Проблемы разработки базы данных числовых временных рядов с нуля на Go

Александр Вишератин

старший научный сотрудник

Лаборатория «Когнитивные системы в промышленности»

Университет ИТМО, Санкт-Петербург

alexvish91@gmail.com

@visheratin

Зачем изобретать новую базу для временных рядов?

Нетривиальный набор требований от заказчика:

1. Хранение терабайтов временных рядов за десятки лет.

Пойдет любая современная зрелая база (PostgreSQL, Cassandra).

2. Поддержка операций поиска по условиям и извлечения данных.

Всё еще любая нормальная база данных.

3. 1 секунда на выполнение как поиска по всем данным, так и извлечение 450 тысяч записей из любого интервала.

Это сложнее, может подойти TimescaleDB или ClickHouse с материализованными представлениями.

4. Данные должны лежать в Amazon S3.

...

Как мы решили эту проблему?

Peregreen – бинарное хранилище, построенное с нуля с учетом специфики работы с временными рядами.

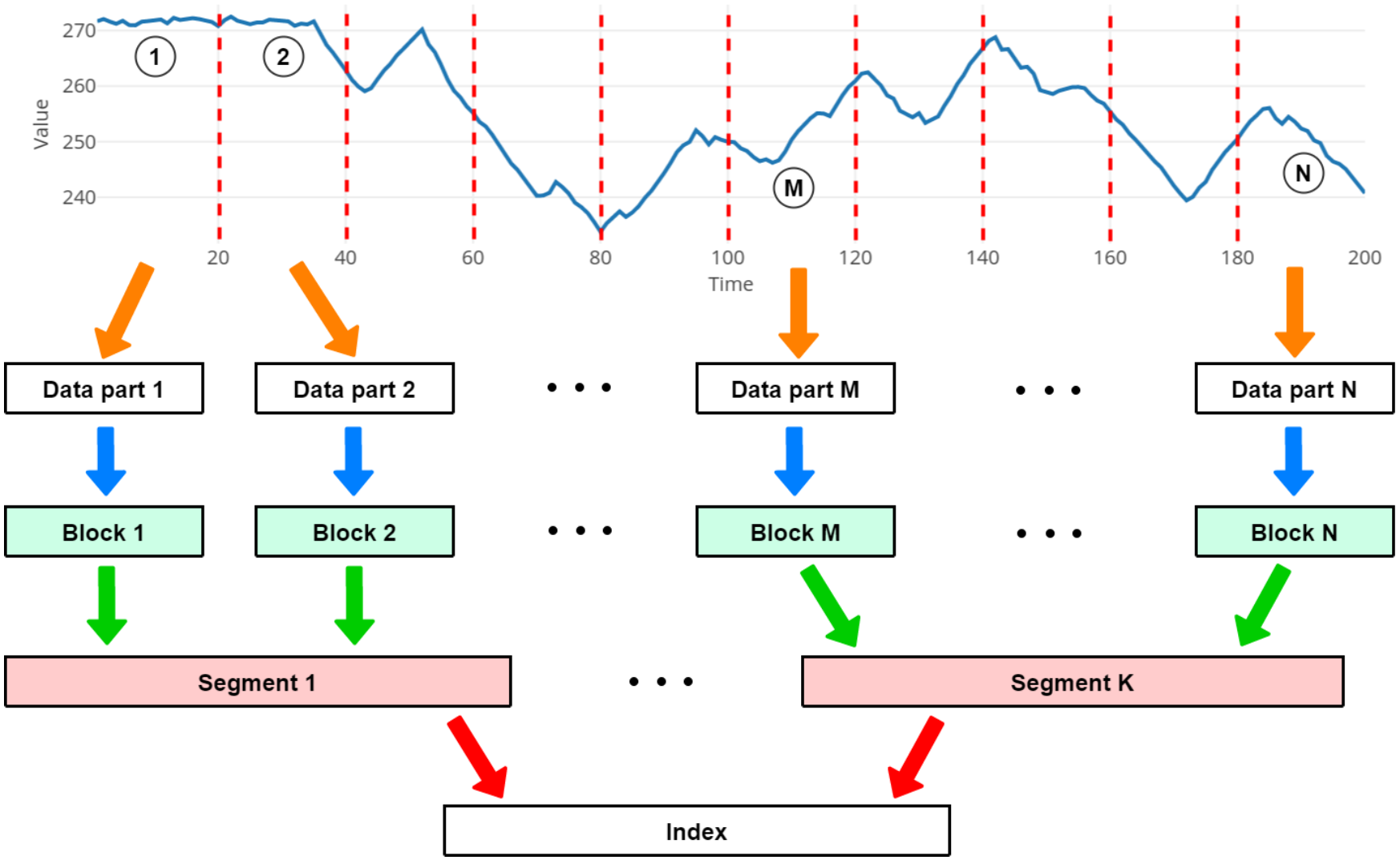
Ни таблиц, ни колонок, только сенсоры с временными метками и численными значениями.

Peregreen

Проблемы, с которыми мы столкнулись

1. Выбор структур данных для индекса. Что лучше – слайсы или деревья?
2. Формат хранения данных. Как хранить данные в базе – массив байтов, gob, колонки (Parquet)?
3. Поддержка множества типов данных. Красивая имплементация через интерфейсы или композитный тип с громоздкими switch’ами по всей кодовой базе?

Структура индекса в Peregreen



Data part	
T1	V1
T2	V2
...	
Tn	Vn

Block
Data size
Elements count
Metrics
Compressed

Segment
ID
Start time
Metrics
Checksum
Blocks

Index
ID
Data type
Segment interval
Block interval
Location
Segments

Как организовать индекс?


```
type Block struct {
        Size  int
        ElNum int
        Min   float64
        Max   float64
}
```

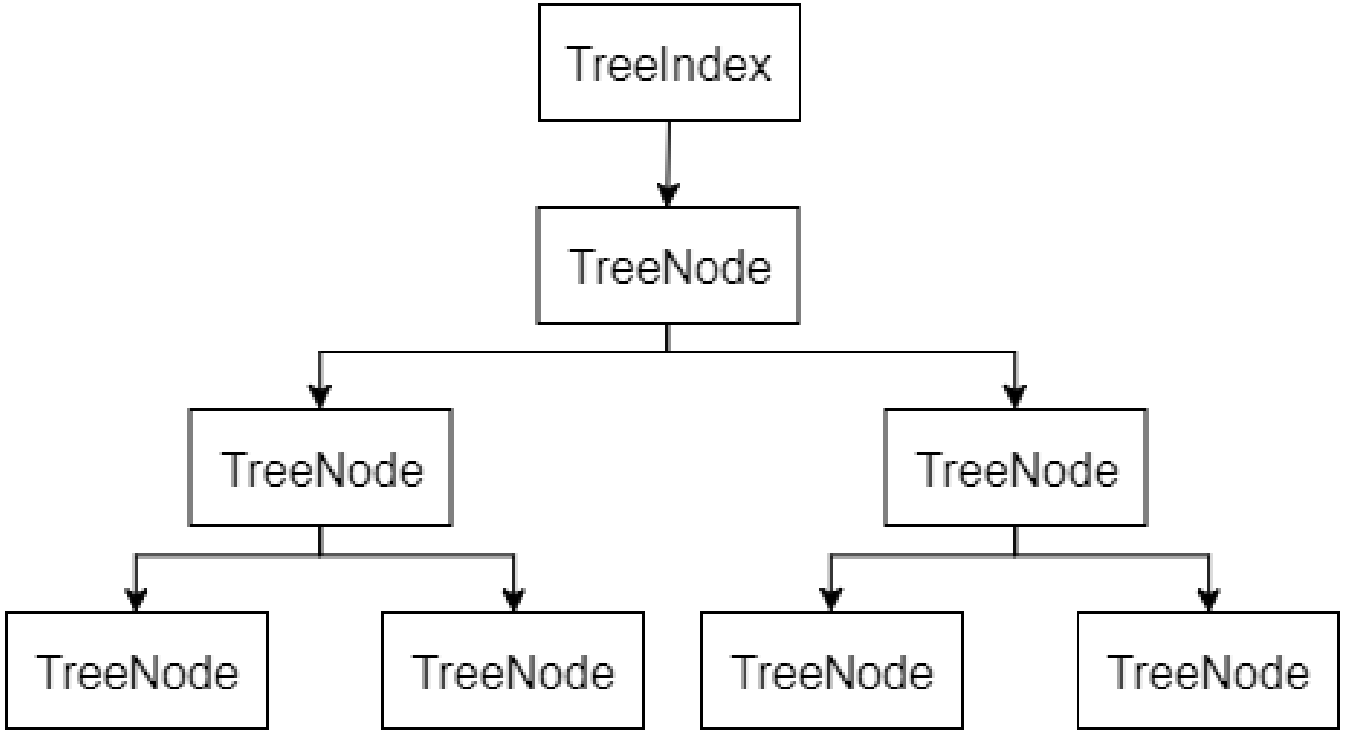
Назначение блоков: фильтрация по заданным условиям – извлечение из индекса всех блоков, которые удовлетворяют критериям клиента.

Решение: деревья. Даже простое бинарное дерево даст $O(\log_2 N)$ сложность поиска.

Простое бинарное дерево

```
type TreeIndex struct {
        ID   string
        Root *TreeNode
}
```

```
type TreeNode struct {
        LeftPart  *TreeNode
        RightPart *TreeNode
        Min       float64
        Max       float64
        Block     data.Block
}
```



Поиск в дереве


```
func (node TreeNode) search(min float64, max float64) []data.Block {
        if filter(node.Min, node.Max, min, max) {
                if node.Block.Size != 0 {
                        return []data.Block{node.Block}
                }
                lp := node.LeftPart.search(min, max)
                rp := node.RightPart.search(min, max)
                return append(lp, rp...)
        }
        return nil
}
```

Создание новых объектов с указателями
Выделение памяти
Сборка мусора

Бенчмарк простого дерева (поиск)

	Время выполнения (мкс)	Количество аллокаций
1000 элементов	35	391
10000 элементов	673	4025
100000 элементов	7370	39922
1000000 элементов	151296	400223

Постоянное создание объектов и выделение памяти снижают производительность.

Решение: упразднить выделение памяти.

Бинарное дерево посложнее


```
type AdvTreeIndex struct {
        ID     string
        Length int
        Root   *AdvTreeNode
}
```

При добавлении данных увеличиваем счетчик количества элементов

```
type AdvTreeNode struct {
        LeftPart  *AdvTreeNode
        RightPart *AdvTreeNode
        Min       float64
        Max       float64
        Block     data.Block
}
```

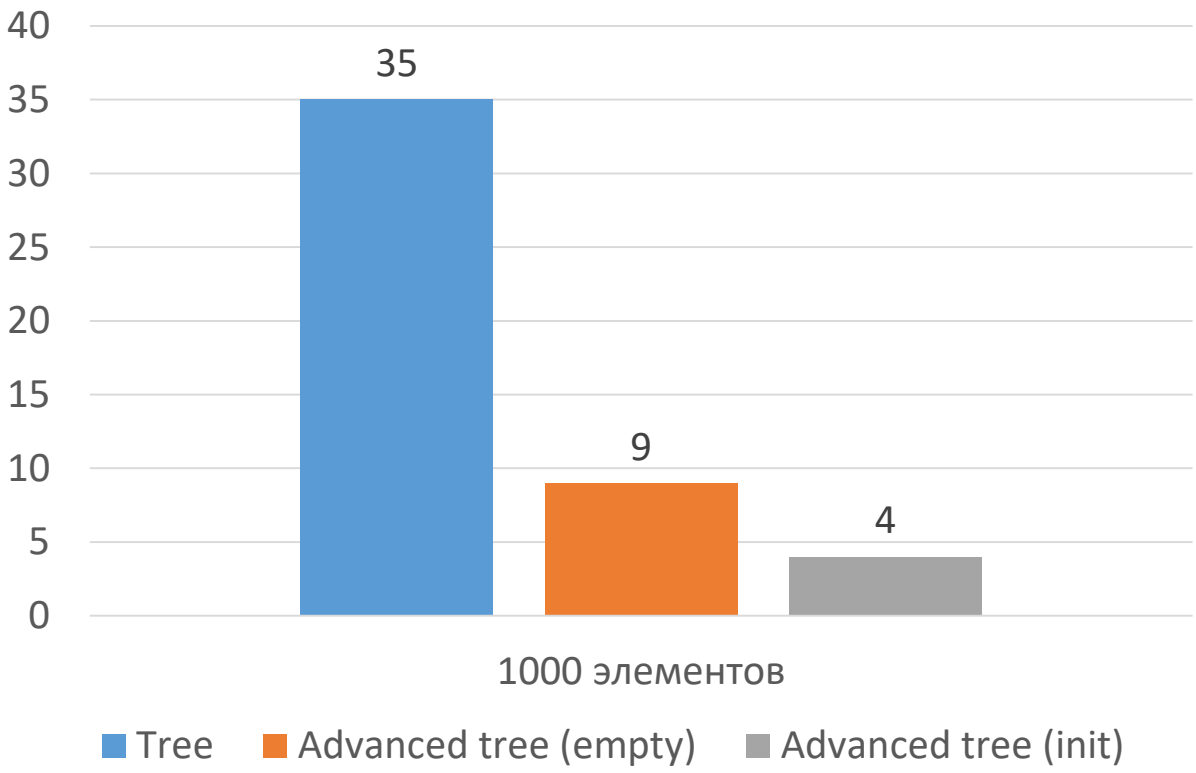
Поиск в новом дереве

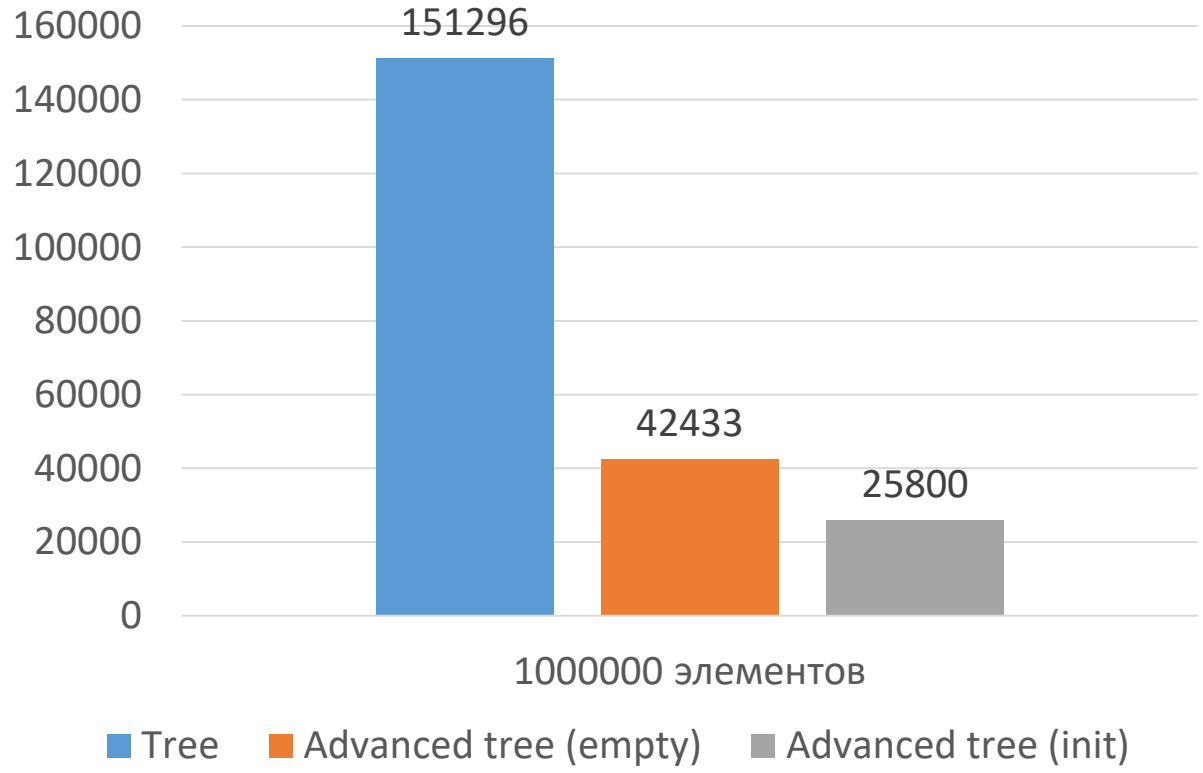
```
func (node AdvTreeNode) search(min float64, max float64, res []data.Block) []data.Block {
        if filter(node.Min, node.Max, min, max) {
                if node.Block.Size != 0 {
                        res = append(res, node.Block)
                        return res
                }
                res = node.LeftPart.search(min, max, res)
                res = node.RightPart.search(min, max, res)
        }
        return res
}
```

Можно инициализировать слайс заранее, используя длину индекса в качестве ёмкости.

Новые участки памяти не выделяются, всё дописывается в уже существующую

Бенчмарк нового дерева (время, мкс)

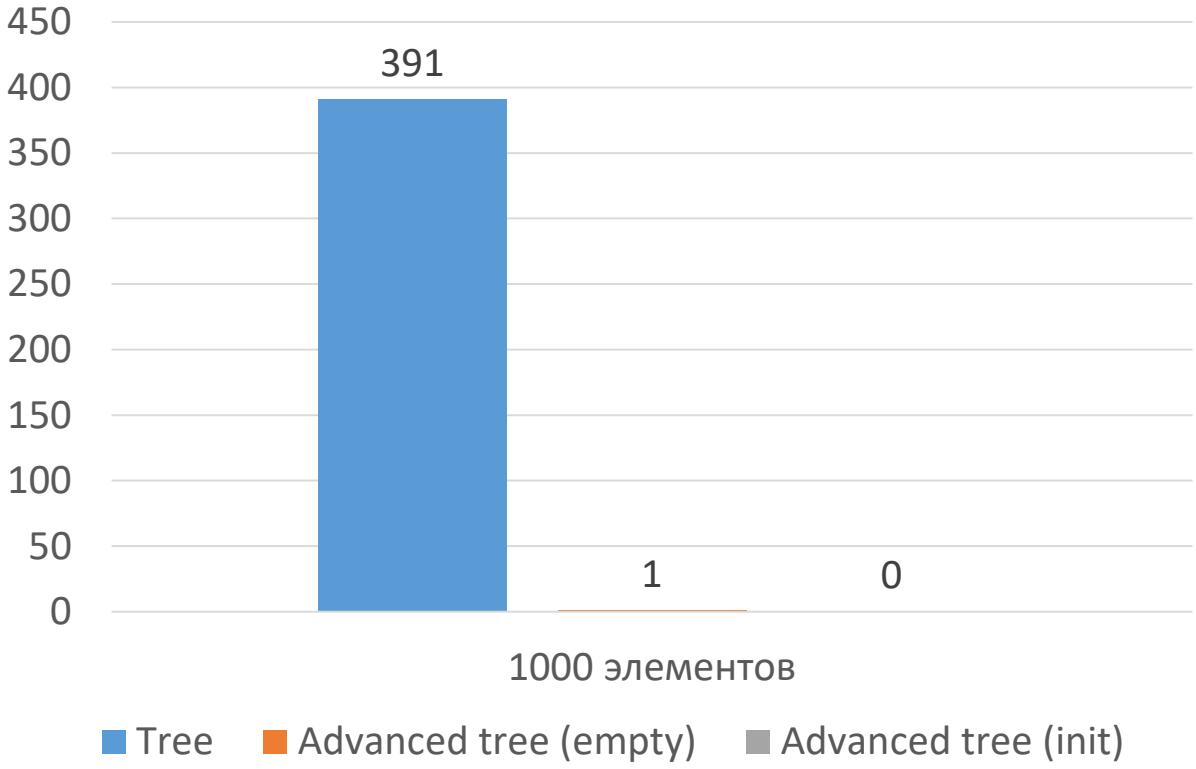


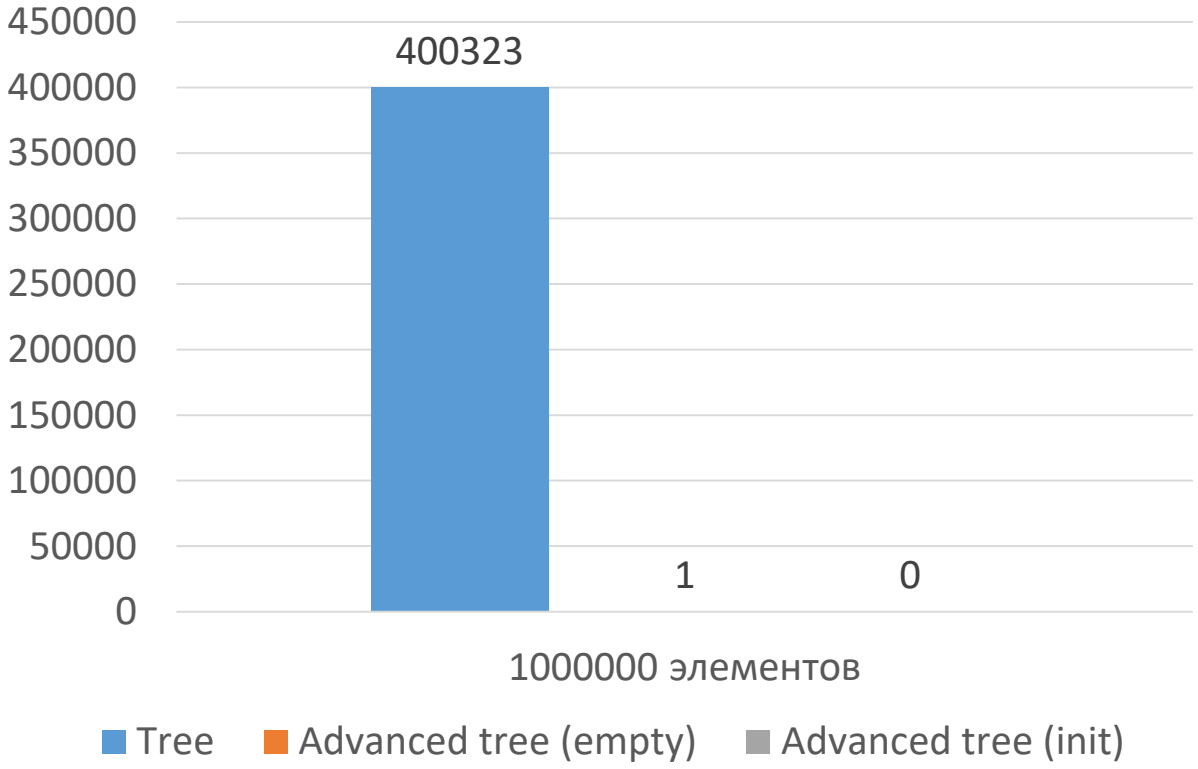


init – слайс создавался заранее, empty – слайс создавался в бенчмарке.

Скорость выросла в 4 раза, поиск занимает почти половину общего времени выполнения запроса.

Бенчмарк нового дерева (аллокации)





Такое дерево получается очень глубоким. Как можно улучшить положение дел?

Хранить несколько элементов вместо одного.

(Что-то похожее на)B-дерево


```
type BTreeNode struct {
        LeftPart  *BTreeNode
        RightPart *BTreeNode
        Min       float64
        Max       float64
        Blocks    []data.Block
}
```

Слайс блоков вместо одного

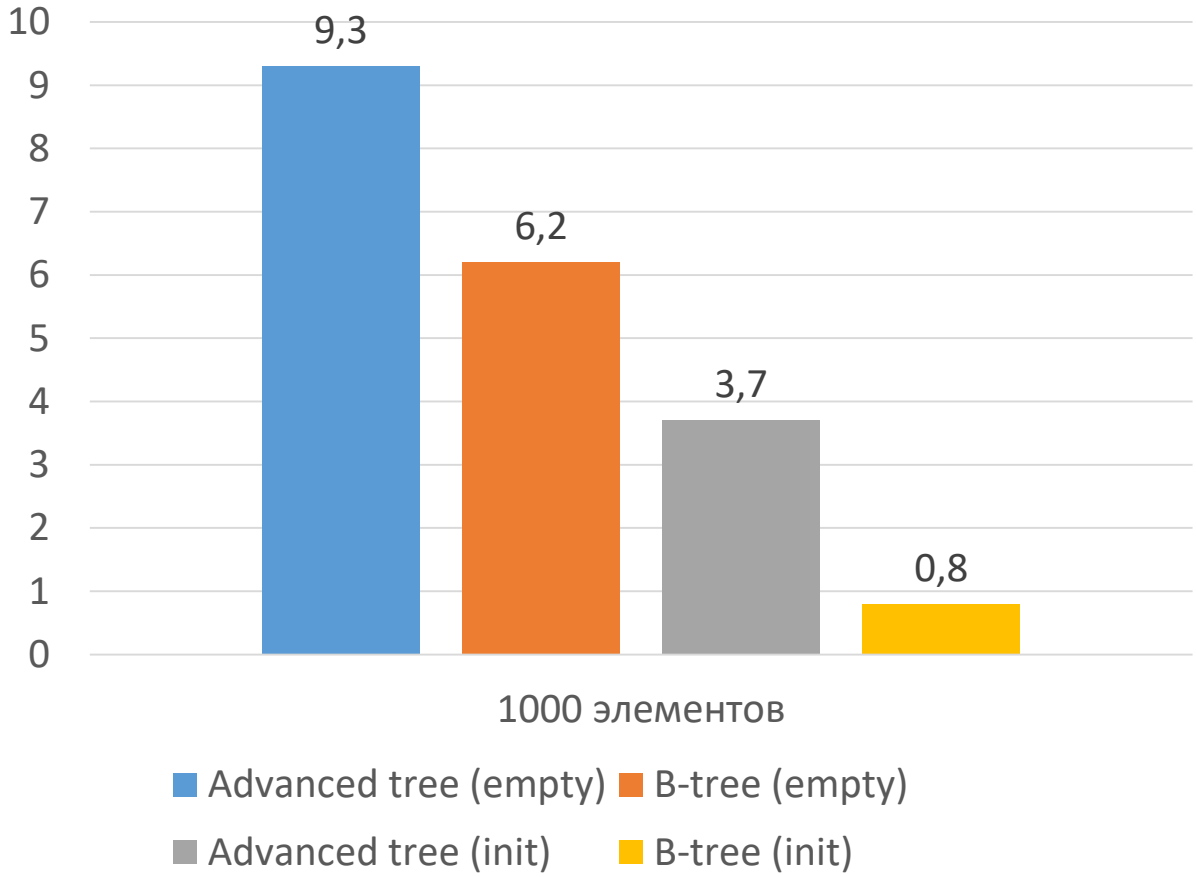
В отличие от других реализаций, данные в не-листовых нодах хранятся

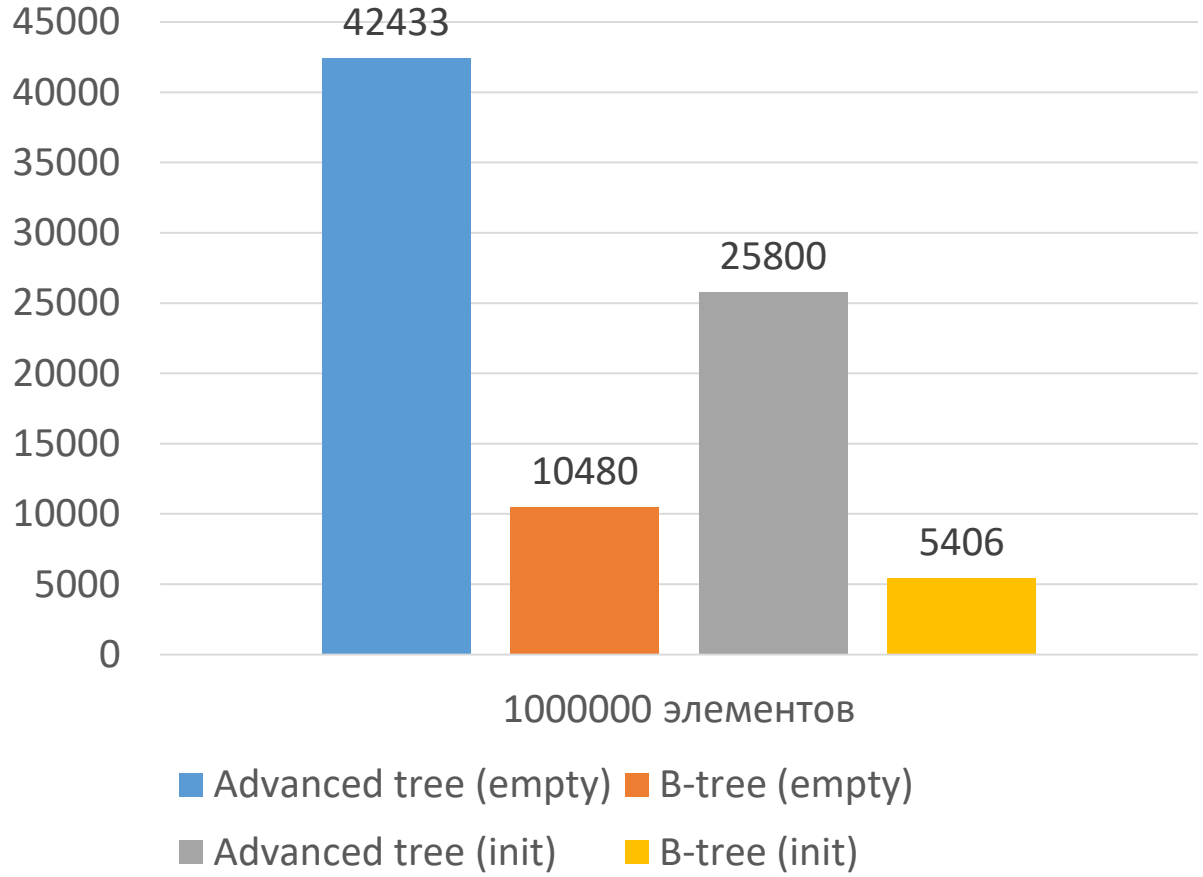
Поиск в псевдо-B-дереве

```
func (node BTreeNode) search(min float64, max float64, res []data.Block) []data.Block {
        if filter(node.Min, node.Max, min, max) {
                for _, b := range node.Blocks {
                        if filter(b.Min, b.Max, min, max) {
                                res = append(res, b)
                        }
                }
                res = node.LeftPart.search(min, max, res)
                res = node.RightPart.search(min, max, res)
        }
        return res
}
```

Фильтруем блоки, потому что не все могут подходить под условия

Бенчмарк псевдо-B-дерева (время, мкс)





Ускорение поиска еще в 4 раза, поиск теперь занимает половину времени. Меньше глубина дерева, меньше рекурсия, меньше вызовов функций.

Индекс на слайсах


```
type SliceIndex struct {
        ID     string
        Blocks []data.Block
}
```

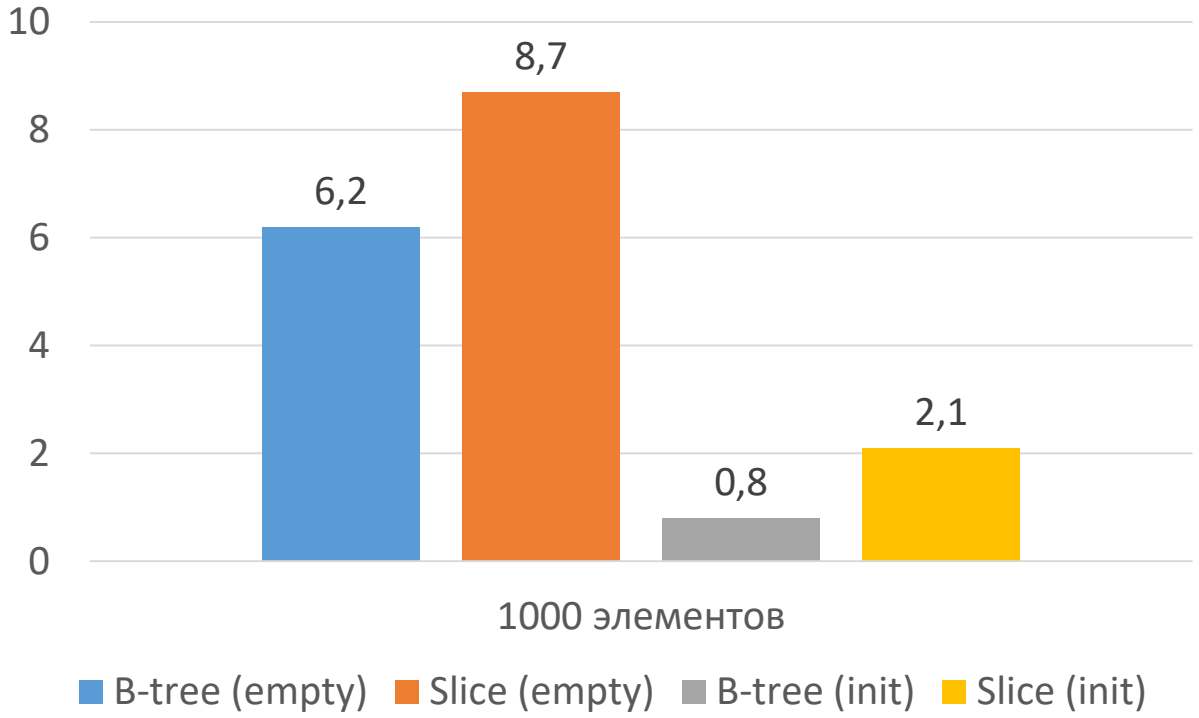
Очень простая структура

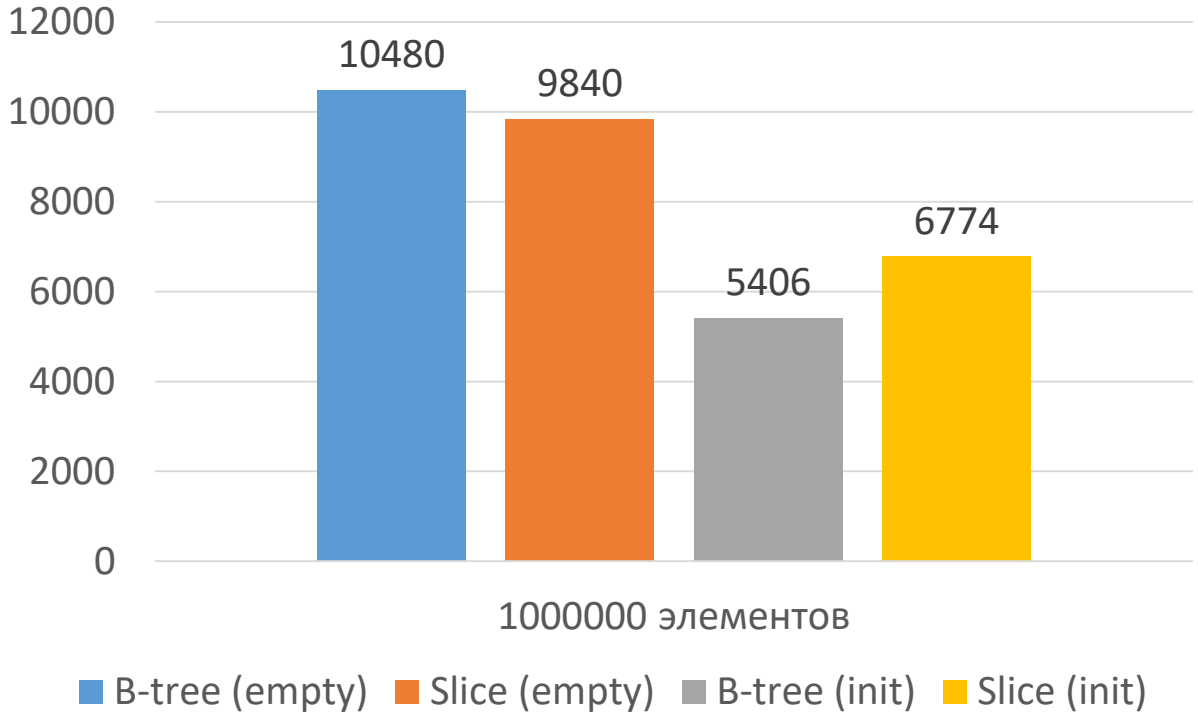
Поиск

```
func (idx *SliceIndex) Search(min float64, max float64, res []data.Block) []data.Block {
        for _, b := range idx.Blocks {
                if filter(b.Min, b.Max, min, max) {
                        res = append(res, b)
                }
        }
        return res
}
```

Поиск также очень простой

Бенчмарк слайса (время, мкс)





Внезапно он такой же быстрый, как самое лучшее дерево.

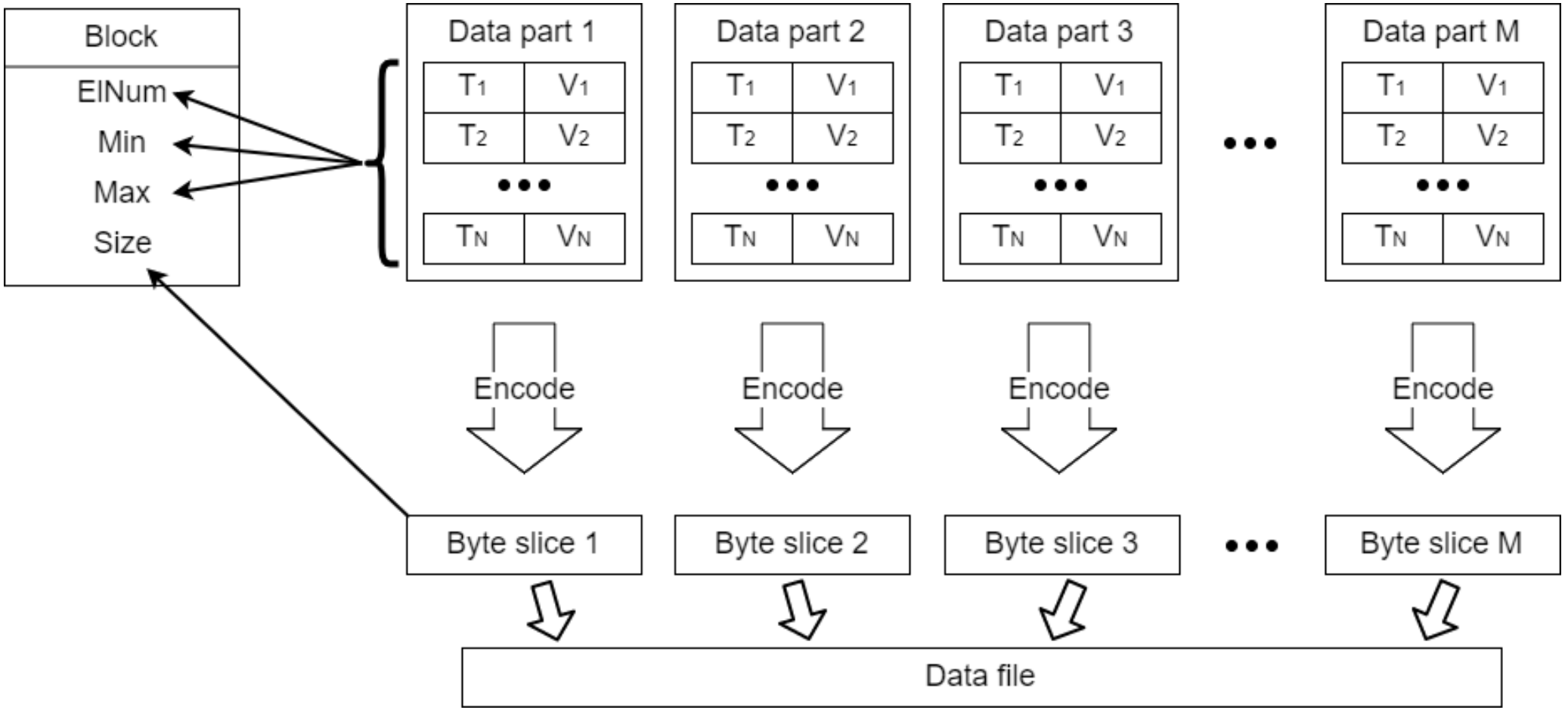
Почему? Нет дерева, нет рекурсии, нет вызовов функций.

Вывод № 1

При проектировании решений стоит начинать с простых вариантов и усложнять по мере необходимости.

Если простого решения достаточно для удовлетворения ваших требований – остановитесь.

Схема хранения данных в Peregreen



Преимущества схемы хранения

1. Возможность разделения данных на блоки произвольной длины.
2. Возможность хранить статистические метрики данных в индексе.
3. Работать с одним большим файлом проще, чем с тысячами маленьких.

Внутреннее представление данных

Нужен способ сериализации данных, который позволит быстро преобразовывать данные во внутреннее представление и обратно.

Варианты:

1. gob – доступен из коробки, прост, довольно быстр.
2. Parquet – популярный формат с доказанной эффективностью, доступно сжатие данных.
3. свой формат – возможно будет быстрее, много подводных камней, долго разрабатывать.

Интерфейс хранилища данных

```
type Store interface {
        Insert(dataParts [][]data.Element) ([]data.Block, error)
        Read(blockIds []int, blockSizes []int,
                blockNums []int, offset int64) ([]data.Element, error)
}
```

Загрузка частей данных в хранилище

Извлечение данных по метаинформации

GobStore

Insert

```
buf := bytes.NewBuffer(nil)
err := gob.NewEncoder(buf).Encode(d)
b := buf.Bytes()
```

Read

```
buf := bytes.NewBuffer(rawData)
err = gob.NewDecoder(buf).Decode(&allData)
res = append(res, allData...)
```

1. Простая и понятная реализация.
2. Минимальное количество усилий.

Но как оно работает внутри?

Профилирование в Go

Одна волшебная строка в функции main:

defer profile.Start().Stop()

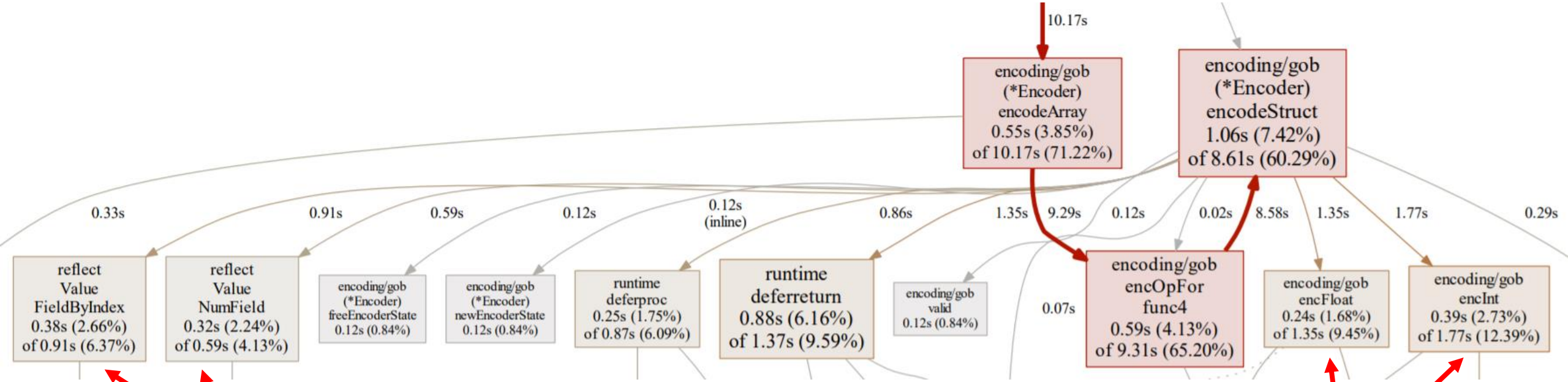
Позволяет получить исчерпывающую информацию о процессе выполнения программы.

Сам пакет – https://github.com/pkg/profile

Профиль работы GobStore

	Количество блоков
GobStore	74

Профиль работы GobStore



Определение значений через reflect

Кодирование непосредственно значений

ParquetStore

Insert

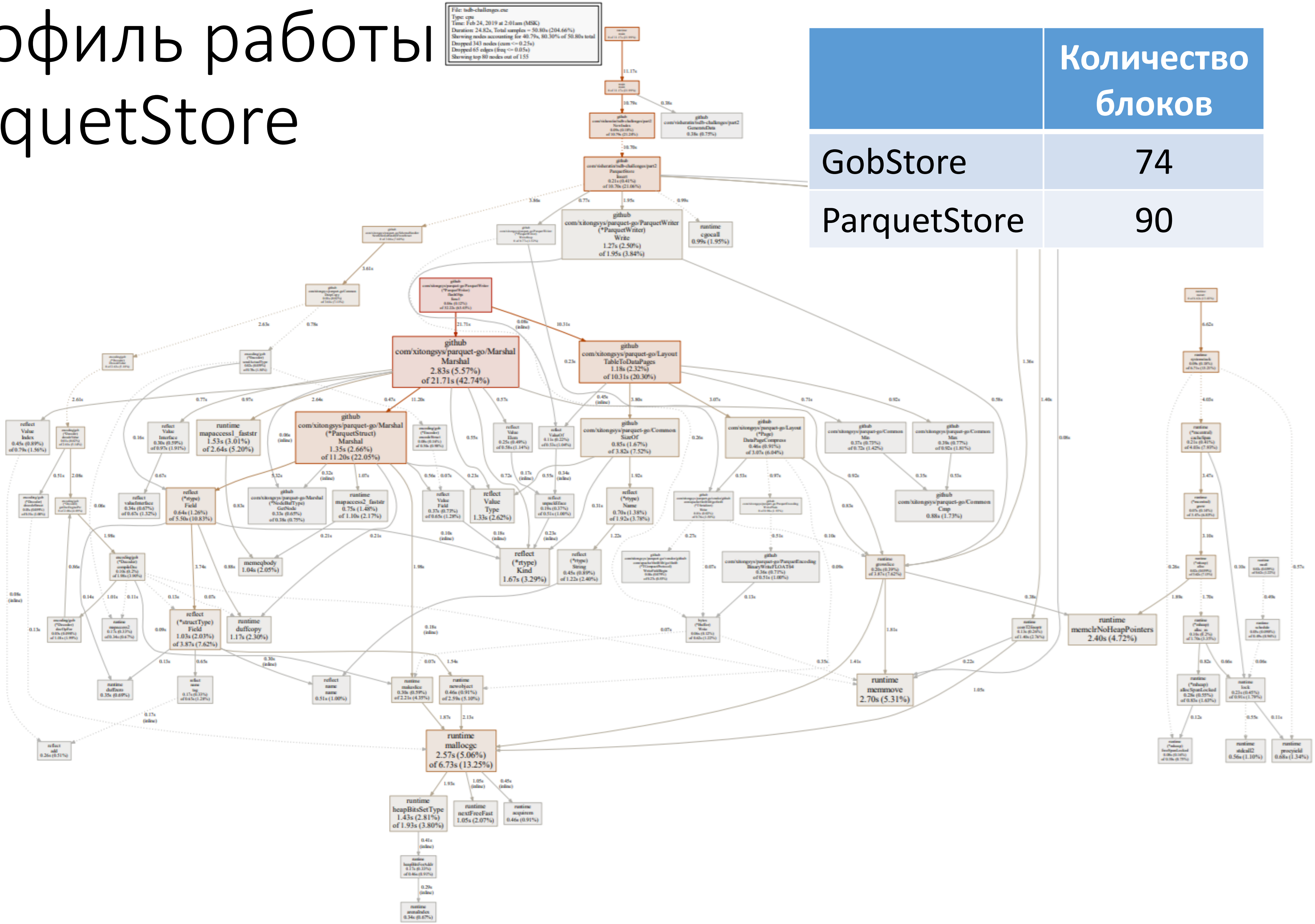
```
fw, err := ParquetFile.NewBufferFile(nil)
pw, err := ParquetWriter.NewParquetWriter(fw, new(data.Element), 4)
for _, el := range d {
    err = pw.Write(el); err != nil
}
b := fw.(ParquetFile.BufferFile).Bytes()
```

Read

```
fw, err := ParquetFile.NewBufferFile(rawData)
pr, err := ParquetReader.NewParquetReader(fw, new(data.Element), 4)
allData := make([]data.Element, blockNums[i])
err = pr.Read(&allData)
res = append(res, allData...)
```

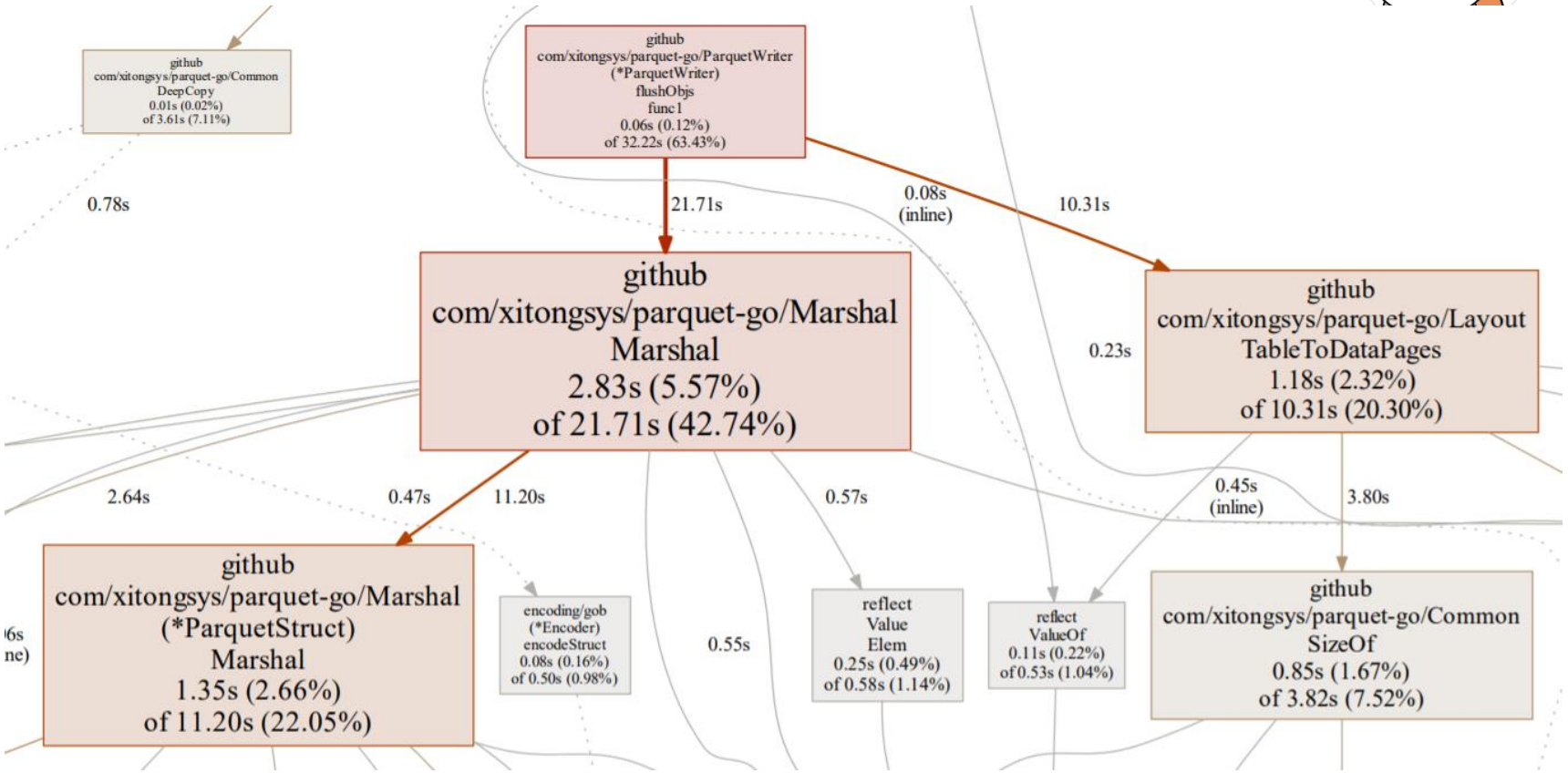
Пакет для работы с Parquet – https://github.com/xitongsys/parquet-go/

Профиль работы ParquetStore



	Количество блоков
GobStore	74
ParquetStore	90

Профиль работы ParquetStore



Большая часть времени – маршаллинг объектов.

Много времени тратится на обработку данных через пакет reflect.

Причина – пакет работает с интерфейсами.

ProtoStore. Схема данных


```
message ProtoElement {
  required int64 Timestamp = 1 [(gogoproto.nullable) = false];
  required double Value = 2 [(gogoproto.nullable) = false];
}

message ProtoElements {
  repeated ProtoElement Data = 1 [(gogoproto.nullable) = false];
}
```

Делаем так, чтобы поля не содержали указателей

Пакет для работы с Protobuf – https://github.com/gogo/protobuf

ProtoStore. Операции

Insert

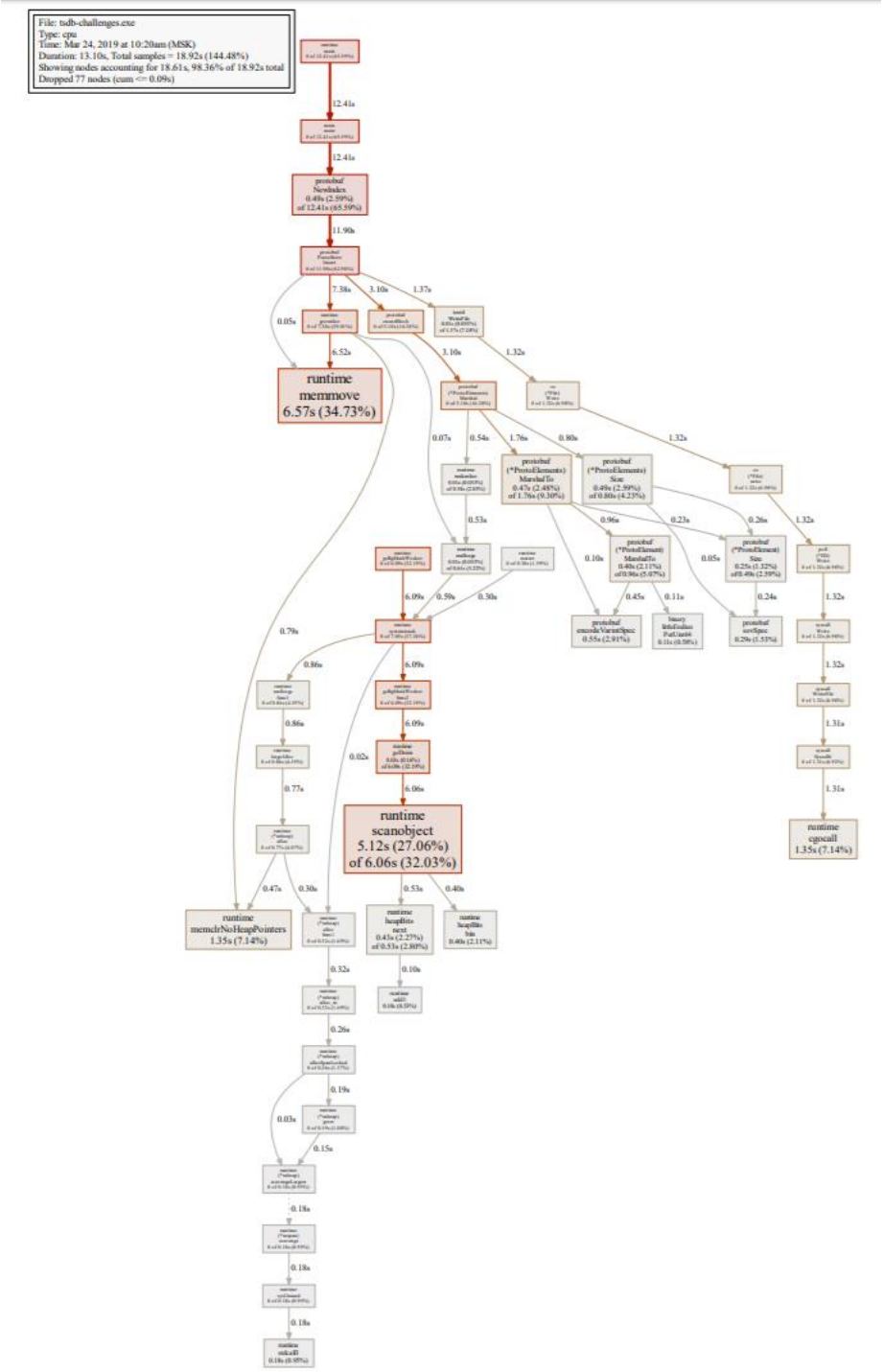
```
blockBuf, err = d.Marshal()
buf = append(buf, bd...)
```

Read

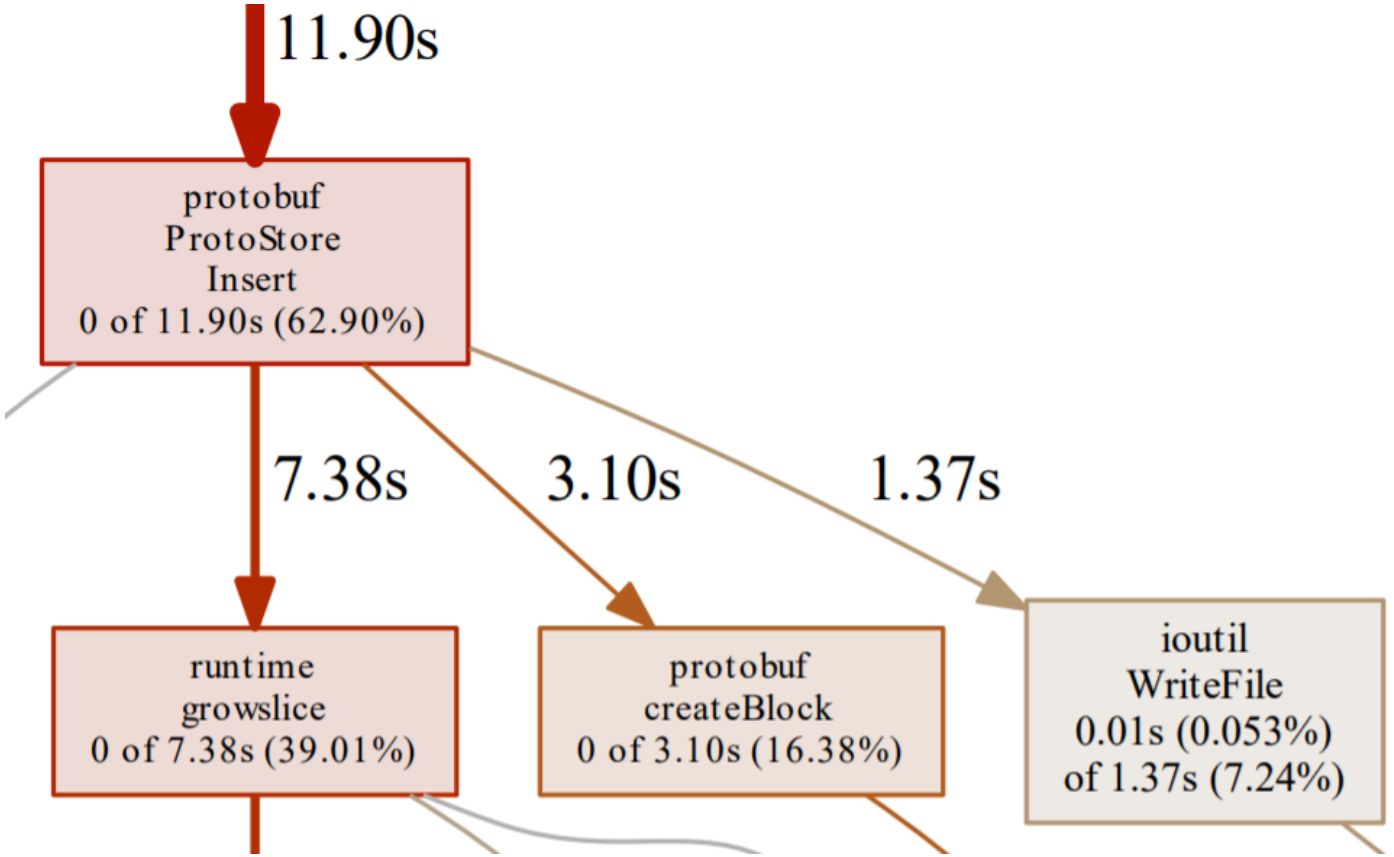
```
var d ProtoElements
err = d.Unmarshal(rawData)
res.Data = append(res.Data, d.Data...)
```

Профиль работы ProtoStore

	Количество блоков
GobStore	74
ParquetStore	90
ProtoStore	46



Профиль работы ProtoStore



Теперь маршаллинг быстрый и большую часть времени занимает выделение памяти.

BinaryStore

Ручная сериализация данных – преобразовываем значения в байты с помощью пакета binary.

Для сжатия данных – delta encoding (https://en.wikipedia.org/wiki/Delta_encoding).

Заранее рассчитываем объем преобразованных данных и создаем итоговый массив.

BinaryStore. Insert


```
bl := 12 * len(d)
buf := make([]byte, bl)
l := len(d)
for i := 0; i < l; i++ {
        t = d[i].Timestamp
        ts = uint32(t - tsp)
        binary.LittleEndian.PutUint32(buf[c:c+4], ts)
        f64 = math.Float64bits(d[i].Value)
        f64d = f64 - f64p
        binary.LittleEndian.PutUint64(buf[c:c+8], uint64(f64d))
}
```

4 байта uint32 для времени
8 байт uint64 для значений

Инициализируем весь массив сразу

delta encoding времени

delta encoding значений

BinaryStore. Read


```
res = make([]data.Element, elNum)
for i < len(bd) {
        tb = bd[i : i+4]
        tsV = binary.LittleEndian.Uint32(tb)
        ts += int64(tsV)
        vb = bd[i : i+8]
        f64 += binary.LittleEndian.Uint64(vb)
        f64e = data.Element{
                Timestamp: ts,
                Value:     math.Float64frombits(f64),
        }
        res[ec] = f64e
}
```

Инициализируем весь массив сразу

delta decoding времени

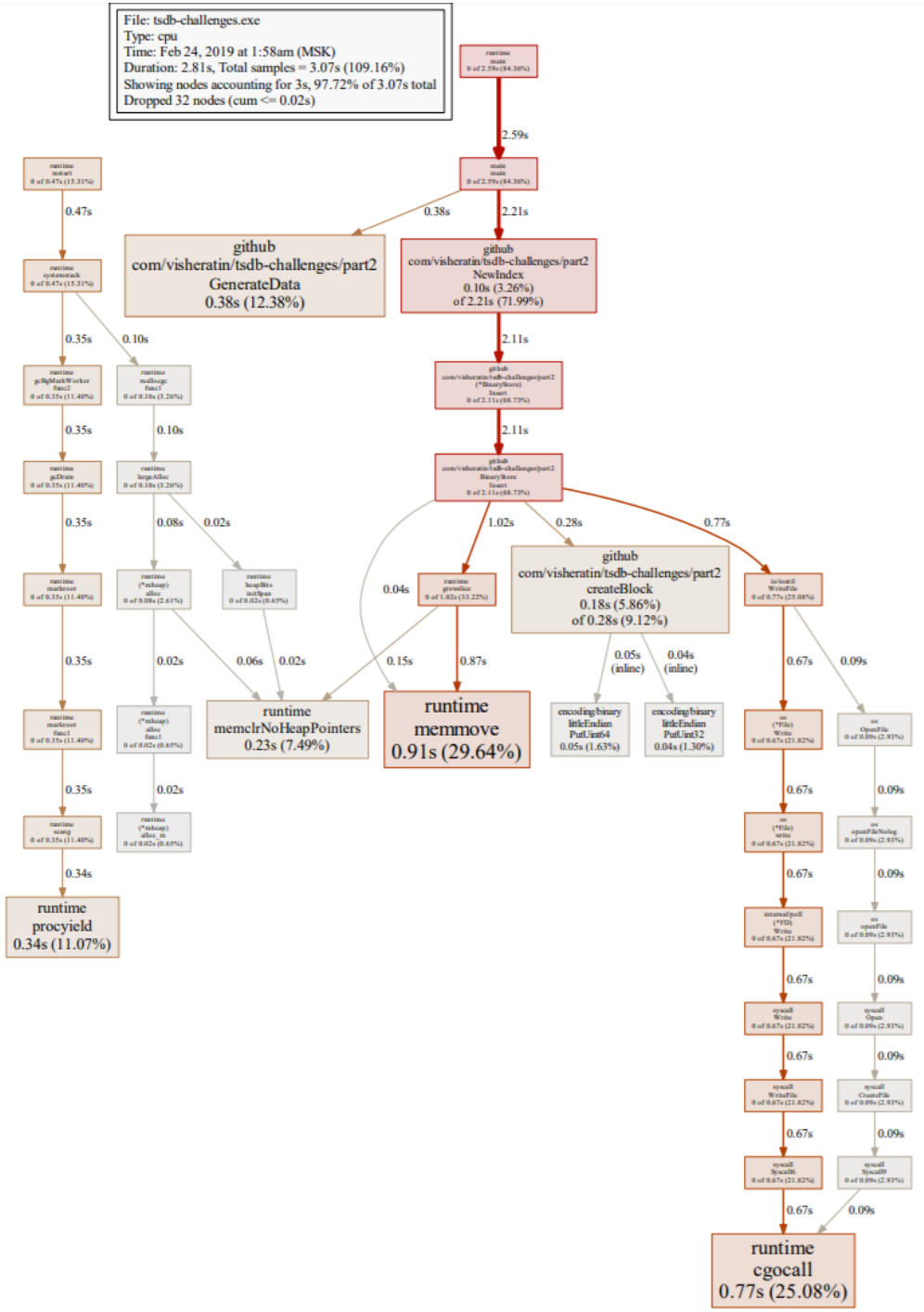
delta decoding значений

Профиль работы BinaryStore

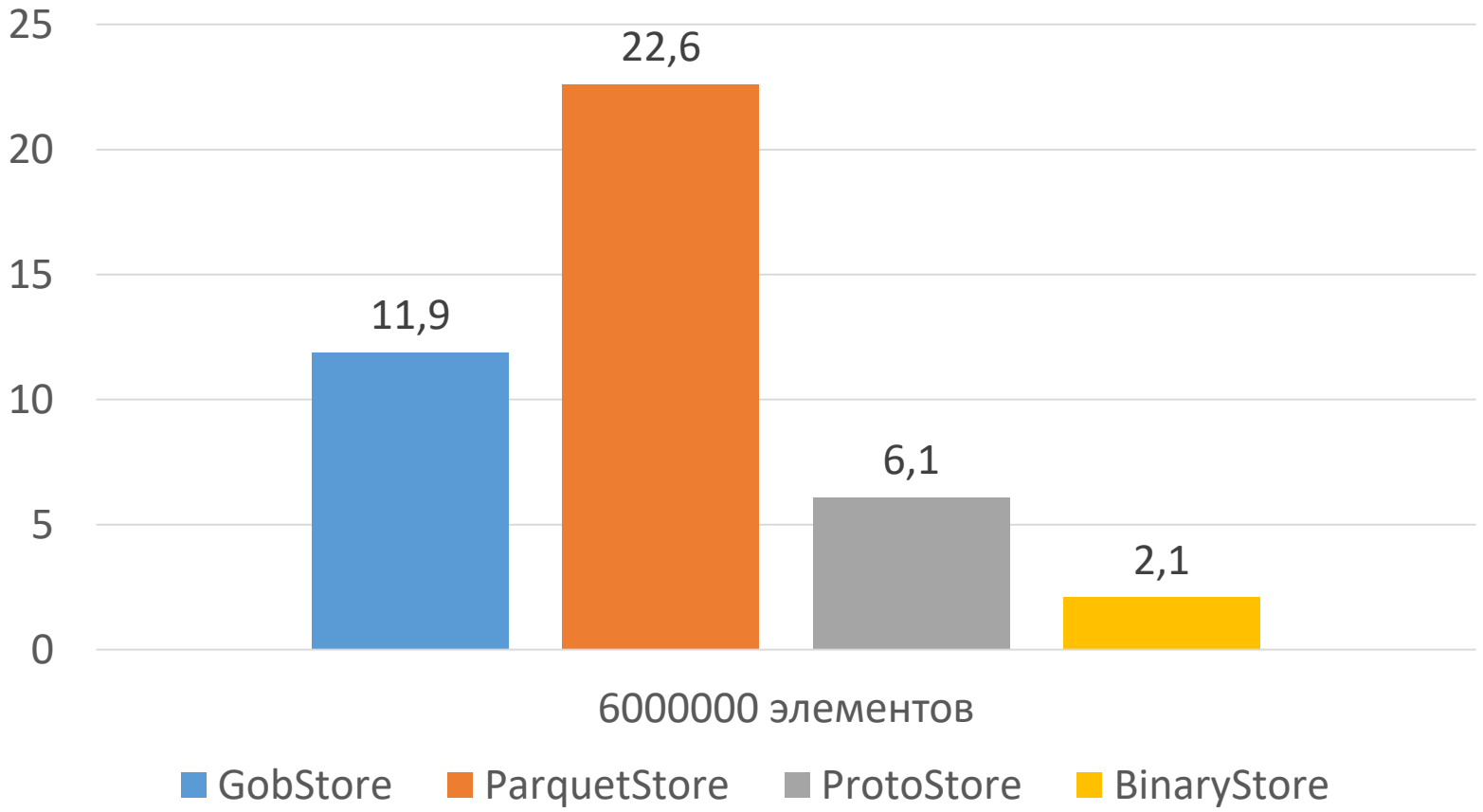
Профиль намного меньше, благодаря фактически ручному управлению процессом.

Всегда точно известны типы данных, нет нужды в преобразованиях.

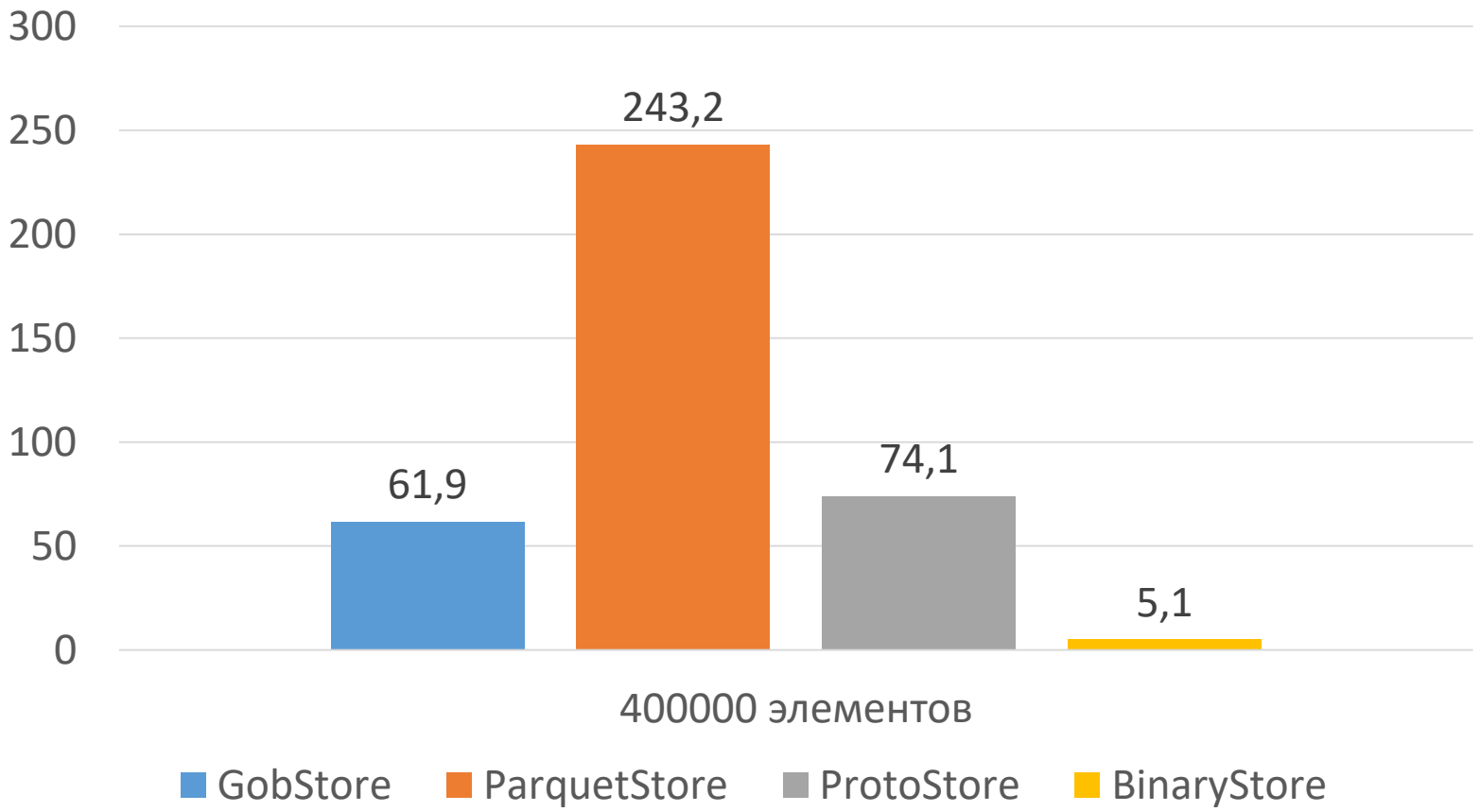
	Количество блоков
GobStore	74
ParquetStore	90
ProtoStore	46
BinaryStore	40



Бенчмарк хранилищ. Insert (время, сек)



Бенчмарк хранилищ. Extract (время, мс)



Вывод № 2

Иногда стандартных и проверенных решений может быть недостаточно.

Когда стоит действительно сложная задача, переход на более низкие уровни абстракции является единственным решением.

Поддержка множества типов данных

Необходимо реализовать поддержку всех базовых численных типов – int8, int16, int32, int64, float32, float64.

Классический способ – использовать интерфейсы.

```
type Element interface {
        Timestamp() int64
        Value() float64
}
```

для простоты только три типа

```
type Int32Element struct {
        Ts  int64
        Val int32
}
```

```
type Float32Element struct {
        Ts  int64
        Val float32
}
```

```
type Float64Element struct {
        Ts  int64
        Val float64
}
```

Реализация на хранилища интерфейсах

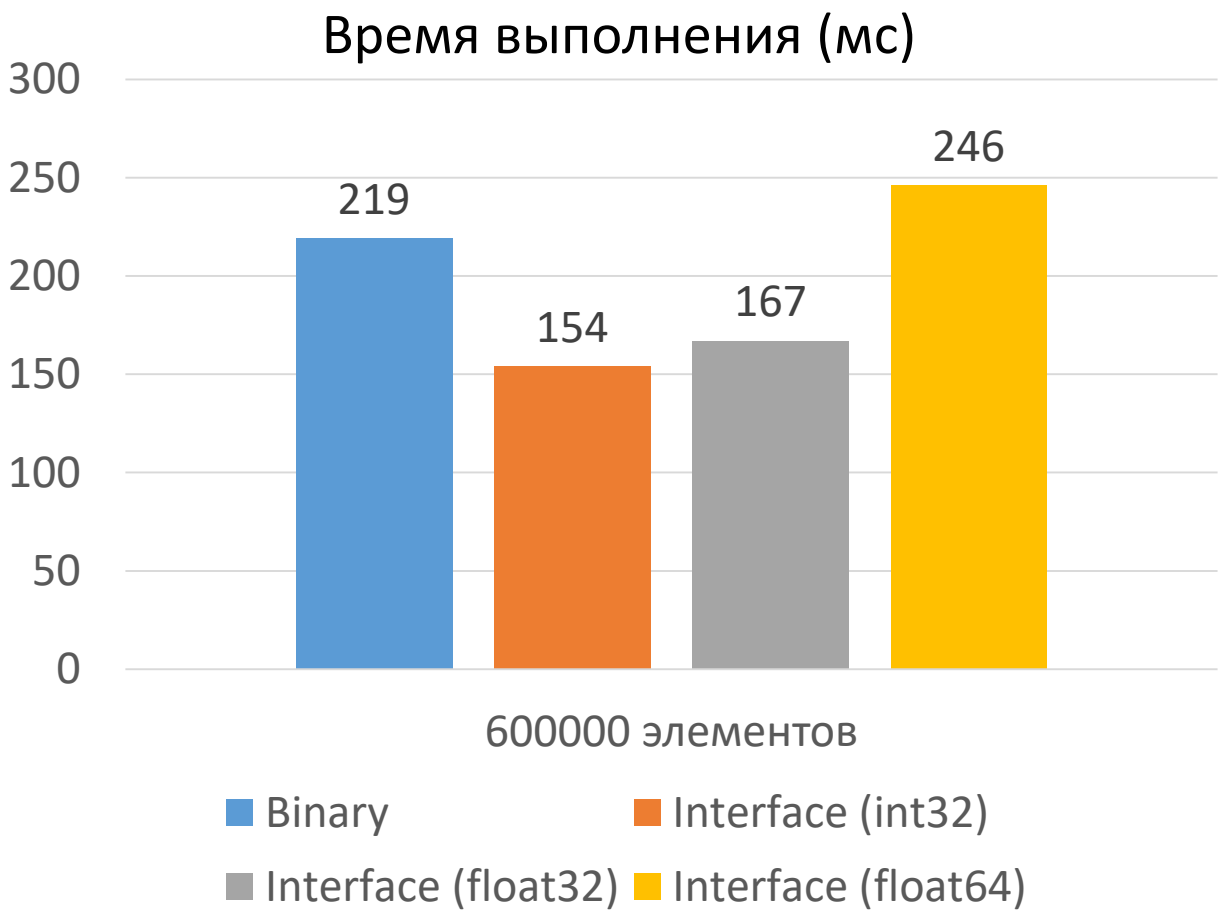
(–) много практически одинакового кода;

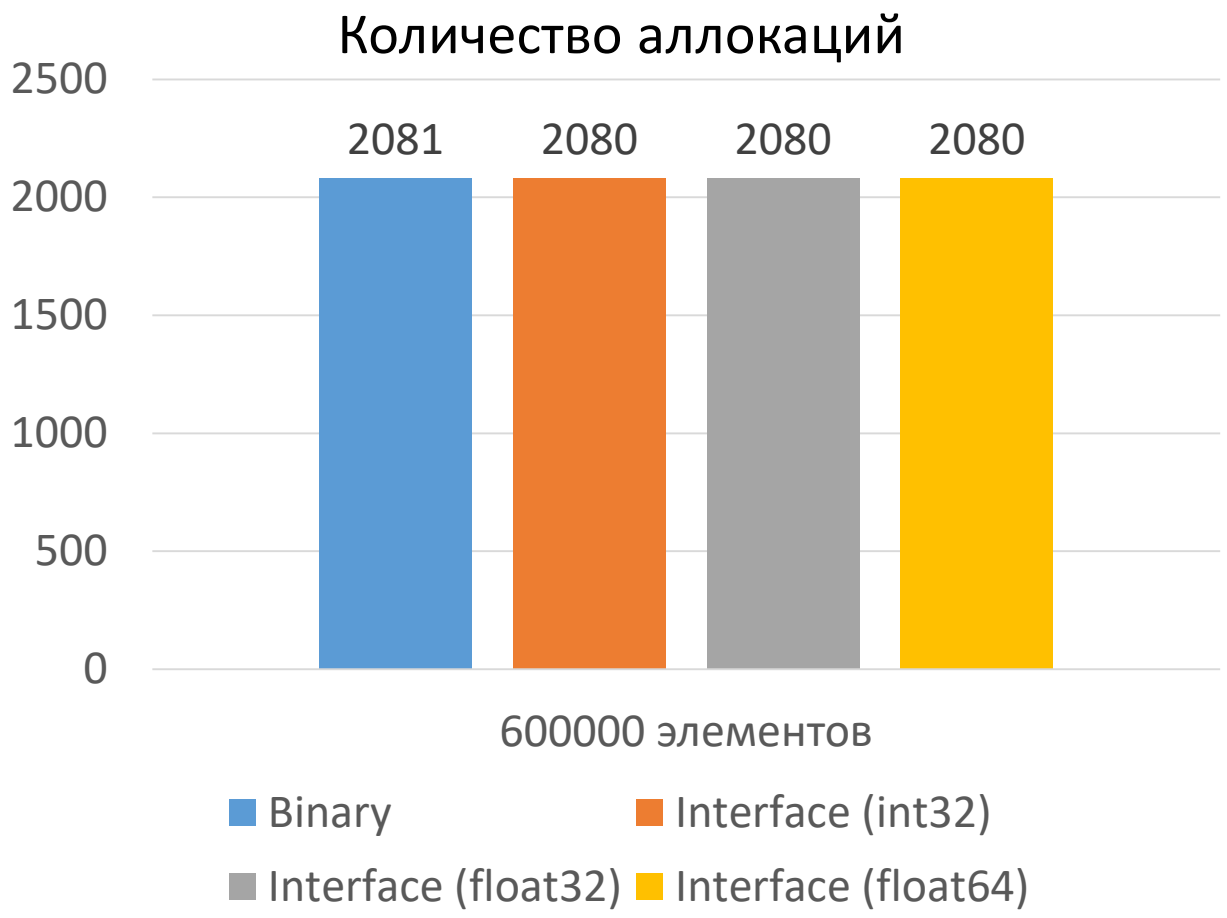
(–) необходимо писать switch’и для обработки типов;

(+) в любой другой части кодовой базы можно пользоваться интерфейсными методами.

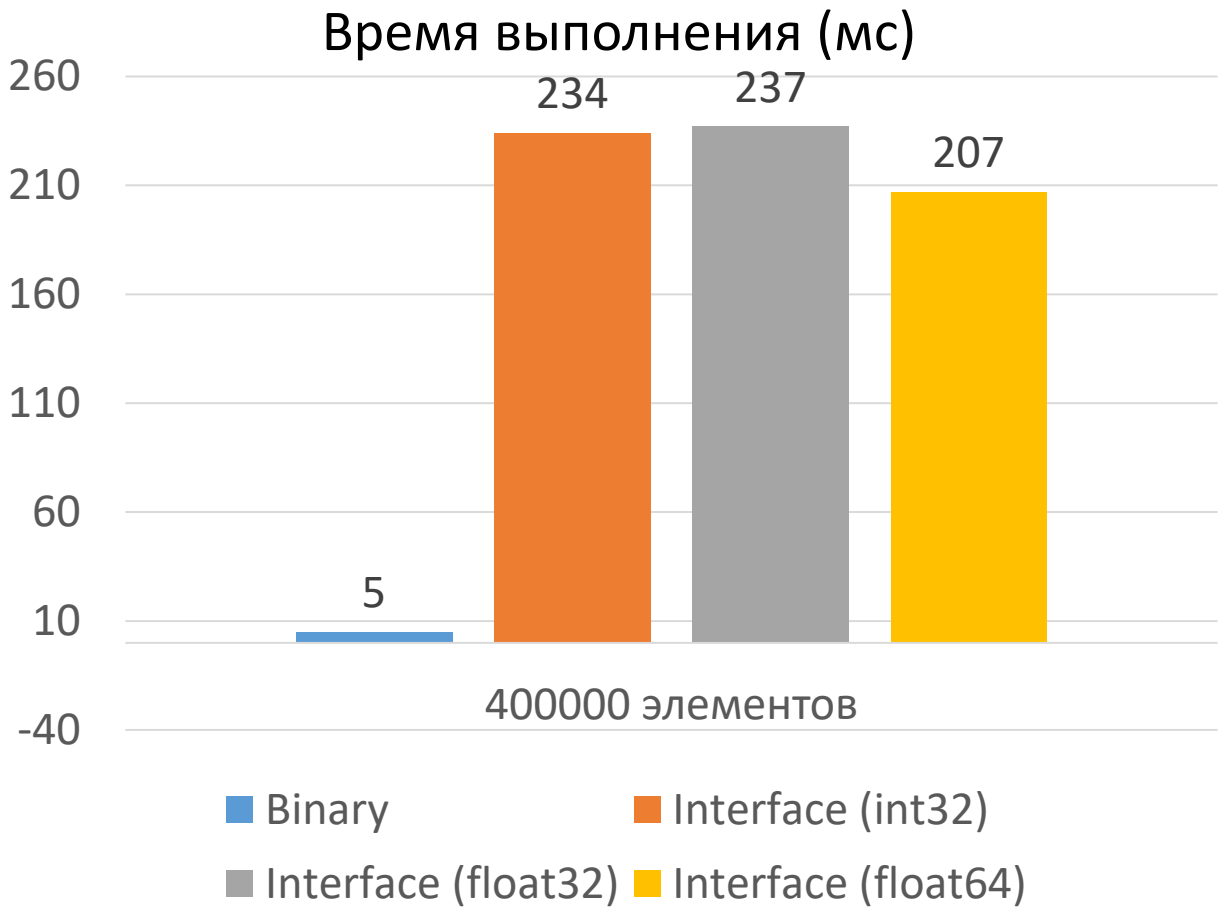
```
switch dtype {
case part3.Int32:
        f32 = uint32(d[i].(part3.Int32Element).Val)
        f32d = f32 - f32p
        f32p = f32
        binary.LittleEndian.PutUint32(buf[c:c+4], f32d)
        c += 4
case part3.Float32:
        …
case part3.Float64:
        …
}
```

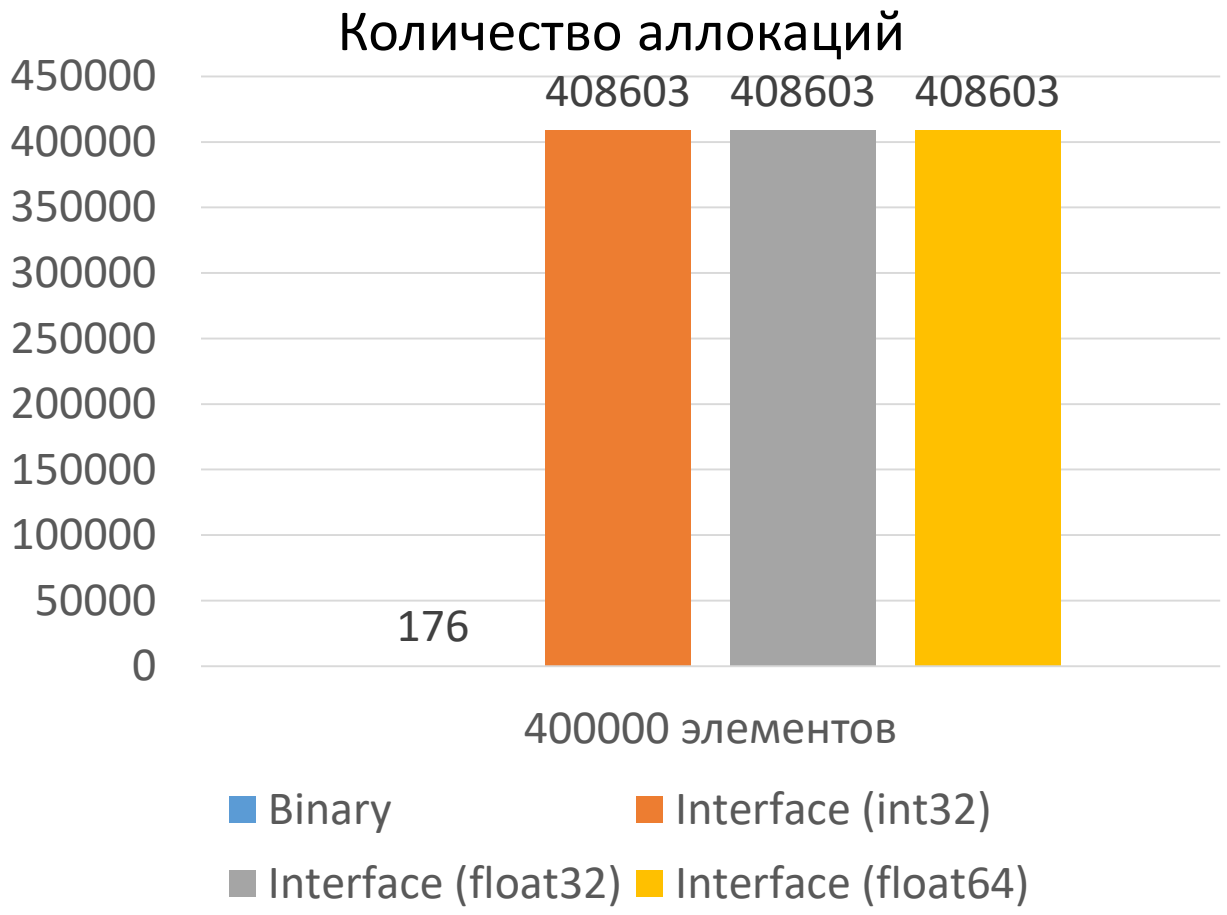
Бенчмарк интерфейсной реализации Insert



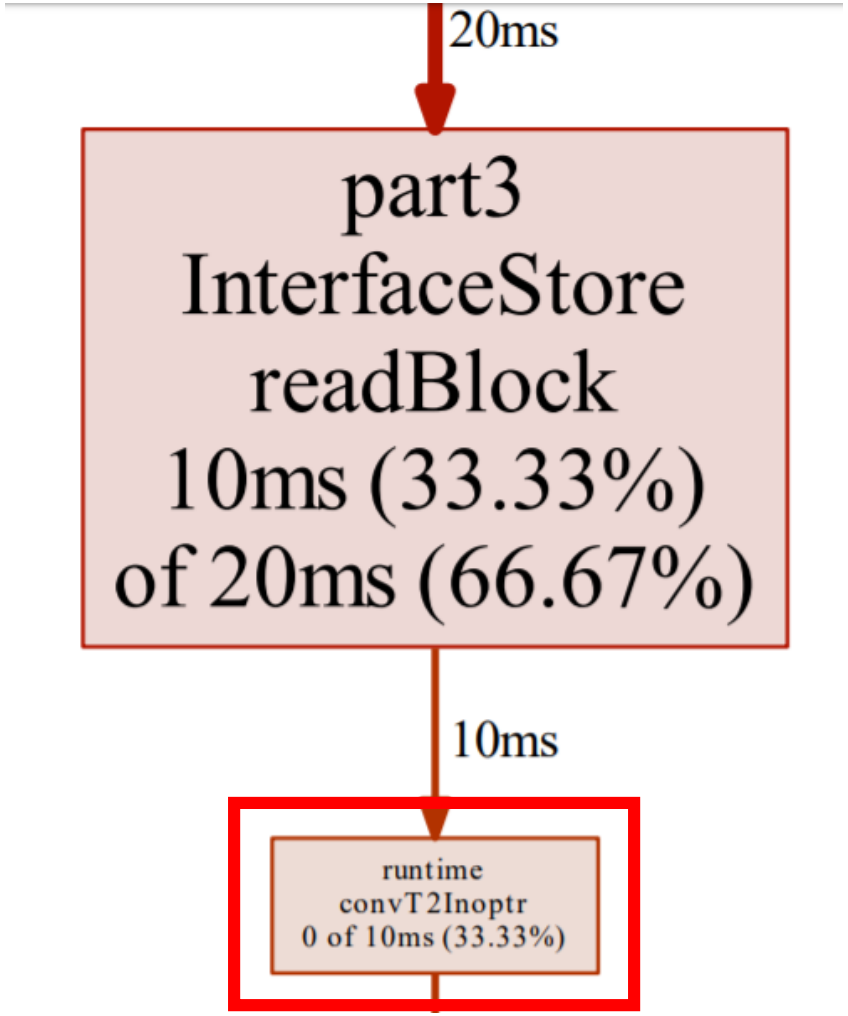


Бенчмарк интерфейсной реализации Extract





Профилирование извлечения данных



Что такое convT2Inoptr?

Согласно официальной документации: «Type to non-empty-interface conversion».

Из исходного кода* понятно, что при каждой конвертации происходит выделение памяти.

Что делать? Придумать, как не выделять память.

* https://github.com/golang/go/blob/master/src/runtime/iface.go

Хранилище на комбинированной структуре


```
type Elements struct {
        Type part3.DataType
        I32   []part3.Int32Element
        F32   []part3.Float32Element
        F64   []part3.Float64Element
}
```

(–) посчитать длину – нужен switch;

(–) извлечь значение – нужен switch;

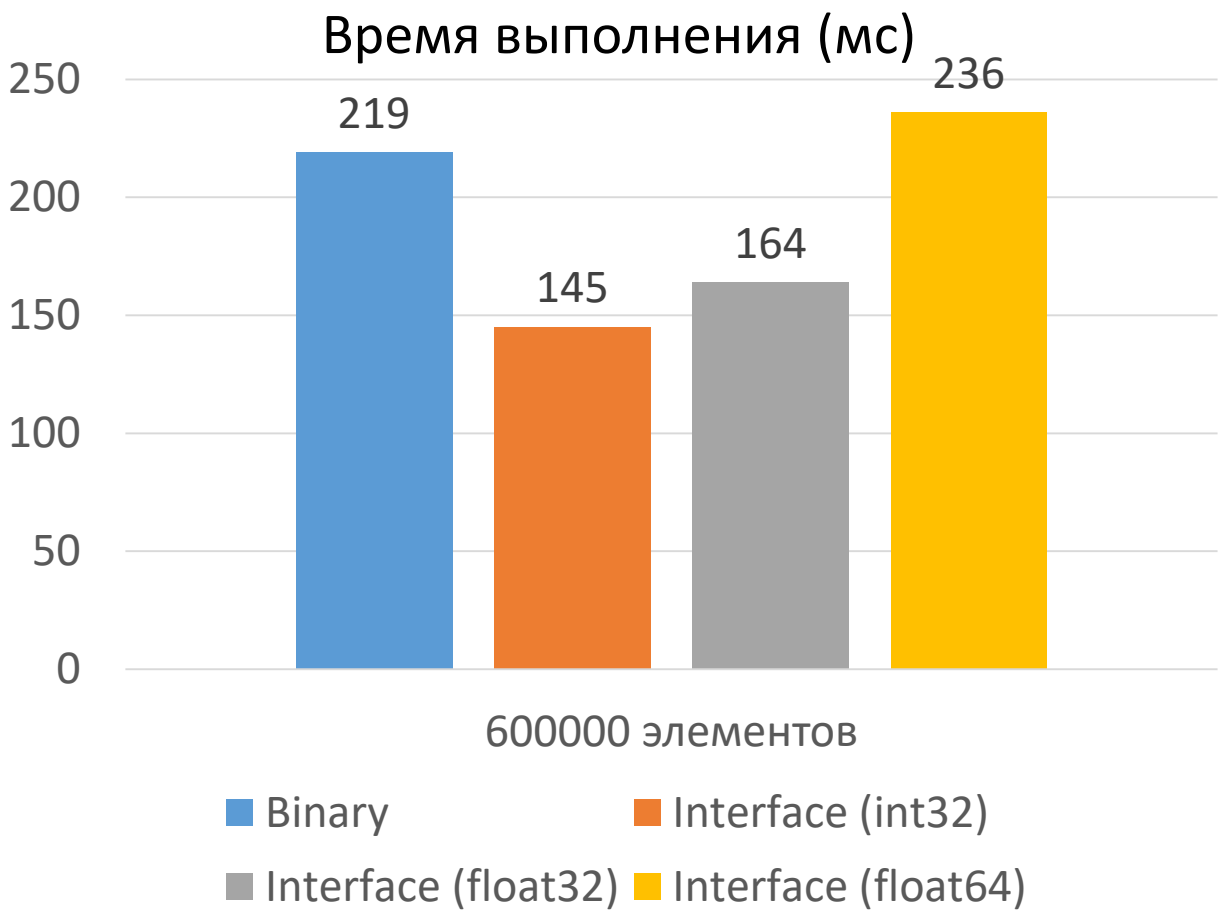
(–) добавить элементы – нужен switch;

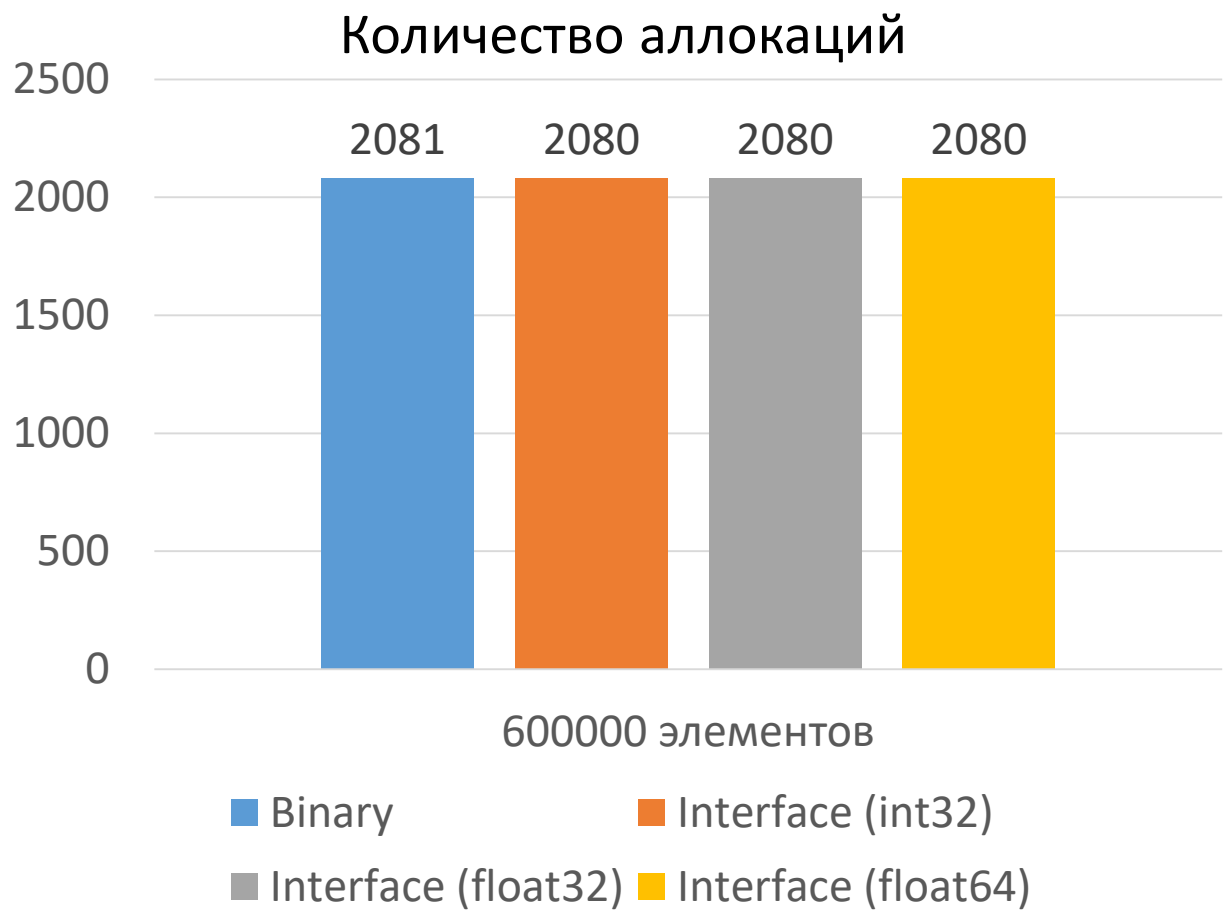
…

(–) во всей кодовой базе придется использовать switch’и;

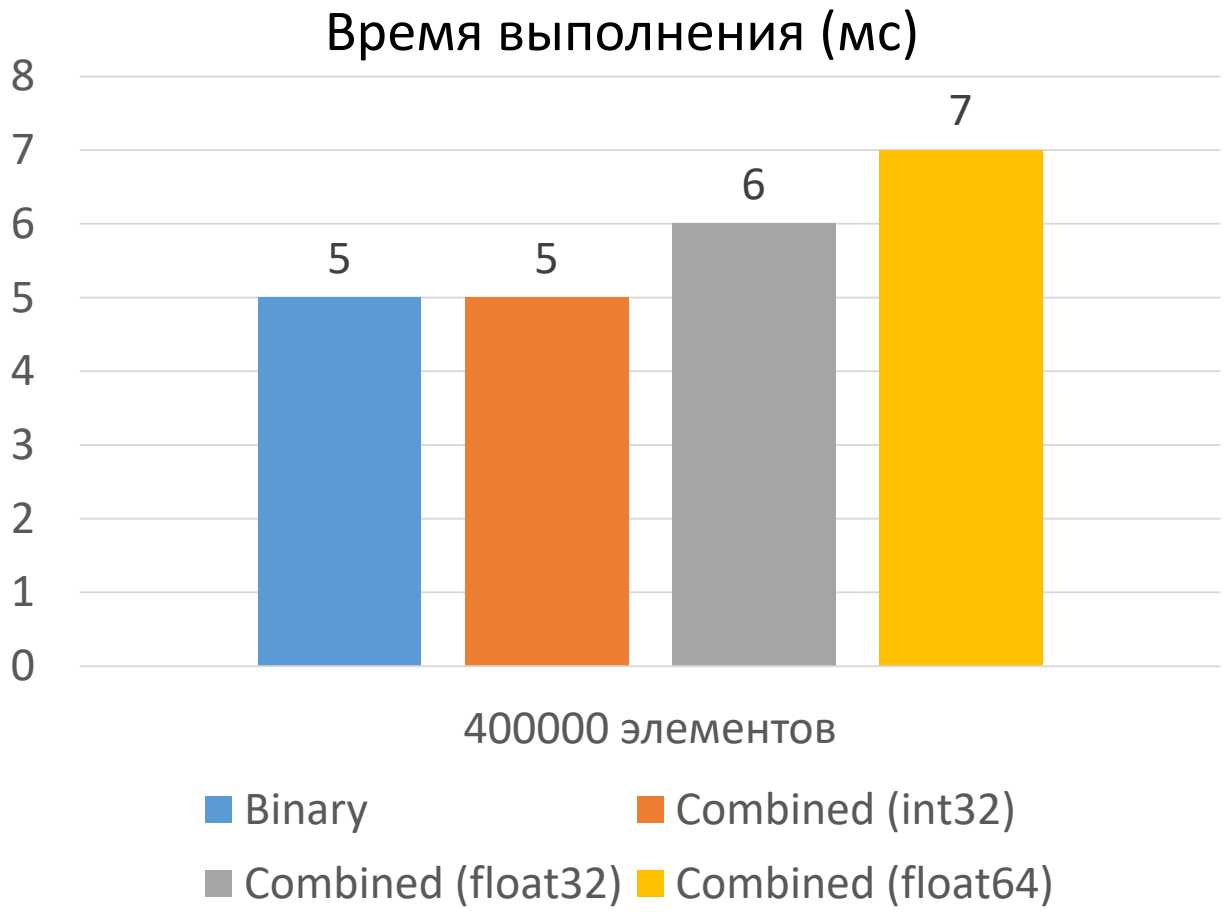
(+) должен быть быстрым.

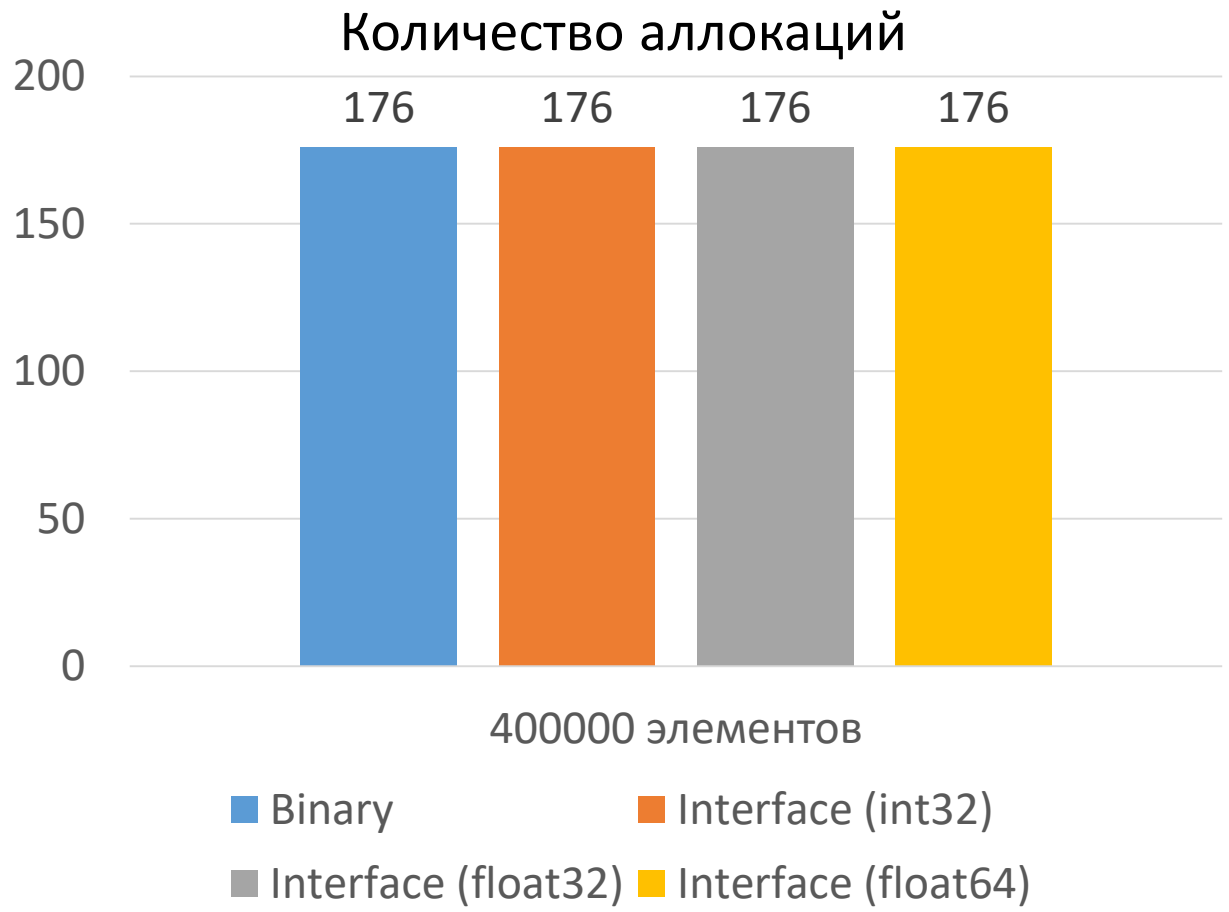
Бенчмарк комбинированной структуры Insert





Бенчмарк комбинированной структуры Extract





Вывод № 3

Использование интерфейсов в Go не бесплатное, но не в том направлении, которое вы бы предположили (из интерфейса в тип, а не наоборот).

Если нужно сделать быстро и с минимальными накладными расходами, иногда придется пожертвовать красотой и простотой кода.

Что мы получили в итоге?

1. Поиск по сложным условиям в миллиардах записей в пределах 100-200 миллисекунд (до 10 раз быстрее конкурентов).
2. Извлечение 15 миллионов записей в секунду на одной машине.
3. Загрузка 2 миллионов записей в секунду на одной машине.
4. Сжатие данных в 6.5 раз (Delta + Zstandard).
5. Семплирование и агрегация.
6. Легковесный индекс (1 миллиард записей – 1 мегабайт).
7. Расширяемость новыми форматами входных/выходных данных и хранилищ (файловая система, Amazon S3, HDFS).

Итоги

1. Необходимо, насколько это возможно, выделять память заранее.
2. Начинать стоит с простых вариантов. Гиперпроектирование – плохо.
3. Самый быстрый (но не в плане времени разработки) способ – делать все руками.
4. В погоне за скоростью иногда приходится жертвовать удобством кода.

Спасибо за внимание!

Презентация и исходный код:

https://github.com/visheratin/tsdb-challenges

Вопросы/уточнения/замечания: @visheratin

Благодарности:

- Юрий Кузнецов
- Александр Логинов
- Денис Насонов
- Александр Бухановский

SIEMENS

Ingenuity for life

ITMO UNIVERSITY