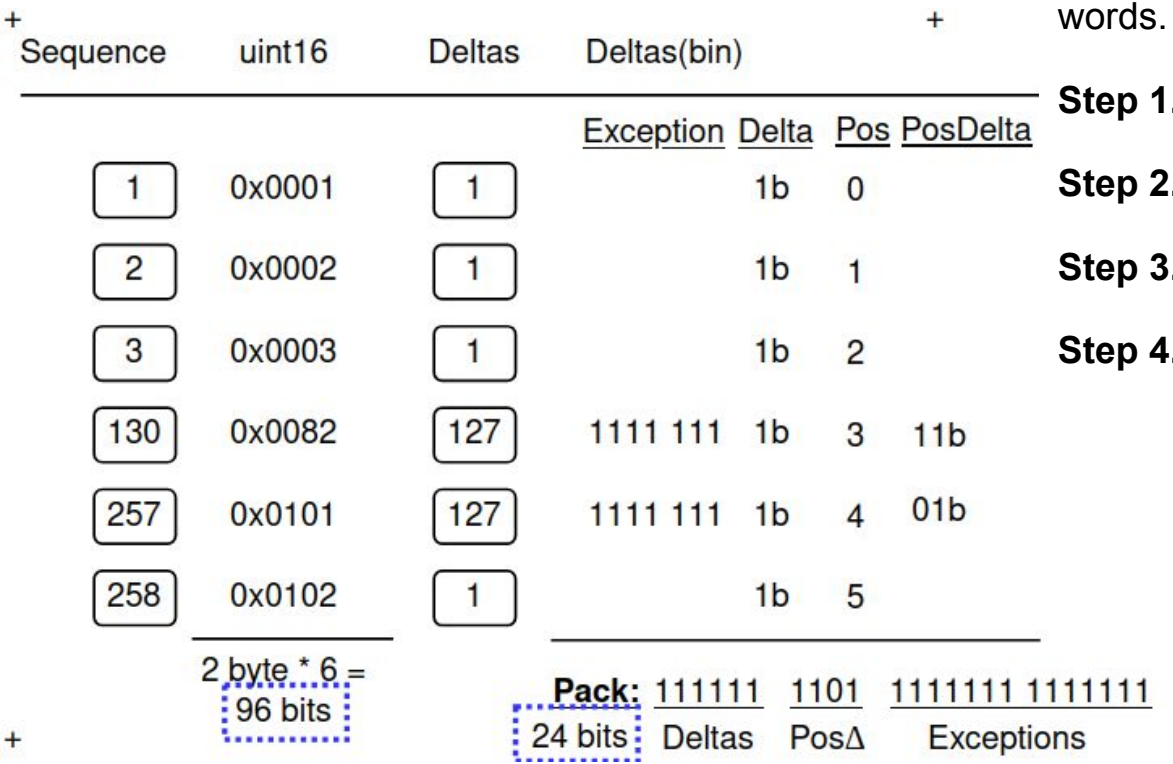
Step 1. Calculate deltas.

Step 2. Remove leading zeros.

Step 3. Withdraw exceptions.

Step 4. Pack everything bit-by-bit.

DFOR in action



Pack deltas, exceptions and position deltas bit-by-bit.

The pack is 24-bit long.
The compression ratio on this data example is 96/24=4

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

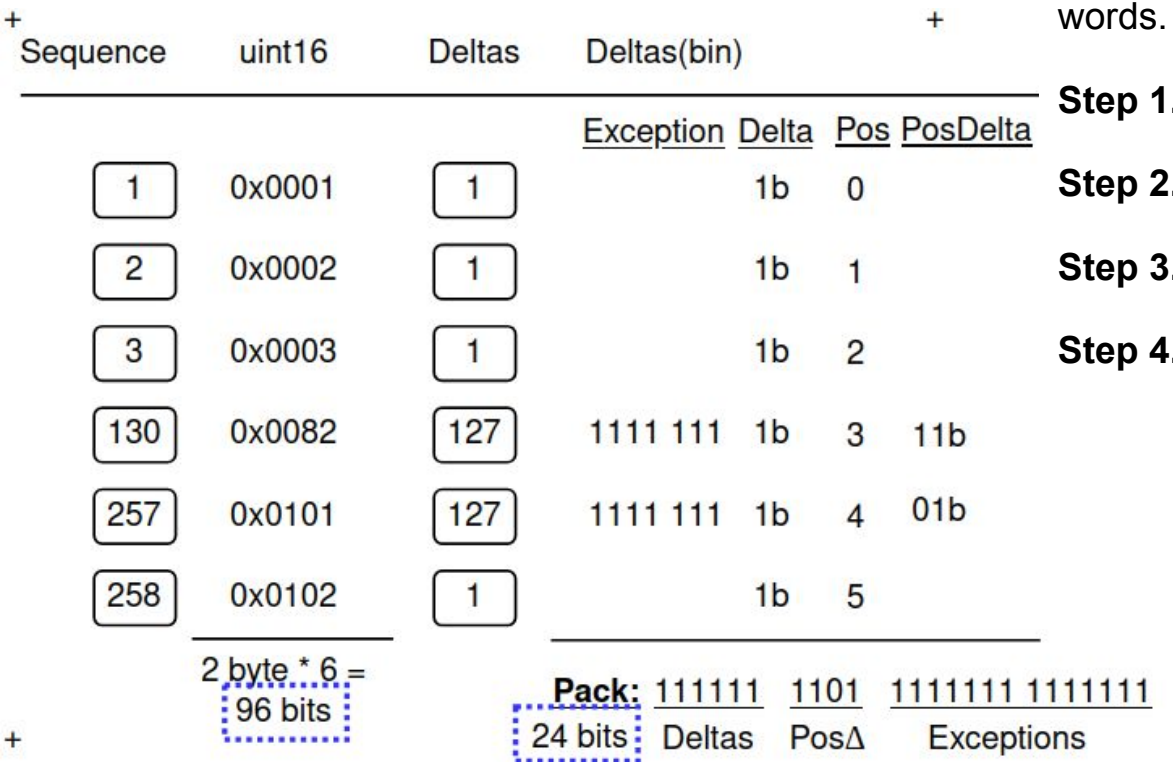
Step 1. Calculate deltas.

Step 2. Remove leading zeros.

Step 3. Withdraw exceptions.

Step 4. Pack everything bit-by-bit.

DFOR in action



Sequence	uint16	Deltas	Deltas(bin)			
			Exception	Delta	Pos	PosDelta
1	0x0001	1		1b	0	
2	0x0002	1		1b	1	
3	0x0003	1		1b	2	
130	0x0082	127	1111 111	1b	3	11b
257	0x0101	127	1111 111	1b	4	01b
258	0x0102	1		1b	5	

2 byte * 6 = 96 bits

Pack: 111111 (Deltas) 1101 (PosΔ) 1111111 1111111 (Exceptions) — 24 bits

Pack deltas, exceptions and position deltas bit-by-bit. The pack is 24-bit long.
The compression ratio of this data example is 96/24=4

Input data: the sequence of integers

Representation in PC: the array of 2-byte words. Which results in a 96-bit size pack.

Step 1. Calculate deltas.

Step 2. Remove leading zeros.

Step 3. Withdraw exceptions.

Step 4. Pack everything bit-by-bit.

Compression ratio:

96/24 = 4

Meta data

Sequence	uint16	Deltas	Deltas(bin)			
			Exception	Delta	Pos	PosDelta
1	0x0001	1		1b	0	
2	0x0002	1		1b	1	
3	0x0003	1		1b	2	
130	0x0082	127	1111 111	1b	3	11b
257	0x0101	127	1111 111	1b	4	01b
258	0x0102	1		1b	5	

2 byte * 6 = 96 bits

Pack: 24 bits — 111111 Deltas | 1101 PosΔ | 1111111 1111111 Exceptions

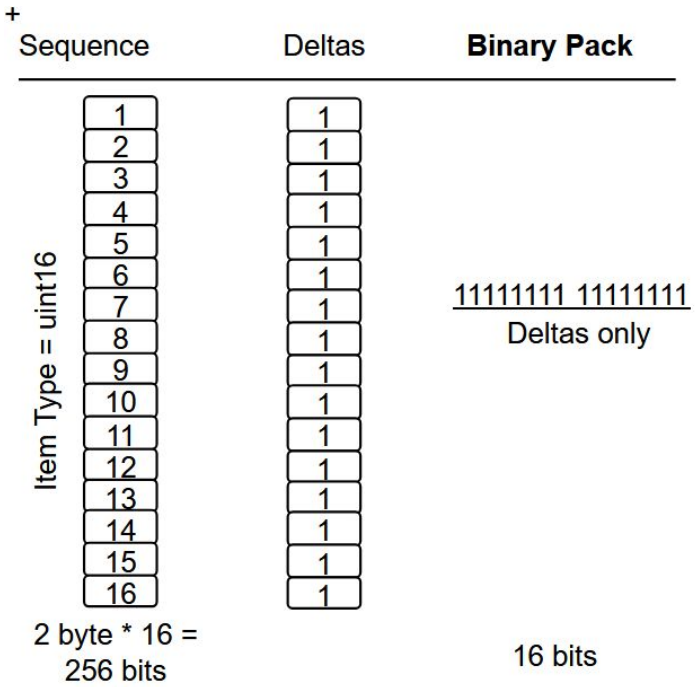
```
typedef struct {
    size_t item_cnt;
    size_t delta_wid;
    size_t exc_cnt;
    size_t exc_wid;
    size_t exc_pos_wid;
    size_t nbytes; /* size
    uint8_t *pack;
    bool outer_mem;
    bool signed_deltas;
} dfor_meta_t;
```

Not only does a pack consume resources, but the meta data, needed for decoding, also takes capacity.

Two counters: of items and of exceptions.
Three widths: of deltas, of exceptions, of exception positions.

And it is the overhead which we have to sacrifice.
The size of each field in meta data is defined by the range of item_cnt. It will be shown later in the example of application of DFOR to PG

The Best-Case Compression Ratio



Compression ratio

256/16 = 16

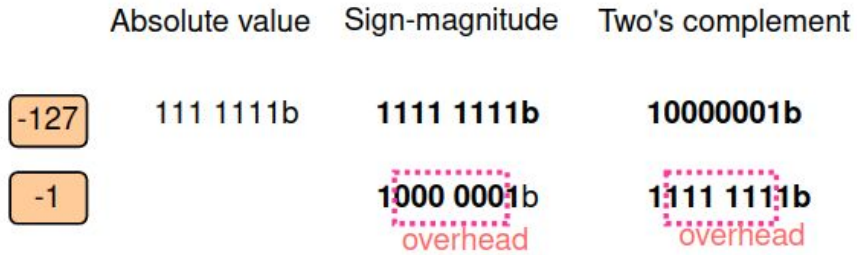
The Best-Case compression ratio
= sizeof(item) * 8

$CR_{uint8} = 8$
$CR_{uint16} = 16$
$CR_{uint32} = 32$
$CR_{uint64} = 64$

Unsorted Sequences and Negative Deltas

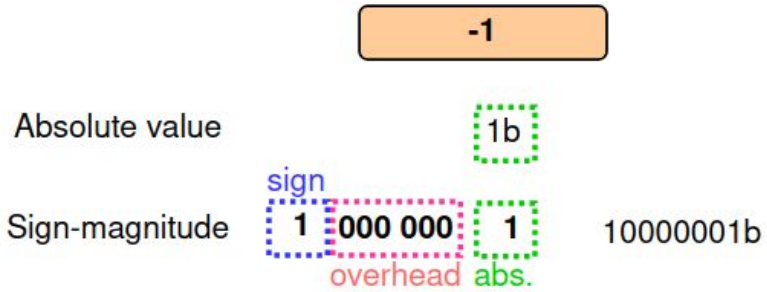
Sequence	uint16	Deltas
1	0x0001	1
130	0x0082	129
3	0x0003	-127
2	0x0002	-1
257	0x0101	255
258	0x0102	1

When we have unsorted sequence of integers as an input, the need of negative deltas appears.

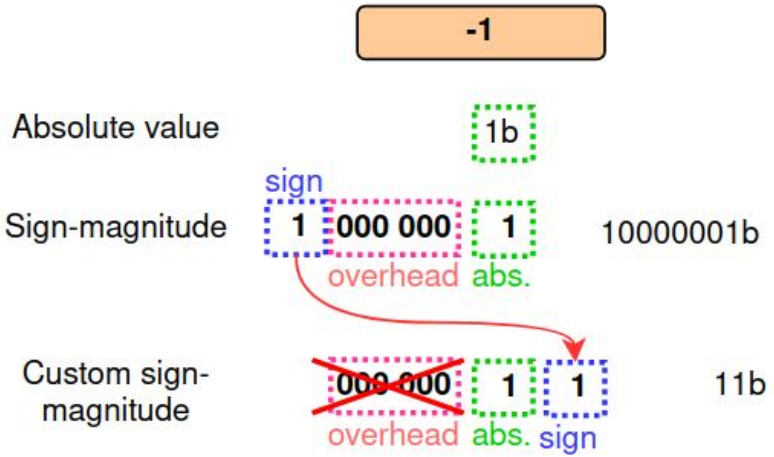


Standard codes depicting negativity can bring a considerable overhead.

Customised Sign-Magnitude code



Customised Sign-Magnitude code

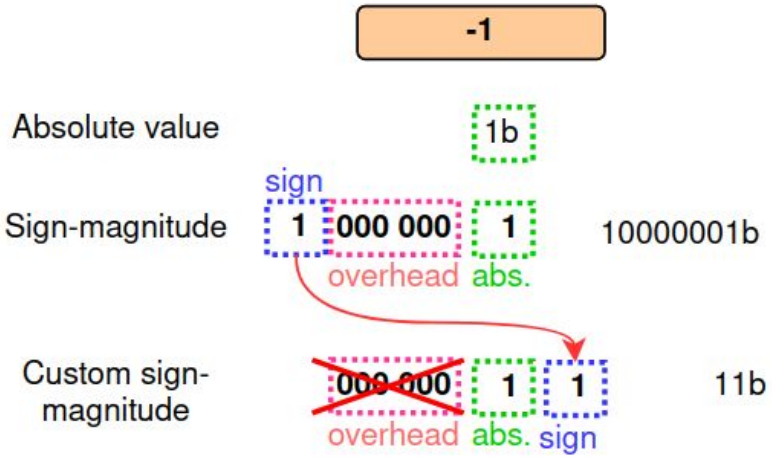


The solution is based on the sign-magnitude coding.

The sign marker bit is just moved to the lowest bit position.

We can afford this since the algorithm of ours already works on a bit-by-bit level.

Customised Sign-Magnitude code



The solution is based on the sign-magnitude coding.

The sign marker bit is just moved to the lowest bit position.

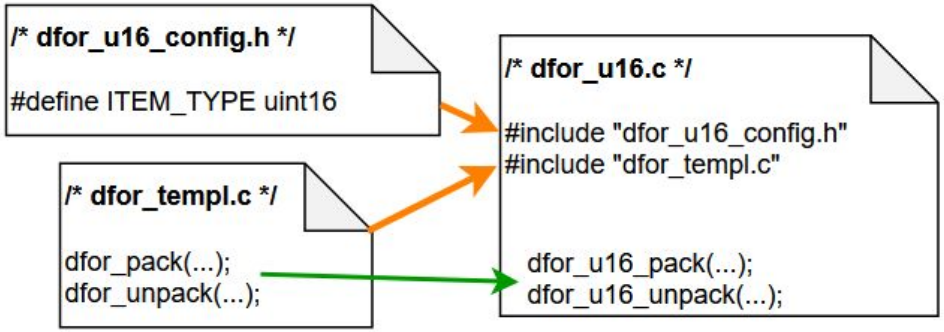
Since after that, leading seroz can be omitted as usual.

We can afford this since the algorithm of ours already works on a bit-by-bit level.

Delta signedness is optional. Sorted input sequences can be presented without signed deltas.

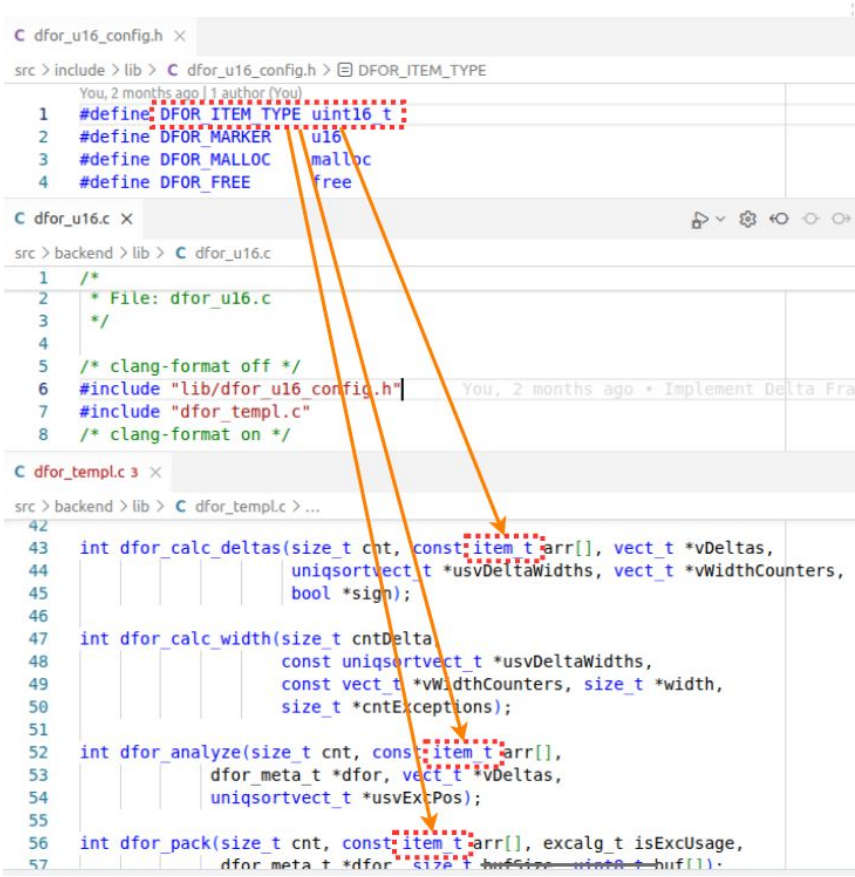
```
typedef struct {
    size_t item_cnt;
    size_t delta_wid;
    size_t exc_cnt;
    size_t exc_wid;
    size_t exc_pos_wid;
    size_t nbytes; /* si
    uint8_t *pack;
    bool outer_mem;
    bool signed_deltas;
} dfor_meta_t;
```

DFOR library code architecture

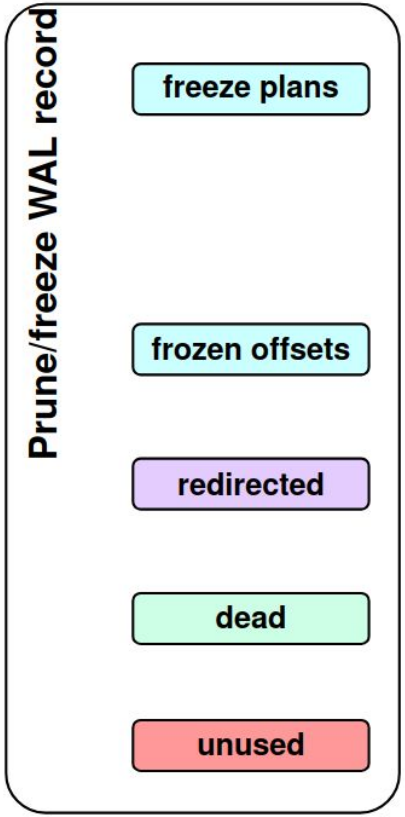


The code architecture of the DFOR unit provides header templates comprising macros and function patterns.

This allows developers to easily create an instantiation of the DFOR supporting an integer item of any size that is a multiple of 8 bits.



DFOR in PostgreSQL. Compression of Prune/Freeze XLog



Array of structure,
not a sequence of integers,
Can't use DFOR here

```
typedef struct xlhp_freeze_plan
{
    TransactionId xmax;
    uint16      t_infomask2;
    uint16      t_infomask;
    uint8       frzflags;

    /* Length of individual page o
    uint16      ntuples;
} xlhp_freeze_plan;
```

A sorted sequence of integers,
Use DFOR here

An unsorted sequence of pairs of integers
Use DFOR here

A sorted sequence of integers
Use DFOR here

A sorted sequence of integers
Use DFOR here

Thread on pgsql-hackers

DFOR in PostgreSQL. Compression of Prune/Freeze XLog

```
* For example, Meta for BLCKSZ equal to 32768 has next sizes of field
* | sect. | byte | bits      |   param            |  size  | range of values |
* |-------|------|-----------|--------------------|--------|-----------------|
* |  main | 0, 1 | 0-11      | item count         | 12 bit | 1...2730        |
* |  main |    1 | 12-15     | delta width        |  4 bit | 1...12          |
* |  main |    2 | 16        | signed deltas flag |  1 bit | 0...1           |
* |  main |    2 | 17        | exception flag     |  1 bit | 0...1           |
* | extra | 2, 3 | 18-26     | exception count    |  9 bit | 1...274         |
* | extra |    3 | 27-29     | exception width    |  4 bit | 1...11          |
* | extra | 3, 4 | 30-33     | except pos. width  |  4 bit | 1...12          |
*/
```

Implementation provides precise calculations of maximum sizes of fields in metadata depending on a page size and, accordingly, on a number of tuples per page.

Metadata is also packed bit-by-bit and transmitted as a part of Prune/Freeze WAL record.

Thread on pgsql-hackers

DFOR in PostgreSQL. Compression of Prune/Freeze XLog

```
=== VACUUM (table with no index) ===
-------------------- ---------------- ---------------- -----------
                     DFOR off, bytes   DFOR on, bytes   Reduction
-------------------- ---------------- ---------------- -----------
WAL total size               6743149          1184446         82%
Prune records size           6710185          1159723        5.8x
-------------------- ---------------- ---------------- -----------

=== VACUUM (table with index) ===
-------------------- ---------------- ---------------- -----------
                     DFOR off, bytes   DFOR on, bytes   Reduction
-------------------- ---------------- ---------------- -----------
WAL total size              20394208          8907090         56%
Prune records size           6812850          1225944        5.6x
-------------------- ---------------- ---------------- -----------
```

Prune/Freeze record size reduction by more than 5 times.

Thread on pgsql-hackers

TODO: DFOR compression of GIN

GIN exploits TIDs sequences and saves it on the disk, so that GIN can be the most appropriate candidate to be developed with DFoR.

GIN already uses delta compression but compression ratio can be considerably improved with DFOR.

TODO

Join ‘DFOR for PostgreSQL’ activity

Compress prune/freeze records with Delta Frame of Reference algorithm (PGSQL-HACKERS)

TODO list:

- **fake overflow feature;**
- **iterator;**
- **applying DFOR to GIN.**

“There are no accidents!”

Master Oogway

Thank you for attention!

Contact us:
Tantor Labs, LLC

Evgeny Voropaev
evorop@gmail.com

TODO: Fake overflow feature