deckhouse
user community

meetup/3

Prometheus на диете

Оптимизация ядра хранения данных Deckhouse Prom++

Deckhouse
ФЛАНТ

Владимир Пустовалов

C++ backend developer

О компании «Флант»

Deckhouse
ФЛАНТ

15+
лет опыта
в Open Source

С 2017
года используем
Kubernetes в production

№ 1
контрибьютор в проекты
CNCF из России

400+
сотрудников

>260
компаний-
пользователей

>11 100
звёзд у наших Open
Source-проектов на GitHub

Prometheus

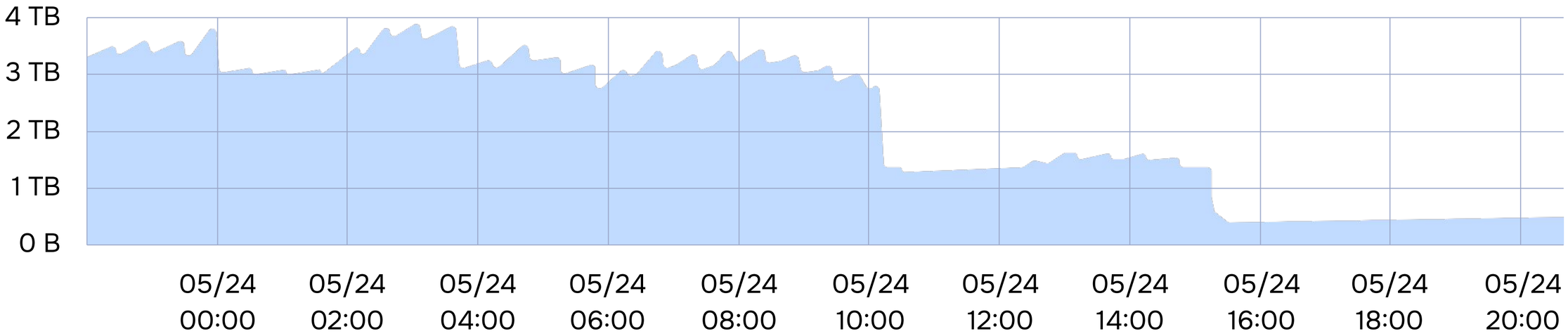
Инструмент мониторинга с открытым исходным кодом, обеспечивающий сбор, хранение и анализ данных о работе приложений и инфраструктуры.

Deckhouse Prom++

Prom++ — оптимизированный Prometheus с потреблением памяти до 10 раз меньше

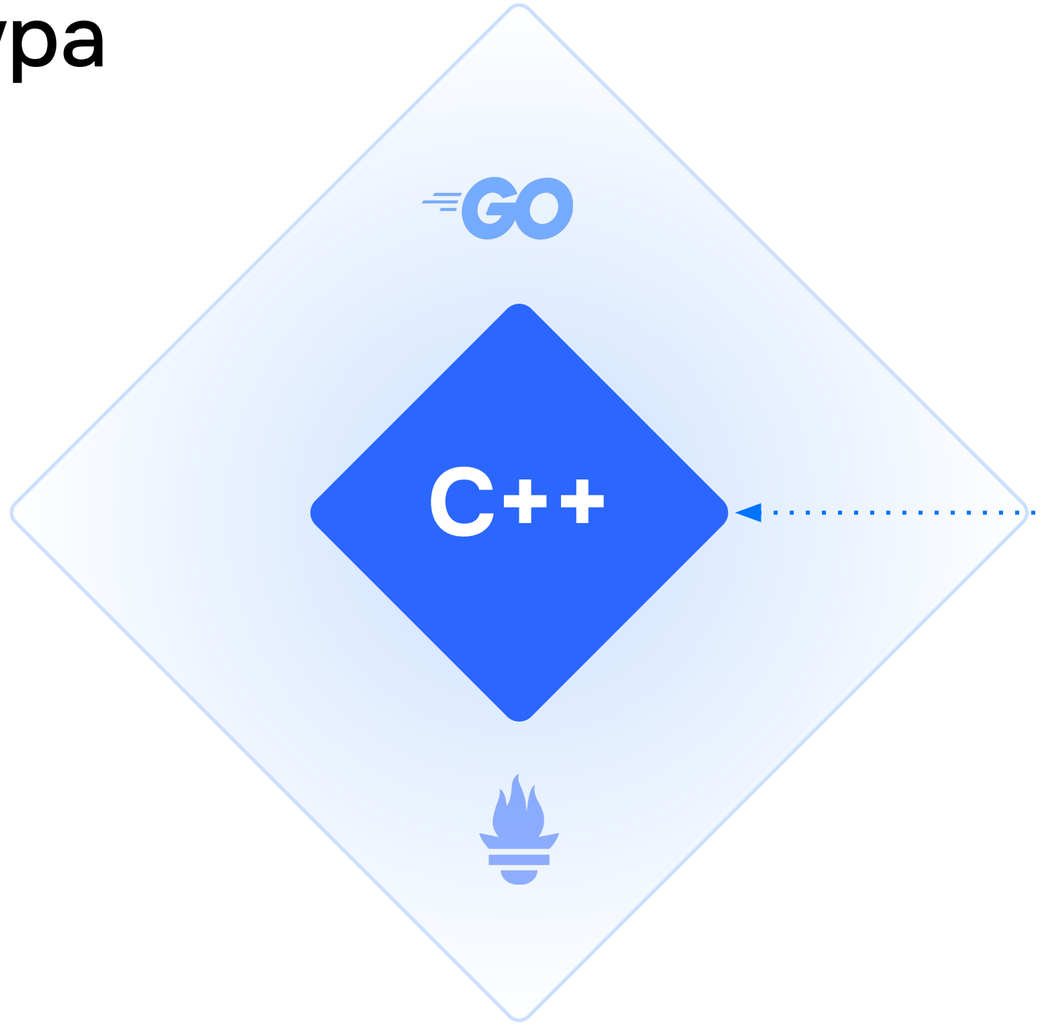
User case

Потребление памяти: было 3,8 TB ·····► стало 0,6 TB




Откуда столько?
Перешел на Prom++

Архитектура Prom++



Ядро хранения и обработки горячих данных

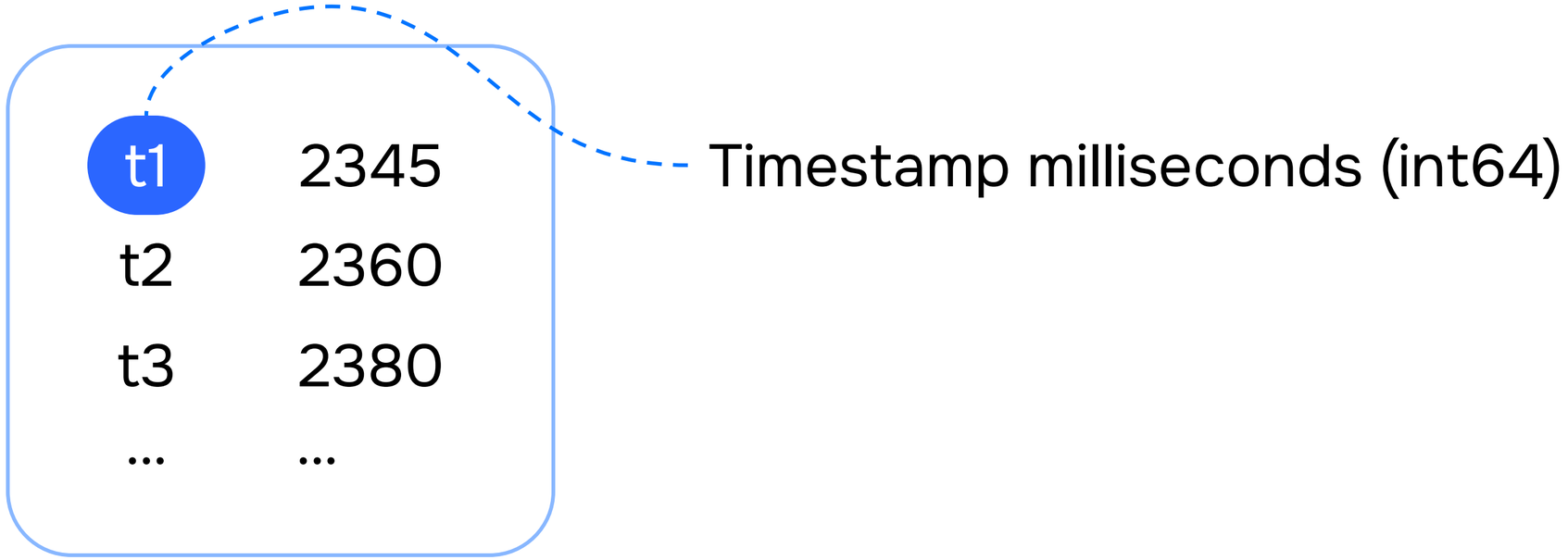
Хранилище данных

Deckhouse
ФЛАНТ

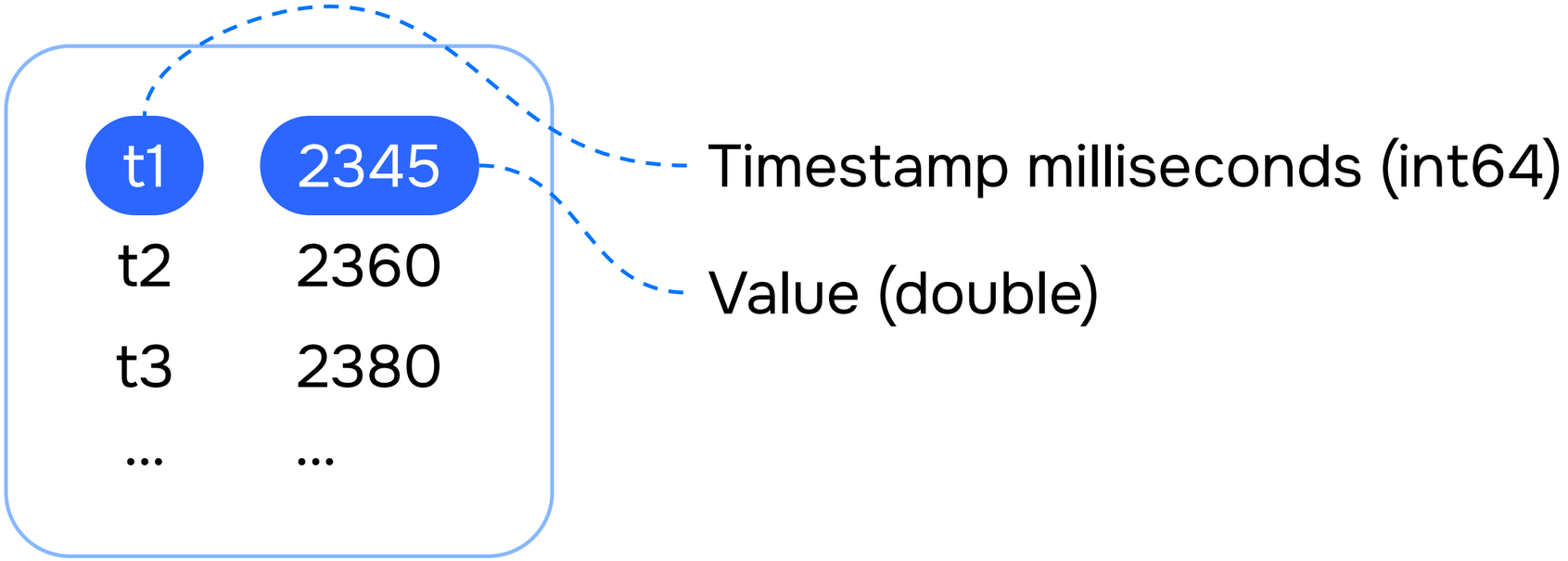
Data storage

t1	2345
t2	2360
t3	2380
...	...

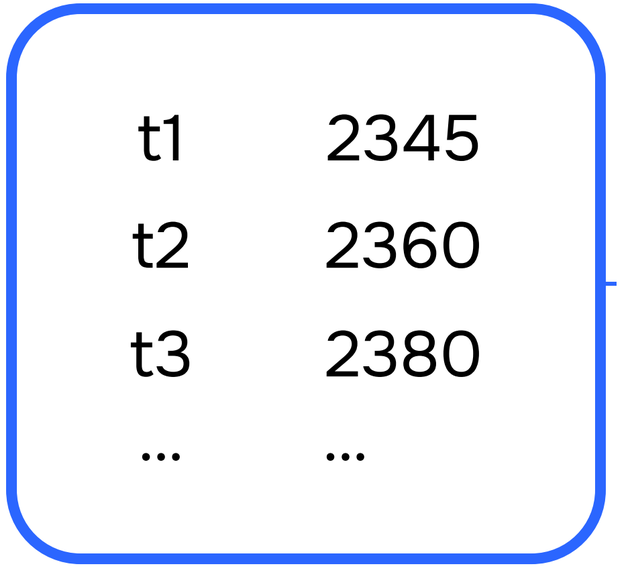
Data storage



Data storage



Data storage



Серия

У каждой серии свой уникальный ID (uint32) (auto increment)

Data storage

t1	2345
t2	2360
t3	2380
...	...

240 точек

за 2 часа = 120 минут

сбор каждые 30 секунд

Data storage

t1	2345
t2	2360
t3	2380
...	...

240 точек

за 2 часа = 120 минут
сбор каждые 30 секунд

1+ млн серий

240+ млн точек

Data storage

t1	2345
t2	2360
t3	2380
...	...

240 точек

за 2 часа = 120 минут
сбор каждые 30 секунд

10+ млн серий

2,4+ млрд точек

Benchmarking

Deckhouse
ФЛАНТ

Бенчмаркинг

Это процесс измерения и сравнения производительности различных алгоритмов

Бенчмаркинг

01 AMD Ryzen 9 3900 @ 3.1GHz, Ubuntu 22.04.4

02 Google Benchmark

03 Множественный прогон бенчмарка → минимальное время

Исходные данные для бенчмарка

Серий: 1 208 872 Точек: 241 984 682

Исходные данные для бенчмарка

Серий: 1 208 872 Точек: 241 984 682

```
struct SeriesSample {
 int64_t timestamp;
 double value;
};
```

Исходные данные для бенчмарка

Серий: 1 208 872 Точек: 241 984 682

```
struct SeriesSample {
 int64_t timestamp;
 double value;
};
```

sizeof(SeriesSample) * 241 984 682 = **3.78 Гб**

Код бенчмарка

```
class DataStorage {
public:
 size_t allocated_memory() const noexcept;
};
```

Код бенчмарка

```
class DataStorage {
public:
 size_t allocated_memory() const noexcept;
};

class Encoder {
public:
 void encode(
    uint32_t series_id, int64_t timestamp, double value);
};
```

Алгоритм бенчмарка

01 Считываем исходные данные из файла

02 Кодируем данные (замер времени)

03 Выводим потребление памяти и среднее время кодирования одной точки

Запуск бенчмарка

```
nice -n -20 \
./benchmark \
  --benchmark_context=samples_file="samples.dat" \
  --benchmark_repetitions=10
```

Реализация 1
Gorilla

Deckhouse
ФЛАНТ

Gorilla

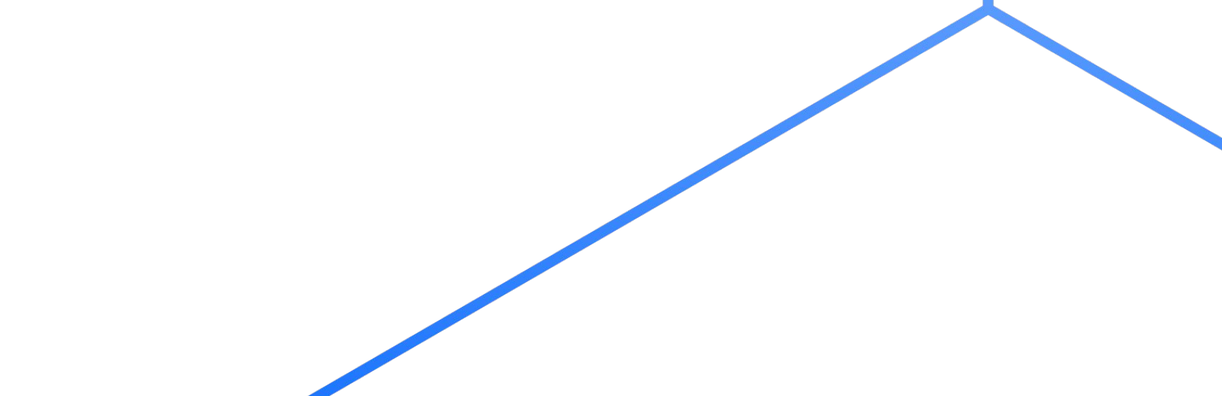
- Fast
- Scalable
- In-Memory Time Series Database

01 Gorilla Values encoder

02 Gorilla Timestamp encoder

01 Gorilla Values encoder

02 Gorilla Timestamp encoder



Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

Первое значение записываем как есть (64 бита)

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 1024.0 **xor** 100.0 = 0xC900000000000

0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 …

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 1024.0 **xor** 100.0 = 0xC900000000000

`0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 …`

02 Leading zeros: 8, length: 8, trailing zeros: 48

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 1024.0 **xor** 100.0 = 0xC900000000000

`0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 …`

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: `1 1` (2 бита)

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 1024.0 **xor** 100.0 = 0xC900000000000

0 0 0 0 0 0 0 0 **1 1 0 0 1 0 0 1** 0 0 0 0 0 0 0 0 …

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: 1 1 (2 бита)

04 Записываем leading zeros: (5 бит)

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 1024.0 **xor** 100.0 = 0xC900000000000

`0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 …`

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: 1 1 (2 бита)

04 Записываем leading zeros: (5 бит)

05 Записываем length: (6 бит)

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 1024.0 **xor** 100.0 = 0xC900000000000

`0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 …`

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: `1 1` (2 бита)

04 Записываем leading zeros: (5 бит)

05 Записываем length: (6 бит)

06 Записываем островок (length бит)

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 1024.0 **xor** 100.0 = 0xC900000000000

`0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 …`

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: 1 1 (2 бита)

04 Записываем leading zeros: (5 бит)

05 Записываем length: (6 бит)

06 Записываем островок (length бит)

Итого мы записали

21 бит вместо 64

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 1024.0 = 0xF900000000000

0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 …

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 1024.0 = 0xF900000000000

0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 …

02 Leading zeros: 8, length: 8, trailing zeros: 48

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 1024.0 = 0xF900000000000

0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 …

0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 …

02 Leading zeros: 8, length: 8, trailing zeros: 48

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 1024.0 = 0xF900000000000

0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 …

0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 …

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: 1 0 (2 бита)

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 1024.0 = 0xF900000000000

0 0 0 0 0 0 0 0 **1 1 0 0 1 0 0 1** 0 0 0 0 0 0 0 0 …

0 0 0 0 0 0 0 0 **1 1 1 1 1 0 0 1** 0 0 0 0 0 0 0 0 …

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: 1 0 (2 бита)

04 Записываем островок (length бит)

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 1024.0 = 0xF900000000000

0 0 0 0 0 0 0 0 **1 1 0 0 1 0 0 1** 0 0 0 0 0 0 0 0 …

0 0 0 0 0 0 0 0 **1 1 1 1 1 0 0 1** 0 0 0 0 0 0 0 0 …

02 Leading zeros: 8, length: 8, trailing zeros: 48

03 Записываем ключ: 1 0 (2 бита)

04 Записываем островок (length бит)

Итого мы записали

10 бит вместо 64

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 200.0 = 0x00

Gorilla Values encoder

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

01 200.0 **xor** 200.0 = 0x00

02 Записываем ключ: 0 (1 бит)

Gorilla Values encoder

01 200.0 **xor** 200.0 = 0x00

02 Записываем ключ: 0 (1 бит)

Последовательность значений:

100.0, 1024.0, 200.0, 200.0

Итого мы записали

1 бит вместо 64

01 Gorilla Values encoder

02 Gorilla Timestamp encoder

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

Первое значение записываем как varint
(от 1 до 10 байт)

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

Рассчитываем дельту (150 - 100 = 50)
и записываем как varint (от 1 до 10 байт)

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

01 Рассчитываем дельту дельты

(200 - 150) - 50 = 0

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

01 Рассчитываем дельту дельты
(200 - 150) - 50 = 0

02 Записываем ключ: 0 (1 бит)

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

01 Рассчитываем дельту дельты

(251 - 200) - 50 = 1

… 0 0 0 0 0 0 0 1

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

01 Рассчитываем дельту дельты

(251 - 200) - 50 = 1

… 0 0 0 0 0 0 0 1

02 Количество значащих бит: 1

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

01 Рассчитываем дельту дельты
(251 - 200) - 50 = 1

… 0 0 0 0 0 0 0 1

02 Количество значащих бит: 1

03 Записываем ключ и сами значащие биты согласно таблице

Gorilla Timestamp encoder

Последовательность значений:

100, 150, 200, 251

01 Рассчитываем дельту дельты
(251 - 200) - 50 = 1

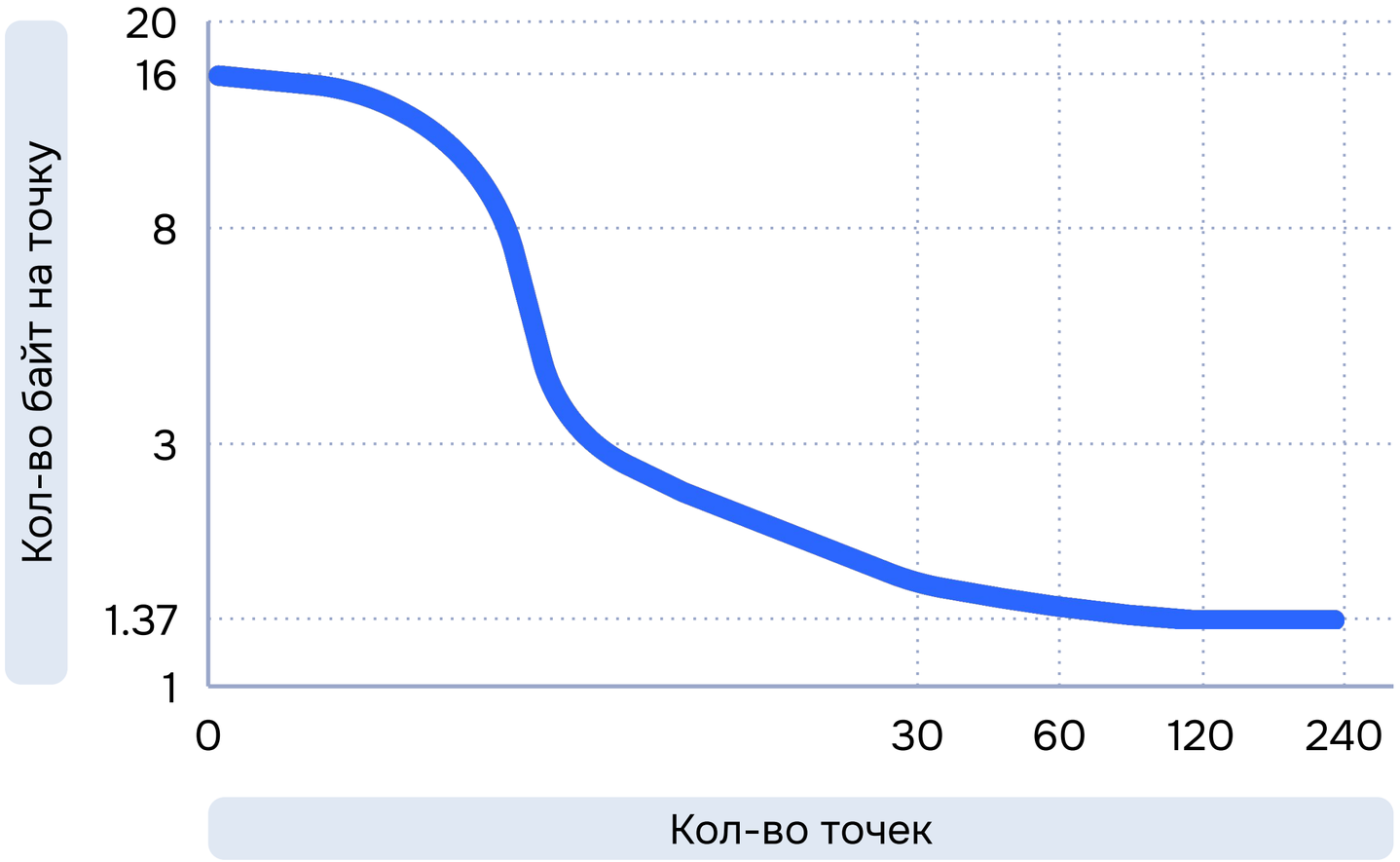
… 0 0 0 0 0 0 0 1

02 Количество значащих бит: 1

03 Записываем ключ и сами значащие биты согласно таблице

Кол-во значащих бит	Ключ
<= 4	1 0
<= 14	1 1 0
<= 17	1 1 1
Остальное	1 1 1 1

Gorilla encoder



Gorilla encoder

Ожидаемый объём памяти

~~1,37~~ 2 * 241 984 682 =

461,54 МБ

Бенчмарк

	Память	Время кодирования точки
Raw	3,78 ГБ	
Gorilla ожидание	461,54 МБ	
Gorilla реальность	545,46 МБ	31,75 ns

Overhead
83,92 МБ

Почему так?

01 Состояние энкодера для каждой серии

```
struct EncoderState {
 TimestampEncoderState ts_state_;
 ValuesEncoderState values_state_;
 uint8_t sample_count_;
 BitSequence bit_sequence_;
};

sizeof(EncoderState) * 1 208 872 =
48,42 МБ
```

Почему так?

01 Состояние энкодера для каждой серии

02 Аллоцирование памяти с запасом

std::vector

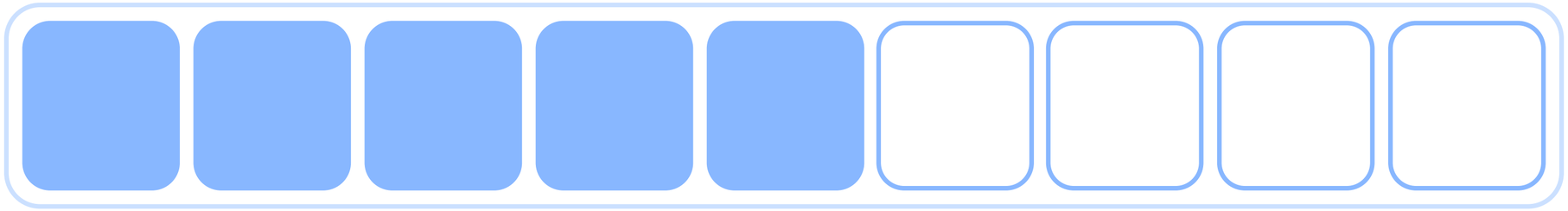
```
struct EncoderState {
 TimestampEncoderState ts_state_;
 ValuesEncoderState values_state_;
 uint8_t sample_count_;
 BitSequence bit_sequence_;
};

std::vector<EncoderState> series;
```

Почему std::vector?

- **Сложность вставки:** амортизированная константа
- **Сложность доступа:** константа
- Cache-friendly

Проблемы std::vector

x2 allocation

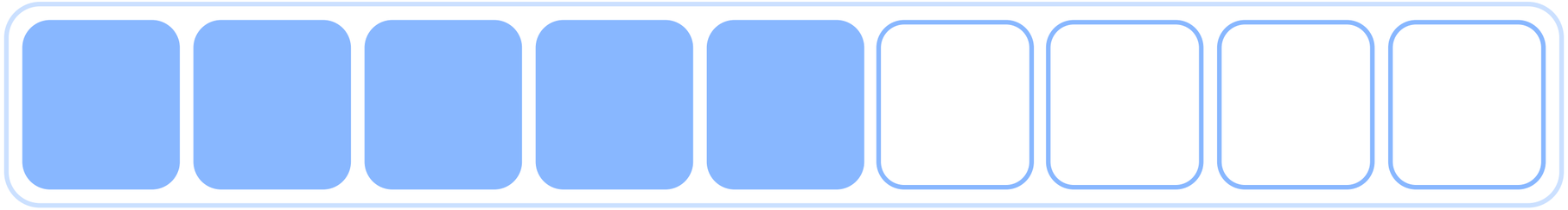


Size

Capacity

Проблемы std::vector

x2 allocation

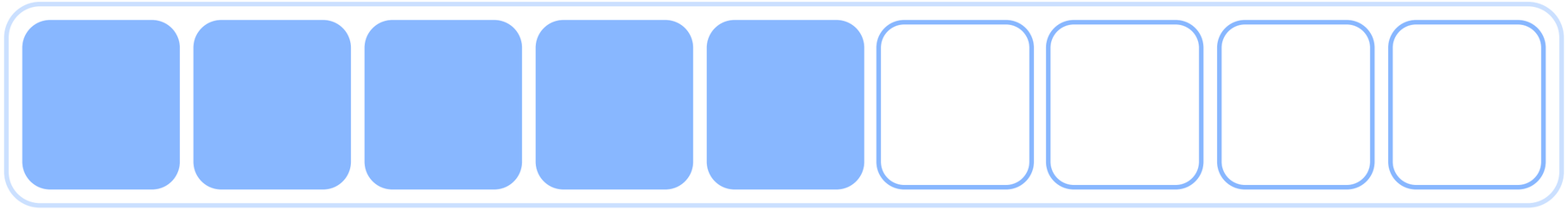


Size: 1 208 872

Capacity: 2 097 152

Проблемы std::vector

x2 allocation



Size: 1 208 872

Capacity: 2 097 152

Неиспользуемая память: ~35,5 МБ

Пишем свой Vector!

BareBones::Vector

Аллокация памяти:

- Рост на 50 % с округлением до 32 байт, если итоговый размер памяти < 256 (только для объектов с sizeof < 8 байт)
- Рост на 50 % с округлением до 256 байт, если итоговый размер памяти < 4096
- Рост на 10 % с округлением до 4096 байт

Jemalloc

Аллокатор памяти, оптимизированный для снижения фрагментации и работы на многопроцессорных системах

Бенчмарк

	Память	Время кодирования точки
Raw	3,78 ГБ	
Gorilla std::vector	545,46 МБ	31,75 ns
Gorilla BareBones::Vector	512,21 МБ	26,93 ns

Отыграли 33,25 МБ

Ускорили кодирование точки на 4,82 ns

Реализация 2
Timestamp storage

Deckhouse
ФЛАНТ

Timestamp storage

Серия # 1

123	1,0
153	1,0
183	1,0
…	…

Серия # 2

123	32,34
153	52,3
183	60,1
…	…

Серия # 3

123	142 876
153	164 547
183	112 776
…	…

Timestamp storage

Серия # 1

123	1,0
153	1,0
183	1,0
…	…

Серия # 2

123	32,34
153	52,3
183	60,1
…	…

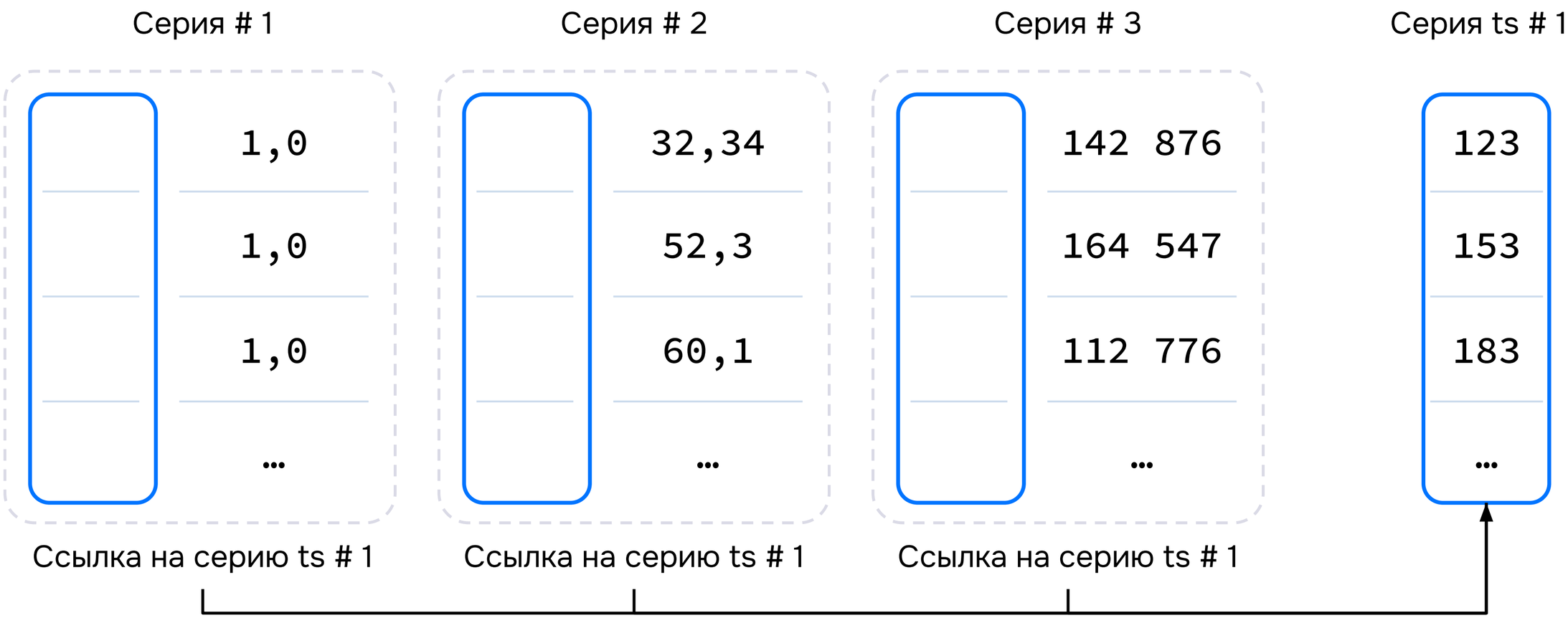
Серия # 3

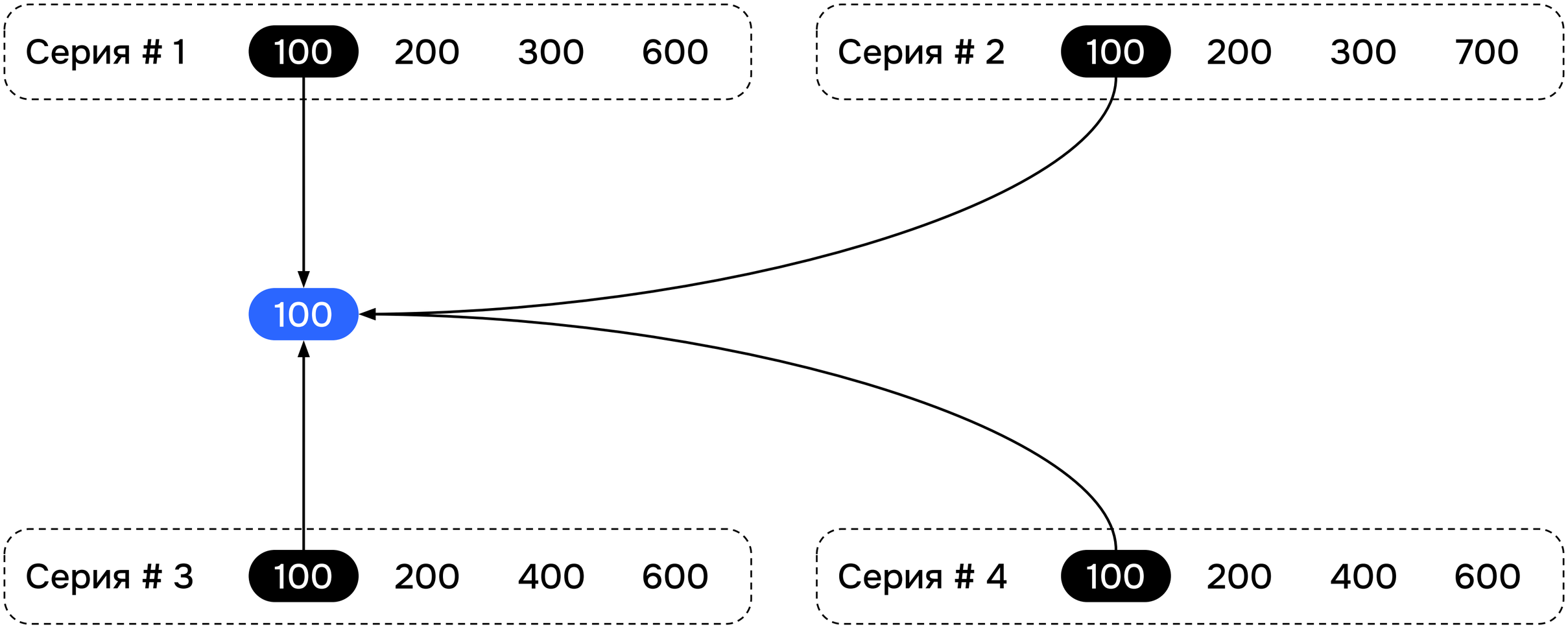
123	142 876
153	164 547
183	112 776
…	…

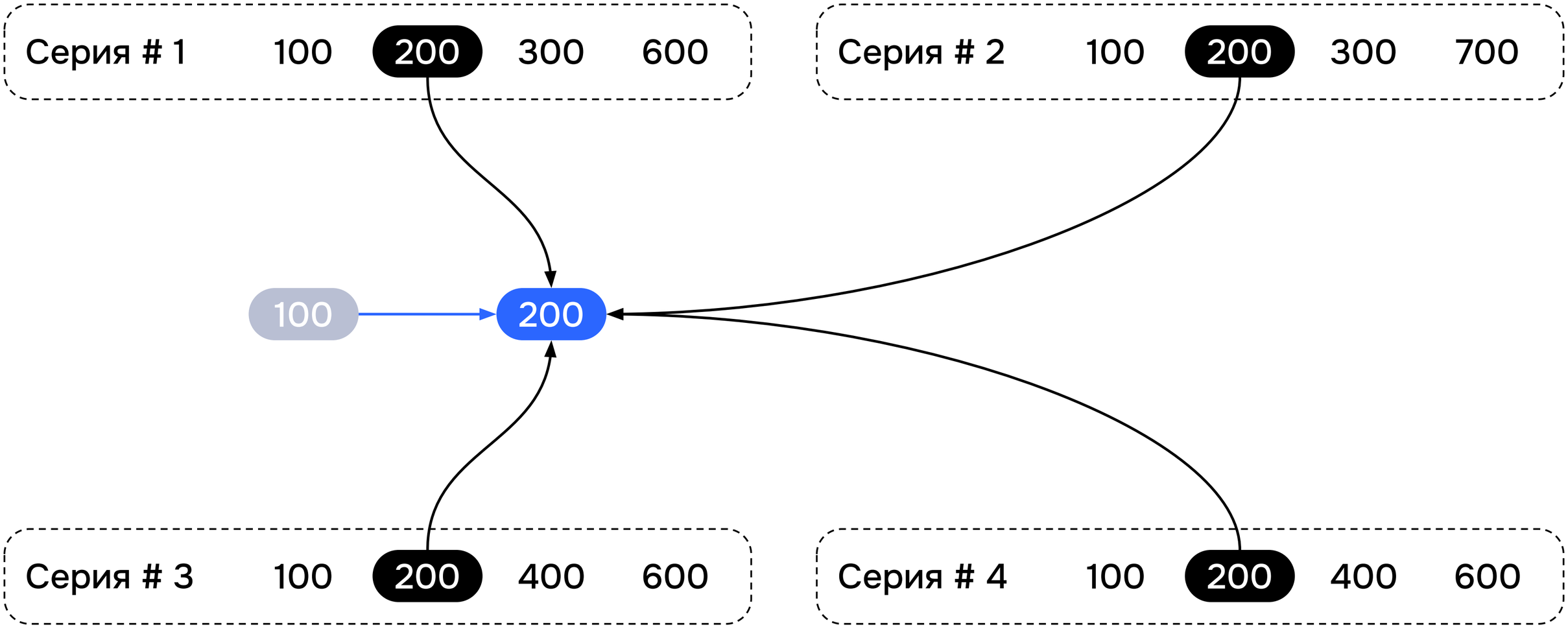
10 %

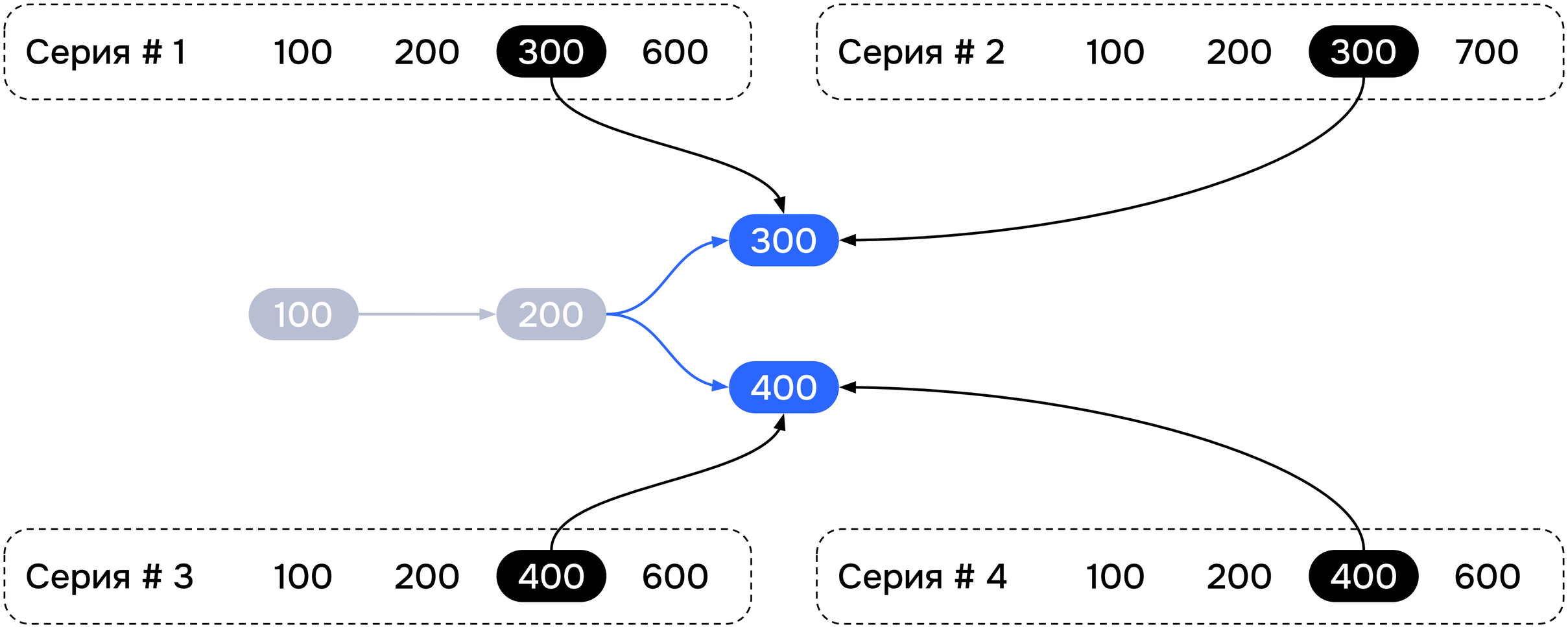
уникальных ts!

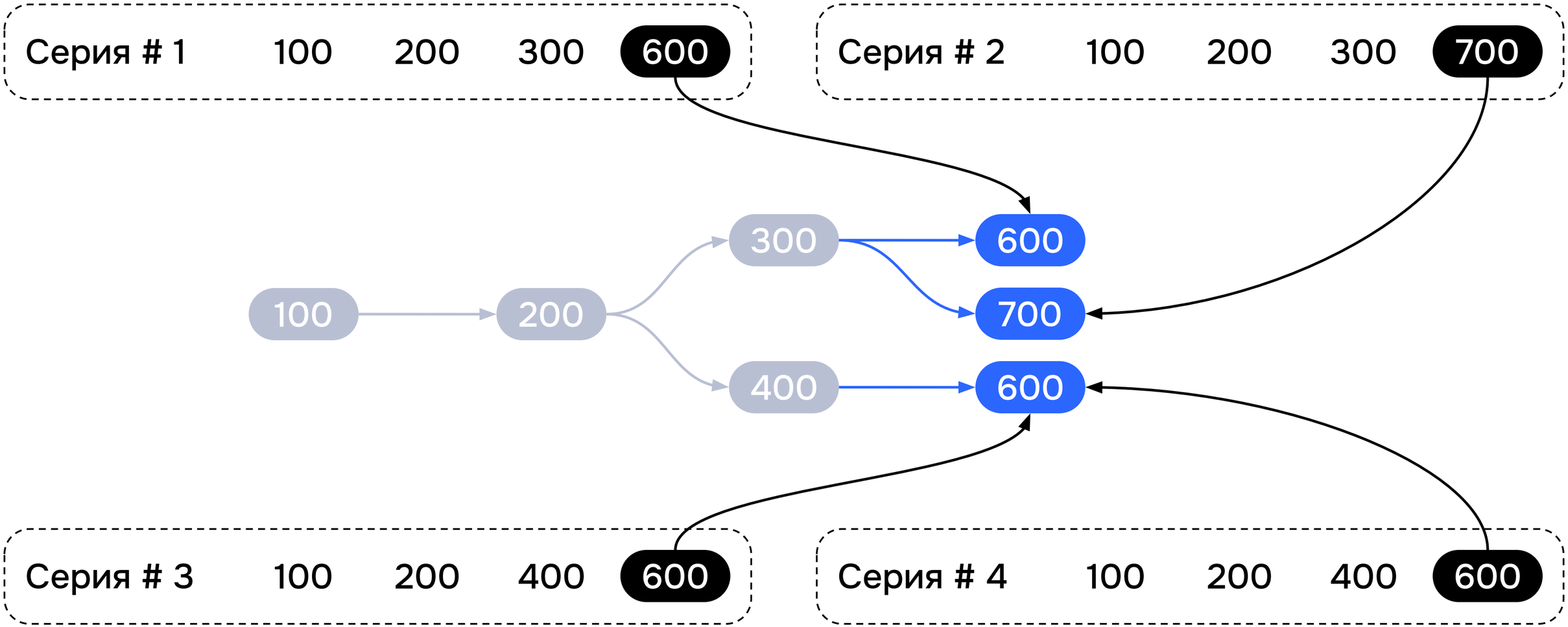
Timestamp storage




Серия # 1
100
200
300
600
Серия # 2
100
200
300
700
100
Серия # 3
100
200
400
600
Серия # 4
100
200
400
600


Серия # 1
100
200
300
600
Серия # 2
100
200
300
700
100
200
Серия # 3
100
200
400
600
Серия # 4
100
200
400
600


Серия # 1
100
200
300
600
Серия # 2
100
200
300
700
300
100
200
400
Серия # 3
100
200
400
600
Серия # 4
100
200
400
600


Серия # 1
100
200
300
600
Серия # 2
100
200
300
700
100
200
300
400
600
700
600
Серия # 3
100
200
400
600
Серия # 4
100
200
400
600

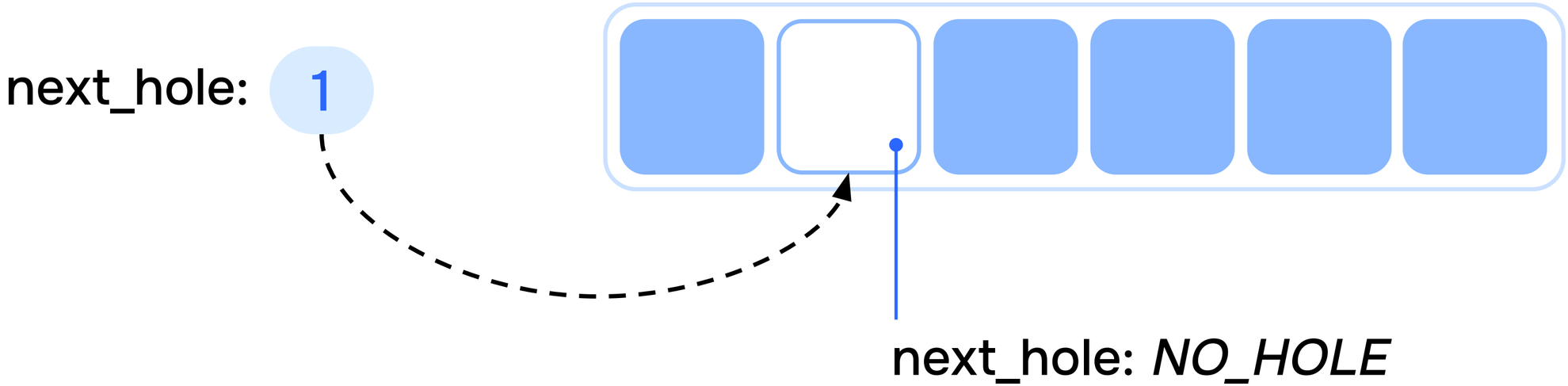
Почему не BareBones::Vector?

- ✓ **Сложность вставки:** амортизированная константа
- ✓ **Сложность доступа:** константа
- ✓ Cache-friendly
- ✗ **Сложность удаления:** линейная

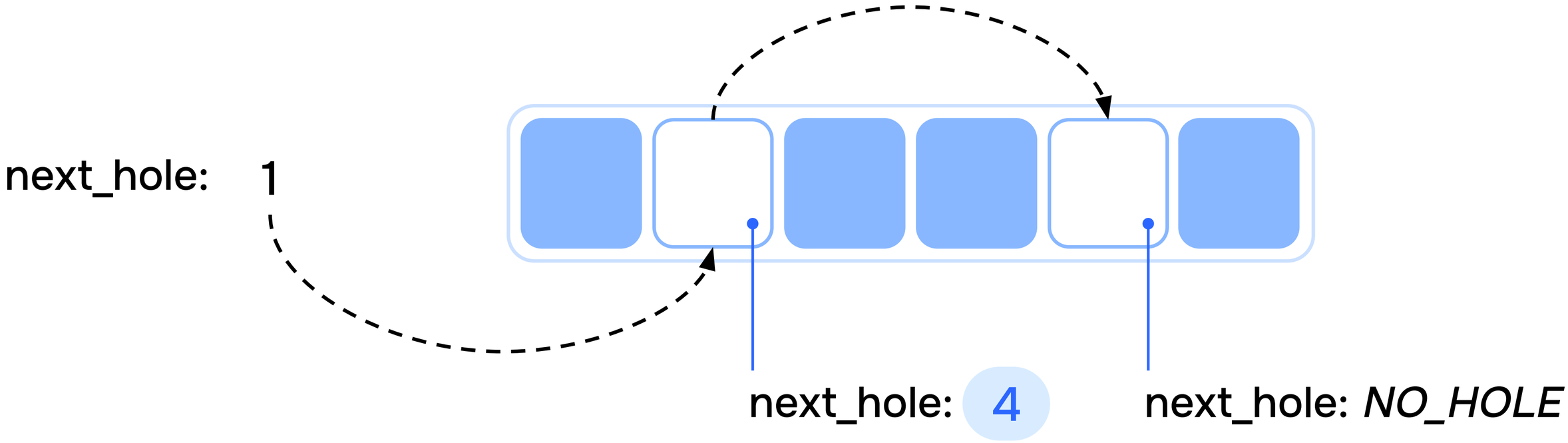
VectorWithHoles

next_hole: *NO_HOLE*

VectorWithHoles



VectorWithHoles



Union

```
union VectorItem {
 Item item;
 uint32_t next_hole;
};

sizeof(VectorItem) == max(sizeof(Item), sizeof(uint32_t))
```

Бенчмаркинг

	Память	Время кодирования точки
Raw	3,78 ГБ	
Gorilla	512,21 МБ	26,62 ns
Timestamp storage	336,12 МБ	26,01 ns

Экономия памяти
176,09 МБ (~34 %)

Ускорение кодирования точки на 0,61 ns


А че, так можно было что ли?

Реализация 3
Value encoders

Deckhouse
ФЛАНТ

ConstantEncoder

123.0

123.0

123.0

123.0

. . .

ConstantEncoder

123.0

123.0

123.0

123.0

123.0

...

```
struct DoubleConstantEncoder {
 const double value; // 8 байт
};
```

ConstantEncoder

123.0

123.0

123.0

123.0

...

```
struct DoubleConstantEncoder {
 const double value; // 8 байт
};

struct Uint32ConstantEncoder {
 const uint32_t value; // 4 байта
};
```

ConstantEncoder

123.0

123.0

123.0

123.0

...

```
struct DoubleConstantEncoder {
 const double value; // 8 байт
};

struct Uint32ConstantEncoder {
 const uint32_t value; // 4 байта
};
```

~66 % серий

TwoDoubleConstantEncoder

123.456
123.456
123.456
. . .
321.456

TwoDoubleConstantEncoder

123.456
123.456
123.456
...
321.456

```
struct TwoDoubleConstantEncoder {
 const double value1;
 const double value2;
 const uint8_t value1_count;
};
```

TwoDoubleConstantEncoder

123.456
123.456
123.456
. . .
321.456

```
struct TwoDoubleConstantEncoder {
 const double value1;
 const double value2;
 const uint8_t value1_count;
};
```

~2 % серий

AscIntegerEncoder

100.0
150.0
170.0
200.0
. . .

AscIntegerEncoder

100.0
150.0
170.0
200.0
...

```
class AscIntegerEncoder {
 GorillaTimestampEncoder encoder;
 BitSequence stream;
};
```

AscIntegerEncoder

100.0
150.0
170.0
200.0
. . .

```
class AscIntegerEncoder {
 GorillaTimestampEncoder encoder;
 BitSequence stream;
};
```

~27 % серий

GorillaEncoder

54.23
67.81
12.43
99.99
. . .

GorillaEncoder

54.23

67.81

12.43

99.99

...

```
struct GorillaEncoder {
 GorillaValuesEncoder encoder;
 BitSequence stream;
};
```

GorillaEncoder

54.23
67.81
12.43
99.99
. . .

```
struct GorillaEncoder {
 GorillaValuesEncoder encoder;
 BitSequence stream;
};
```

~5 % серий

Типы энкодеров

01 uint32_t-константа

02 float32_t-константа

03 double-константа

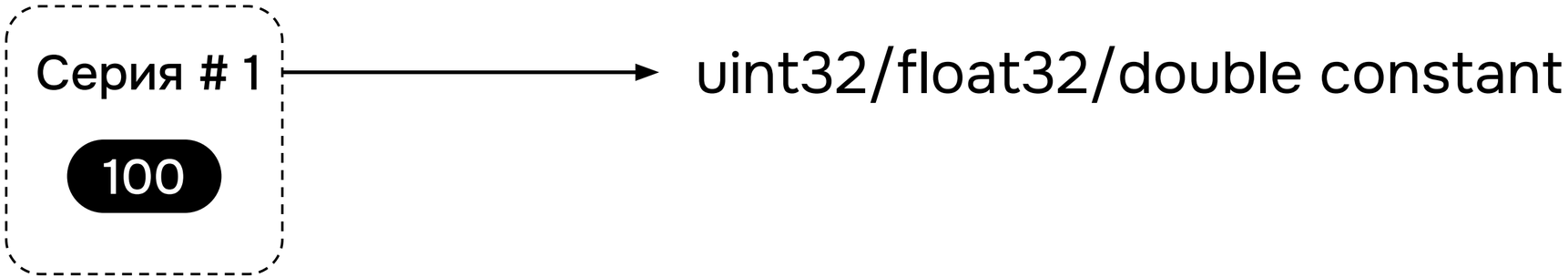
04 two-double-константа

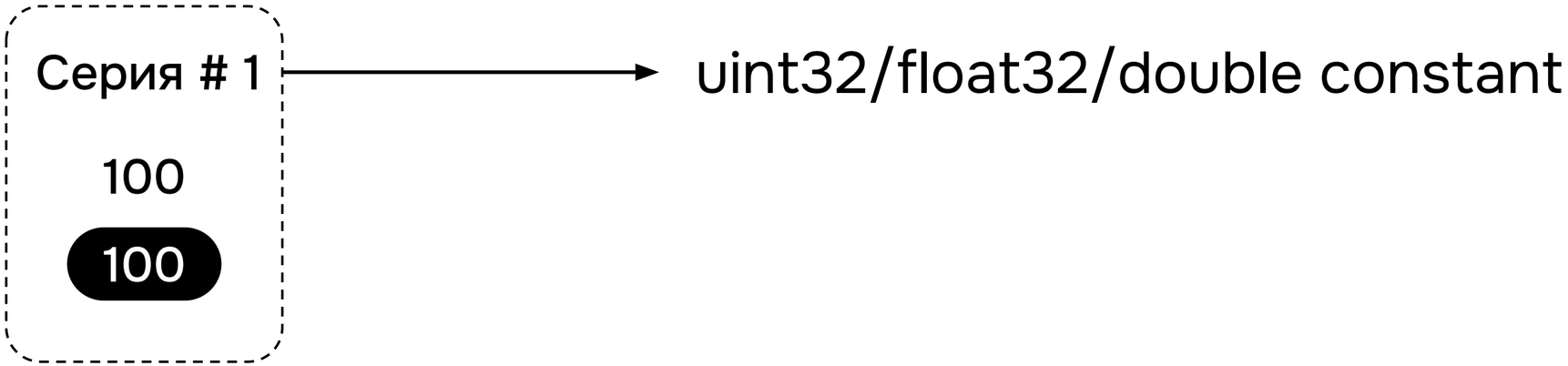
05 AscInt-последовательность

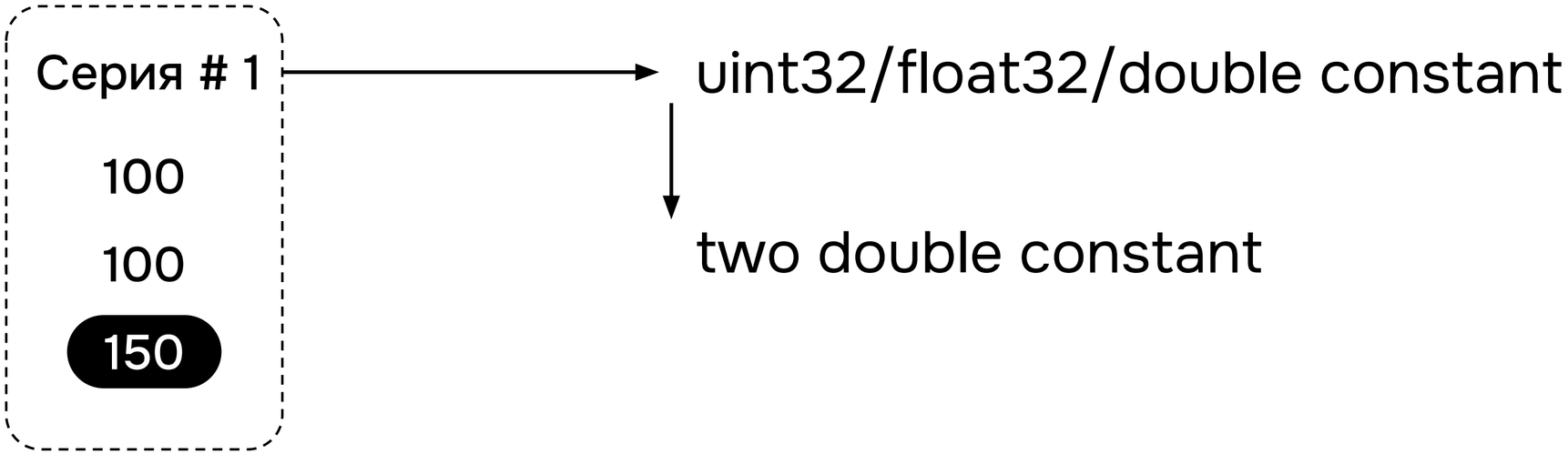
06 AscInt then ValuesGorilla

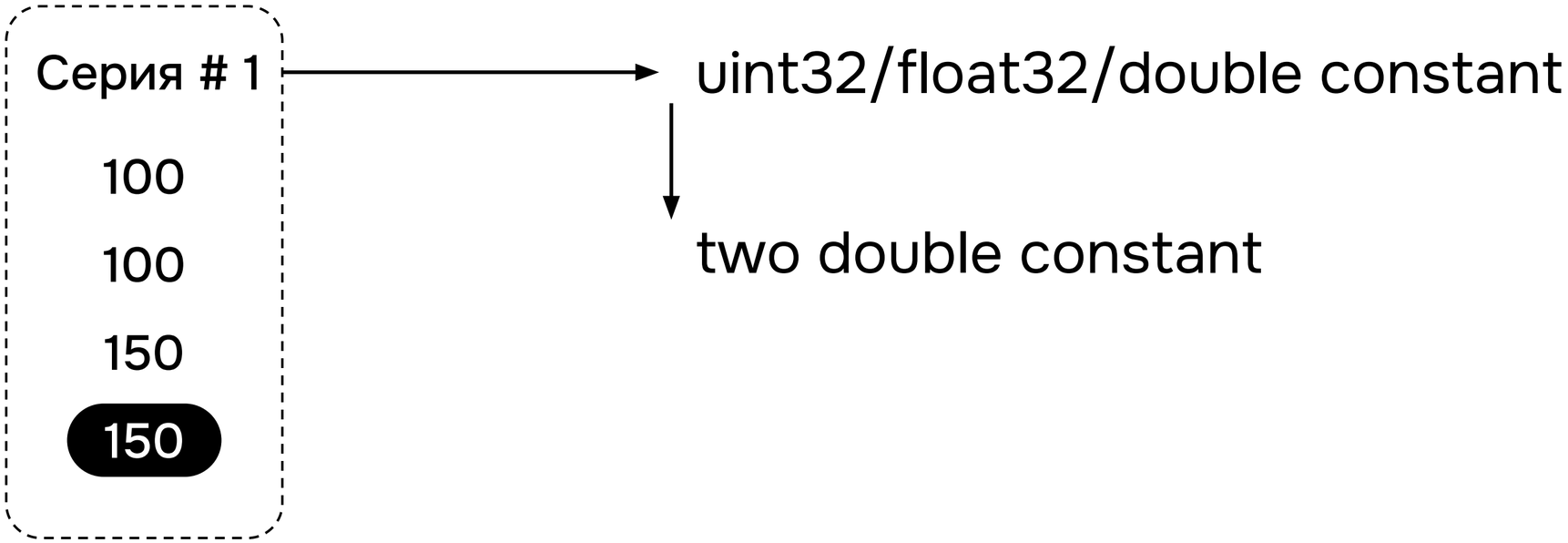
07 ValuesGorilla

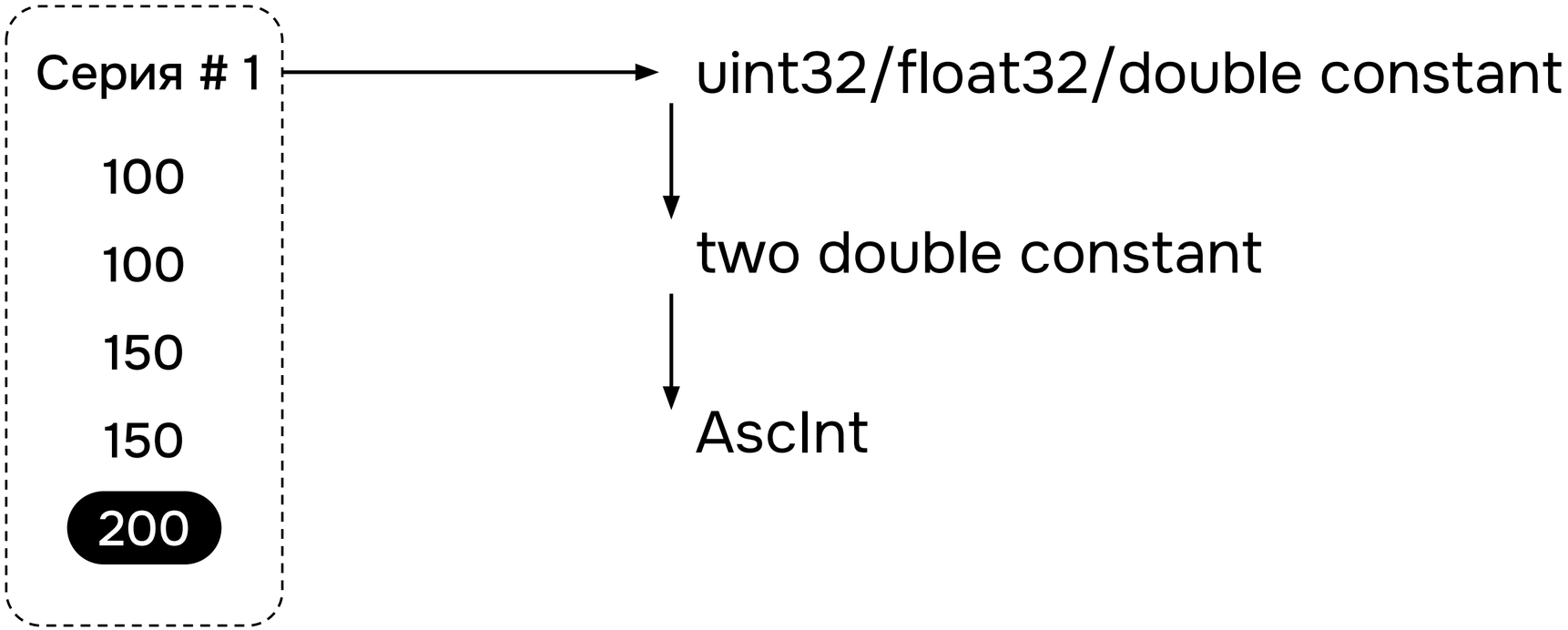
08 Gorilla (Timestamp + Value)

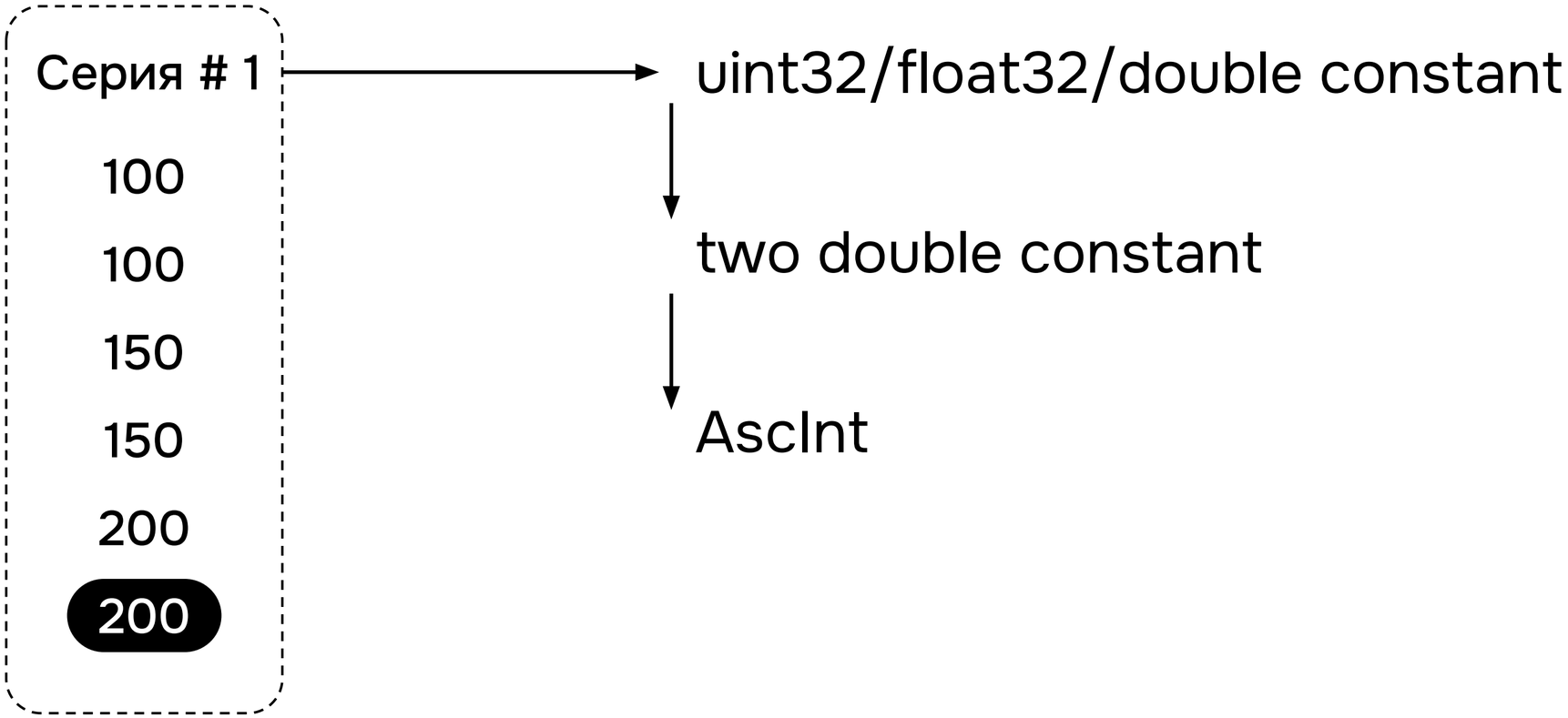

Серия # 1
100
uint32/float32/double constant

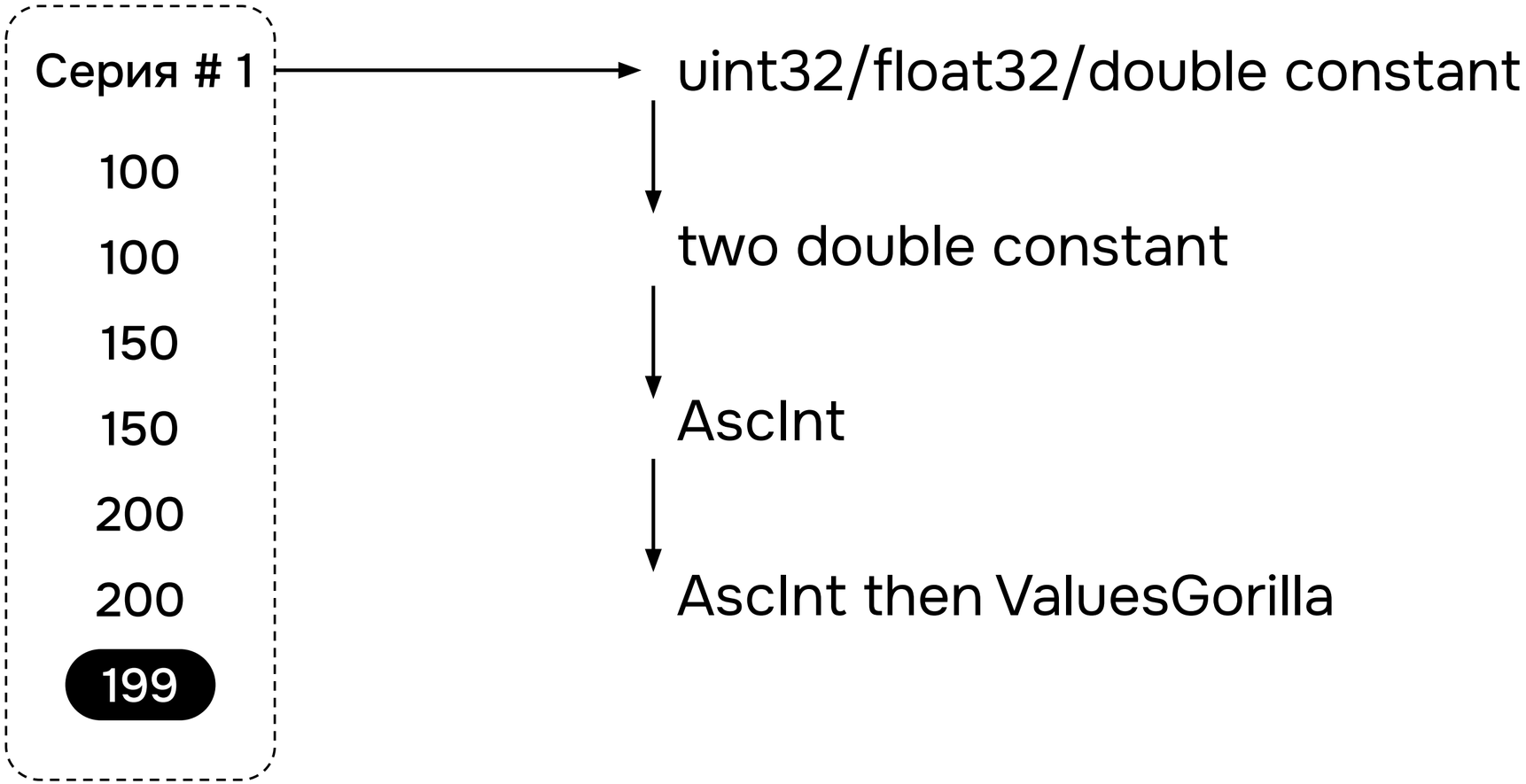

Серия # 1
100
100
uint32/float32/double constant

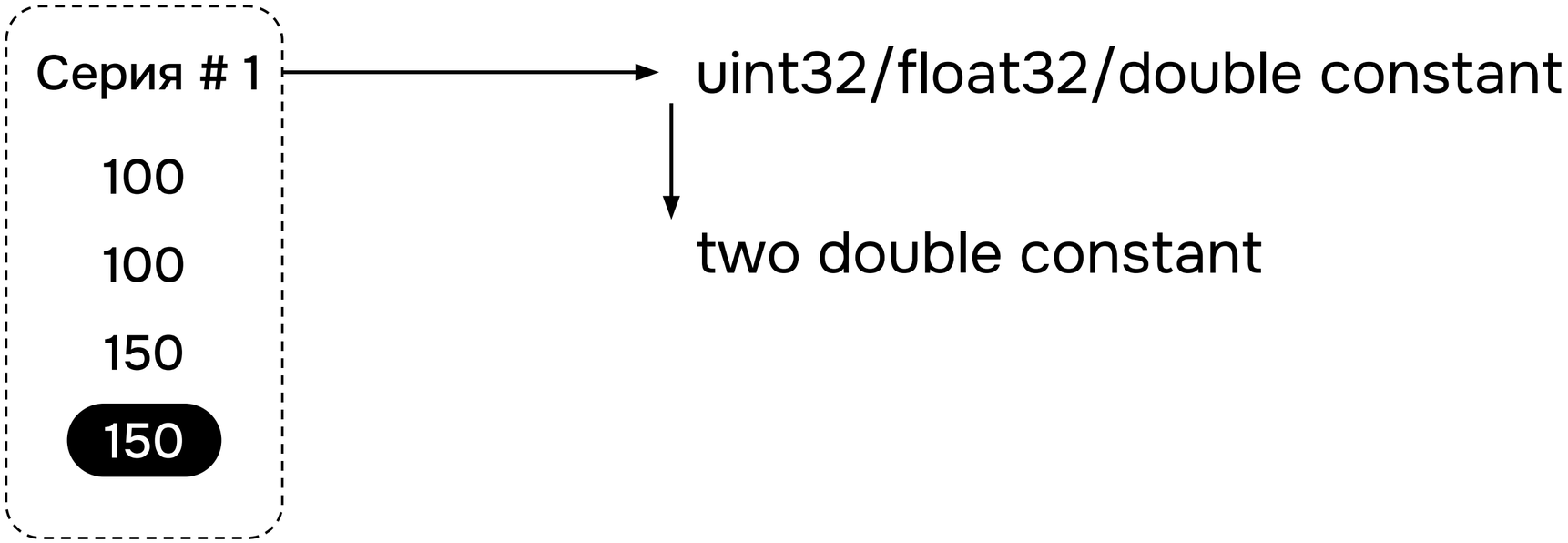

Серия # 1
100
100
150
uint32/float32/double constant
two double constant

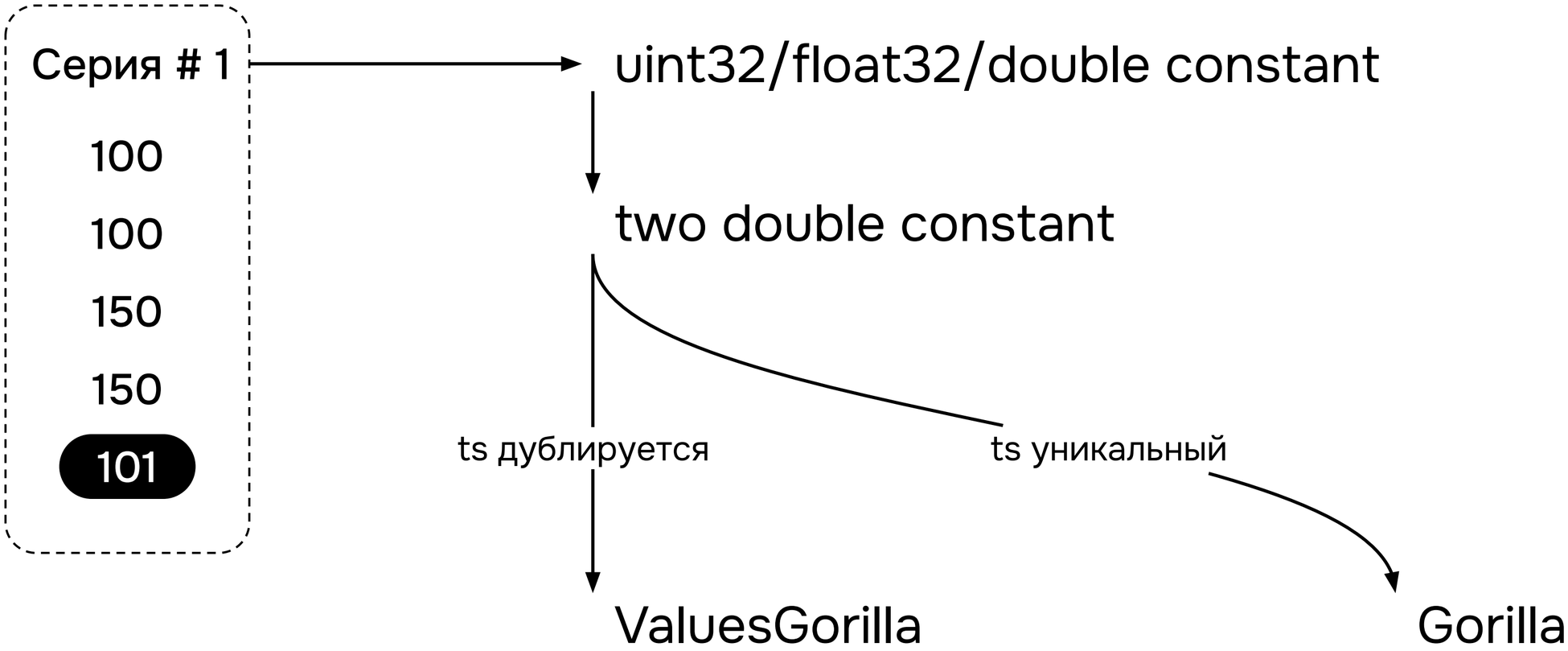

Серия # 1
100
100
150
150
uint32/float32/double constant
two double constant


Серия # 1
100
100
150
150
200
uint32/float32/double constant
two double constant
AscInt


Серия # 1
100
100
150
150
200
200
uint32/float32/double constant
two double constant
AscInt


Серия # 1
100
100
150
150
200
200
199
uint32/float32/double constant
two double constant
AscInt
AscInt then ValuesGorilla


Серия # 1
100
100
150
150
uint32/float32/double constant
two double constant


Серия # 1
100
100
150
150
101
uint32/float32/double constant
two double constant
ts дублируется
ts уникальный
ValuesGorilla
Gorilla

Бенчмаркинг

	Память	Время кодирования точки
Raw	3,78 ГБ	
Gorilla	512,21 МБ	26,62 ns
Timestamp storage	336,12 МБ	26,01 ns
Value encoders	151,31 МБ	32,56 ns
	Экономия памяти 184,81 МБ (~55 %)	Замедление кодирования точки на 6,55 ns

Распределение энкодеров

Энкодер	Кол-во серий	Память
uint32_t-константа	756 070	2,88 МБ
float32_t-константа	1 273	< 1 МБ
double-константа	29 651	< 1 МБ
two-double-константа	22 577	< 1 МБ
AscInt	324 180	65,66 МБ
AscInt then ValuesGorilla	8 426	2,08 МБ
ValuesGorilla	66 501	54,11 МБ
Gorilla	194	< 1 МБ

Компактные структуры

```
struct CompactStruct {
 const double value1; // 8 байт
 const double value2; // 8 байт
 const uint8_t value1_count; // 1 байт
};


sizeof(CompactStruct) == ??? байт
```

Компактные структуры

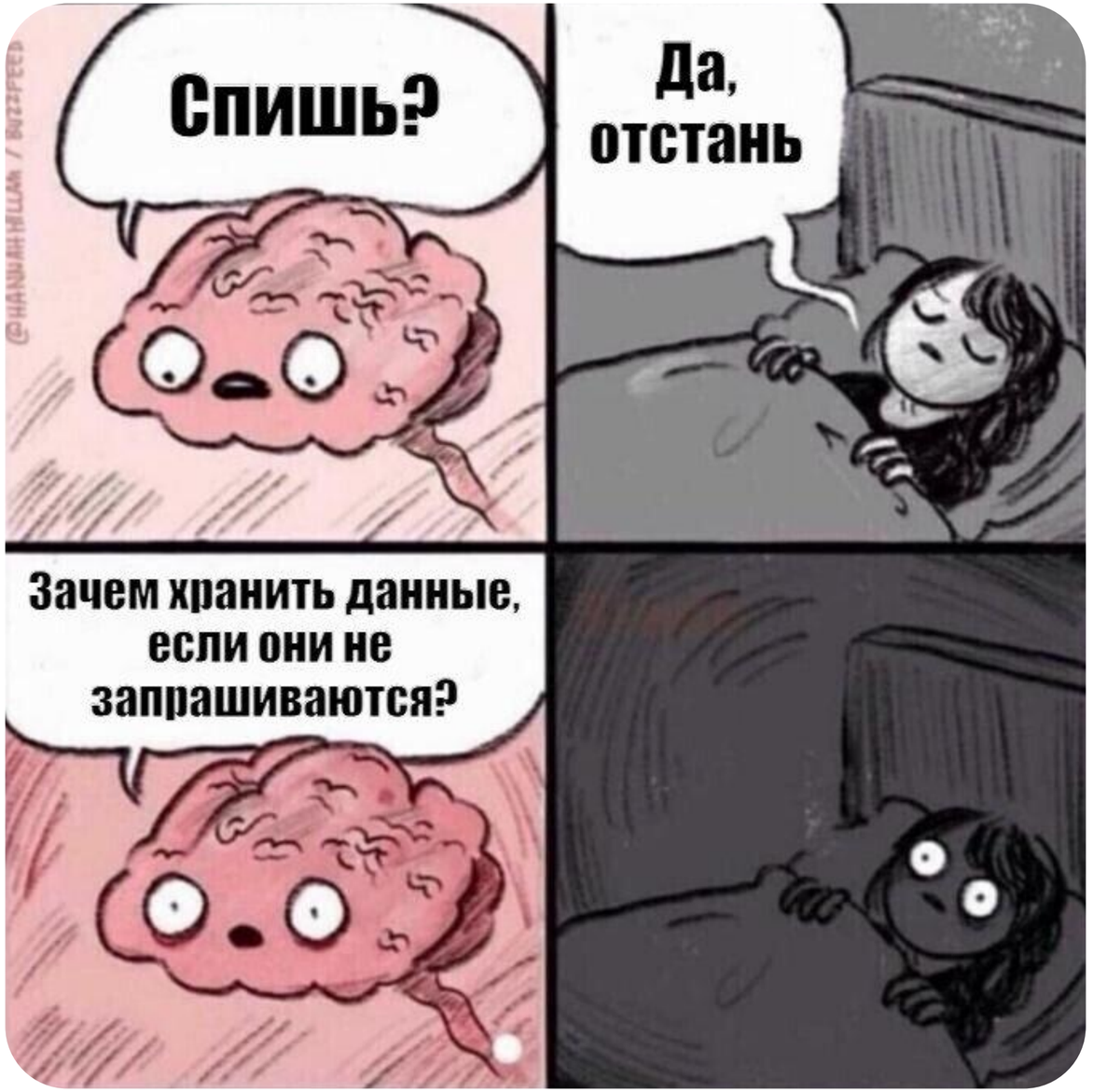
```
struct CompactStruct {
 const double value1; // 8 байт
 const double value2; // 8 байт
 const uint8_t value1_count; // 1 байт
 // padding 7 байт
};

sizeof(CompactStruct) == 24 байта
```

Компактные структуры

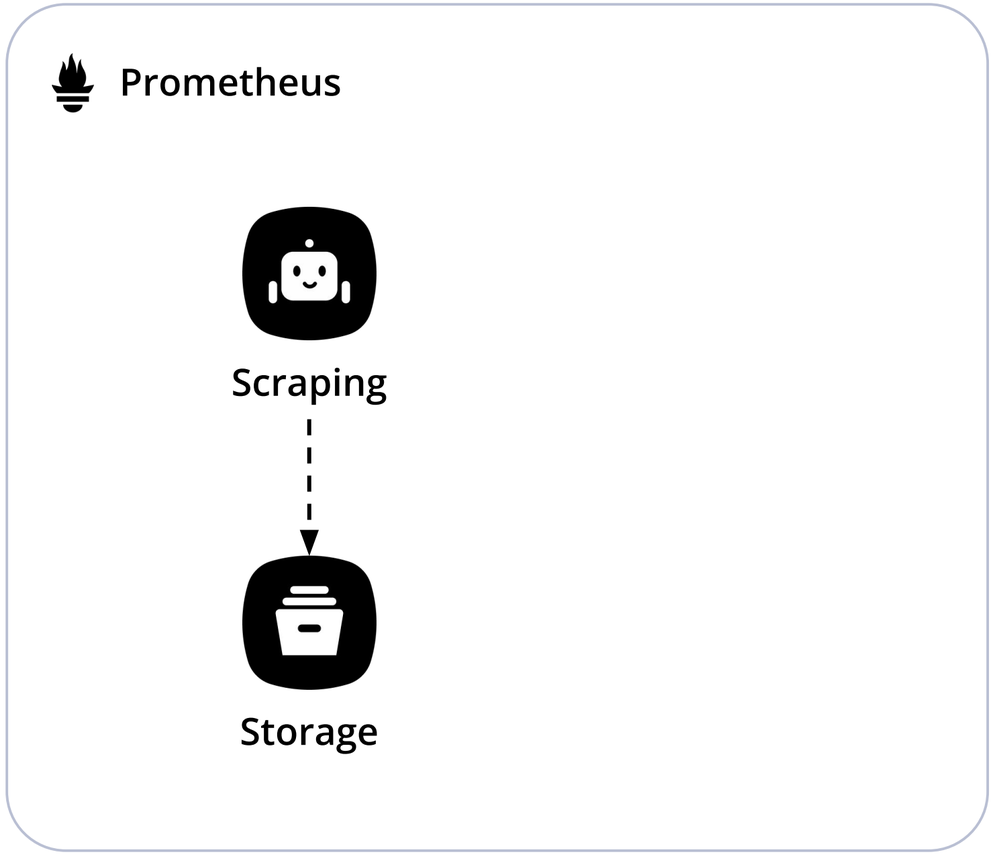
```
struct __attribute__((__packed__)) CompactStruct {
 const double value1; // 8 байт
 const double value2; // 8 байт
 const uint8_t value1_count; // 1 байт
};


sizeof(CompactStruct) == 17 байт
```

Реализация 4
Unused data unloading

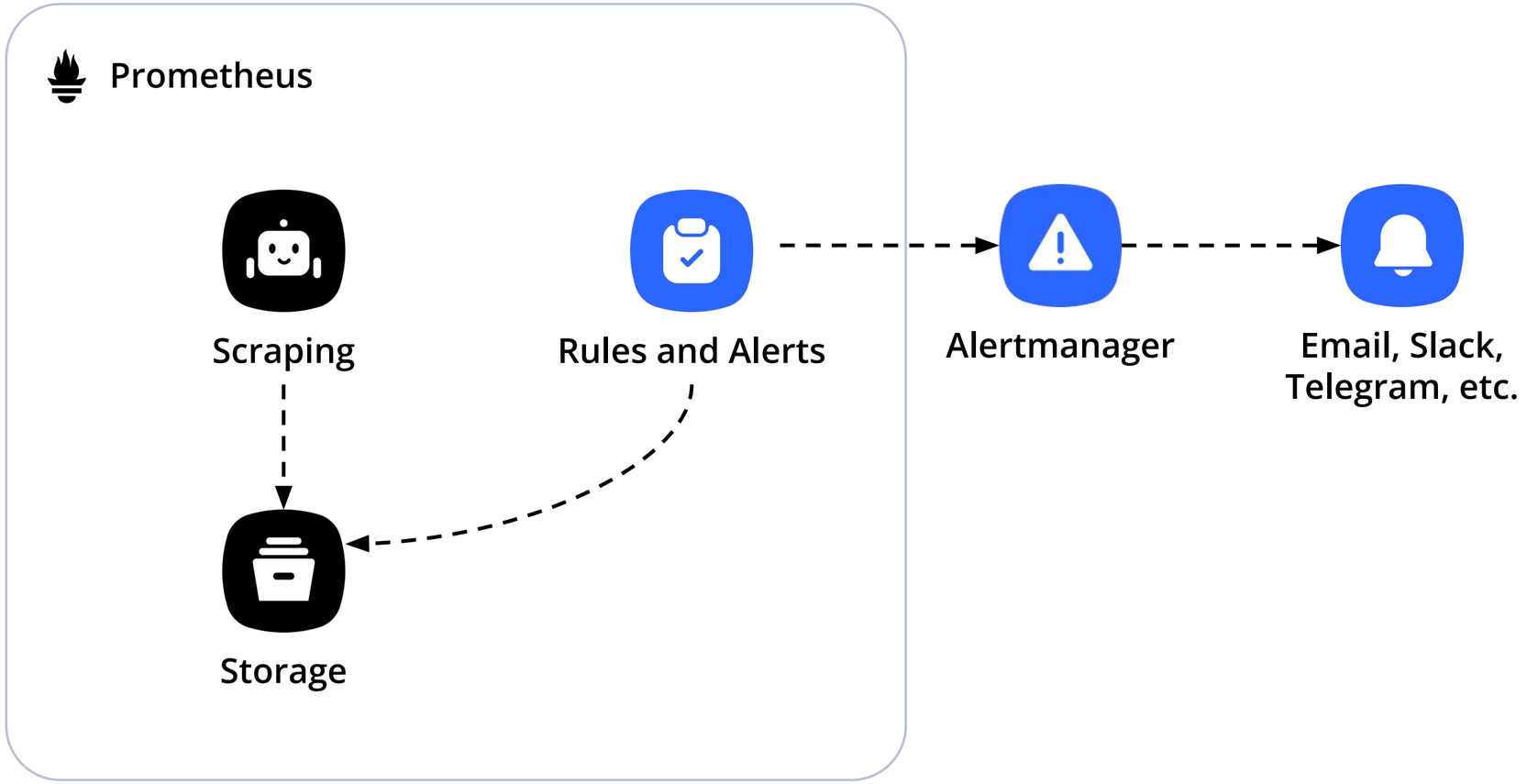
Deckhouse
ФЛАНТ


Спишь?
Да, отстань
Зачем хранить данные, если они не запрашиваются?

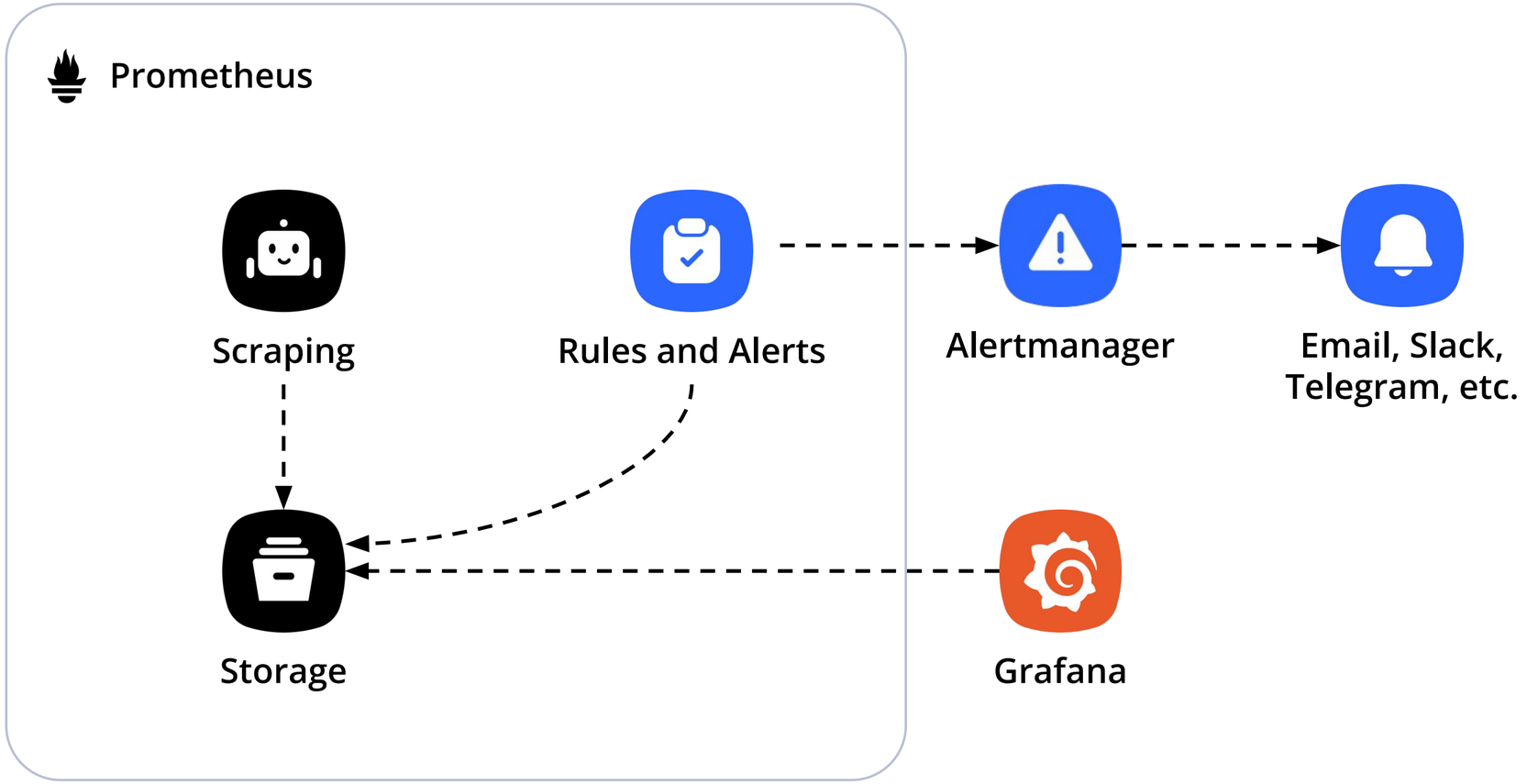
Кто запрашивает данные?



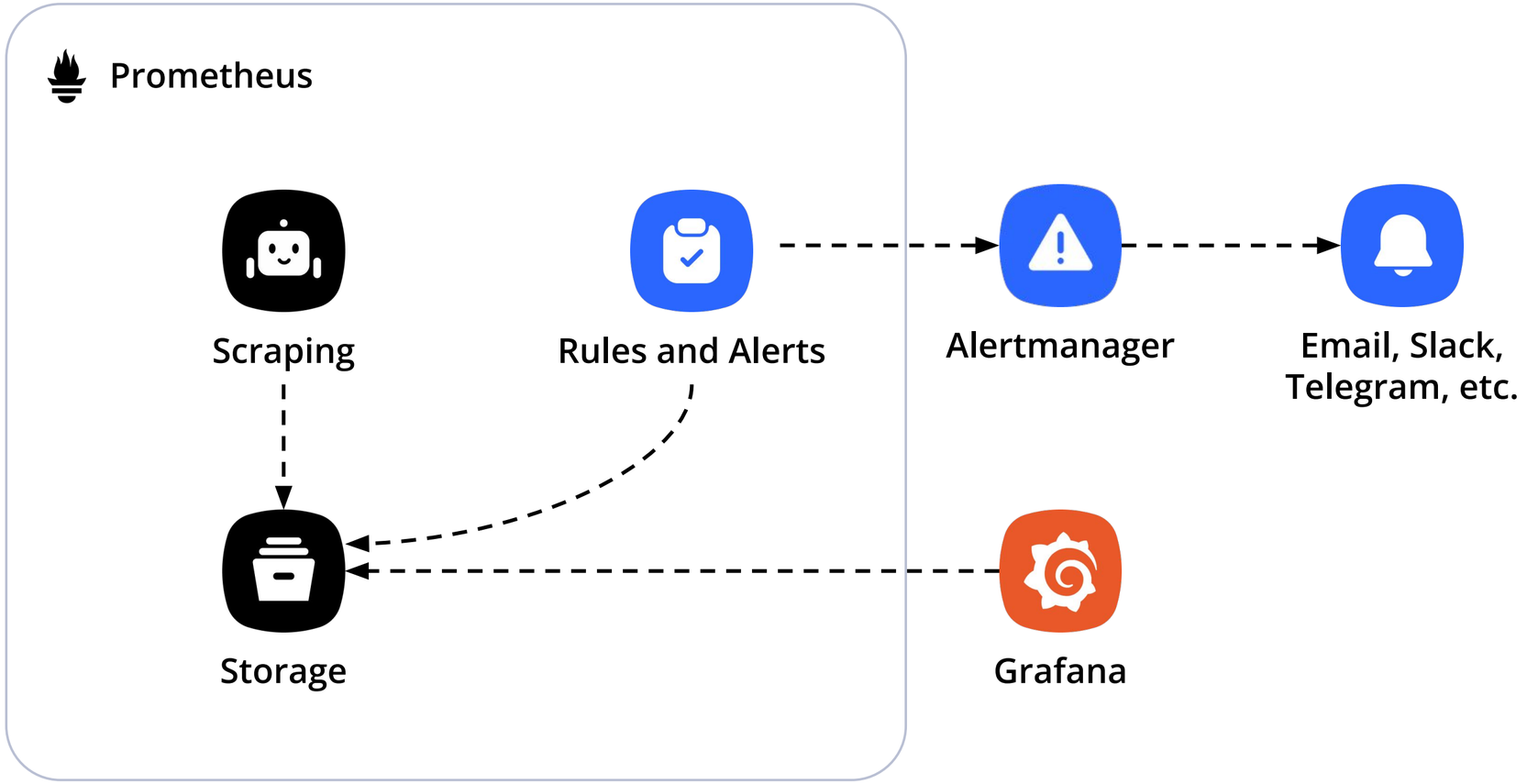
Кто запрашивает данные?



Кто запрашивает данные?



Кто запрашивает данные?



Запрашивается
только 6-8 % серий!

Распределение энкодеров

Энкодер	Память
uint32_t-константа	2,88 МБ
float32_t-константа	< 1 МБ
double-константа	< 1 МБ
two-double-константа	< 1 МБ
AscInt	65,66 МБ
AscInt then ValuesGorilla	2,08 МБ
ValuesGorilla	54,11 МБ
Gorilla	< 1 МБ

Распределение энкодеров

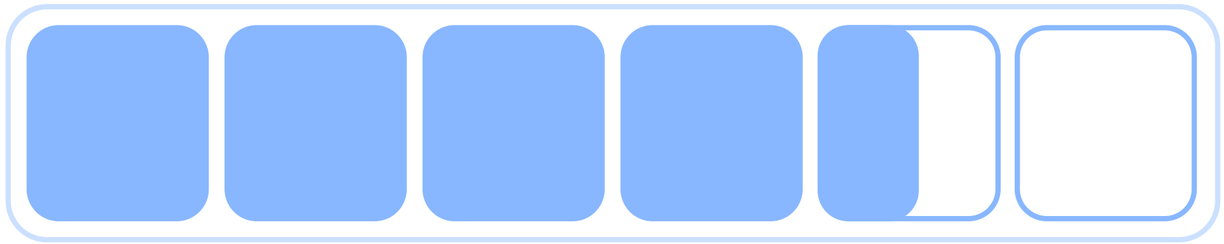
Энкодер	Память
uint32_t-константа	2,88 МБ
float32_t-константа	< 1 МБ
double-константа	< 1 МБ
two-double-константа	< 1 МБ
AscInt	65,66 МБ
AscInt then ValuesGorilla	2,08 МБ
ValuesGorilla	54,11 МБ
Gorilla	< 1 МБ

Алгоритм выгрузки

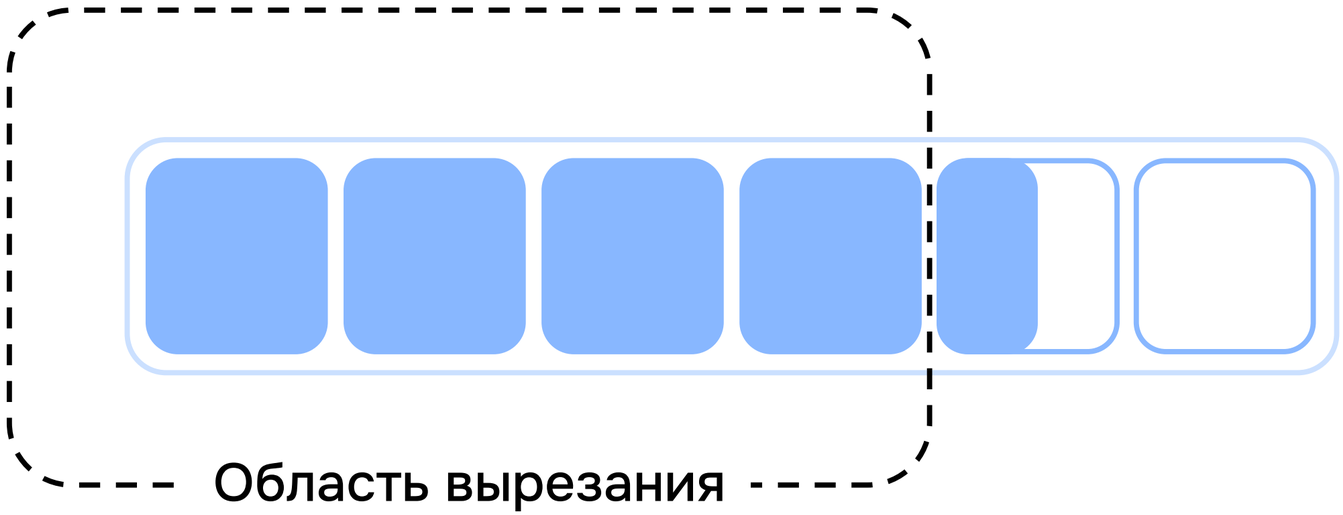
Каждые 5 минут:

- Создаем снапшот
- Выгружаем память энкодеров в снапшот
- Помечаем серию как выгруженную
- Сохраняем снапшот на диск

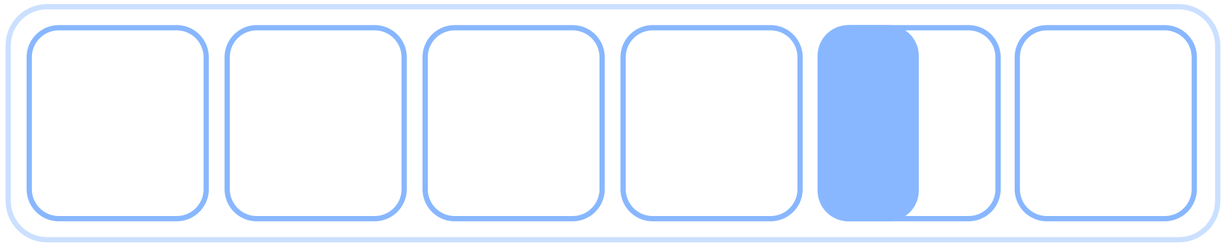
Выгрузка памяти энкодера



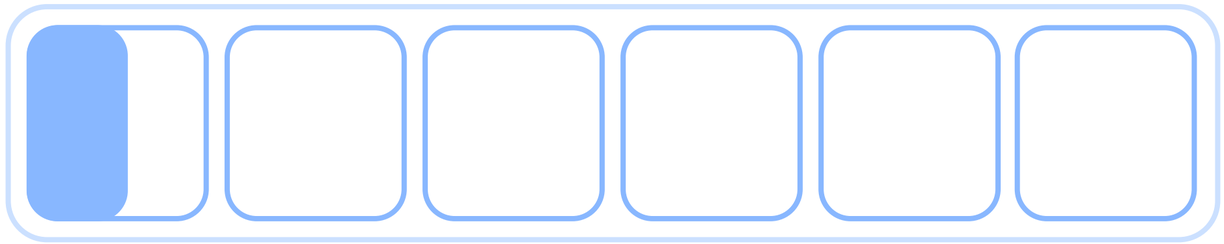
Выгрузка памяти энкодера



Выгрузка памяти энкодера



Выгрузка памяти энкодера



Бенчмаркинг

	Память	Время кодирования точки
Raw	3,78 ГБ	
Gorilla	512,21 МБ	26,62 ns
Timestamp storage	336,12 МБ	26,01 ns
Value encoders	151,31 МБ	32,56 ns
Выгрузка данных	56,38 МБ	28,50 ns
Каждая 10-я серия невыгружаемая	Экономия памяти 94,93 МБ (~63 %)	Ускорение кодирования точки на 4,06 ns

Алгоритм загрузки

При запросе выгруженной серии:

- Считываем снапшот с диска
- Загружаем память энкодера
- Помечаем серию как используемую

Заключение

Deckhouse
ФЛАНТ

Бенчмарки

	Память	Время кодирования точки
Raw	3,78 ГБ	
Gorilla	512,21 МБ	26,62 ns
Timestamp storage	336,12 МБ	26,01 ns
Value encoders	151,31 МБ	32,56 ns
Выгрузка данных	56,38 МБ	28,50 ns

Экономия памяти

455,84 МБ (~89 %)

Вывод

Важно понимать данные, которые хранишь

Мониторинг должен быть бесплатным!

TO BE CONTINUED...

deckhouse
user community

meetup/3

оцените доклад

Владимир Пустовалов

vladimir.pustovalov@flant.ru

@cherep_92

20:50 | открытая дискуссия