Building a High-Performance Binary Serialization Format with In-Place Modification

HAMISH MORRISON

Cppcon
The C++ Conference

2025

September 13 - 19

Building a High-Performance Binary Serialization Format with In-Place Modification

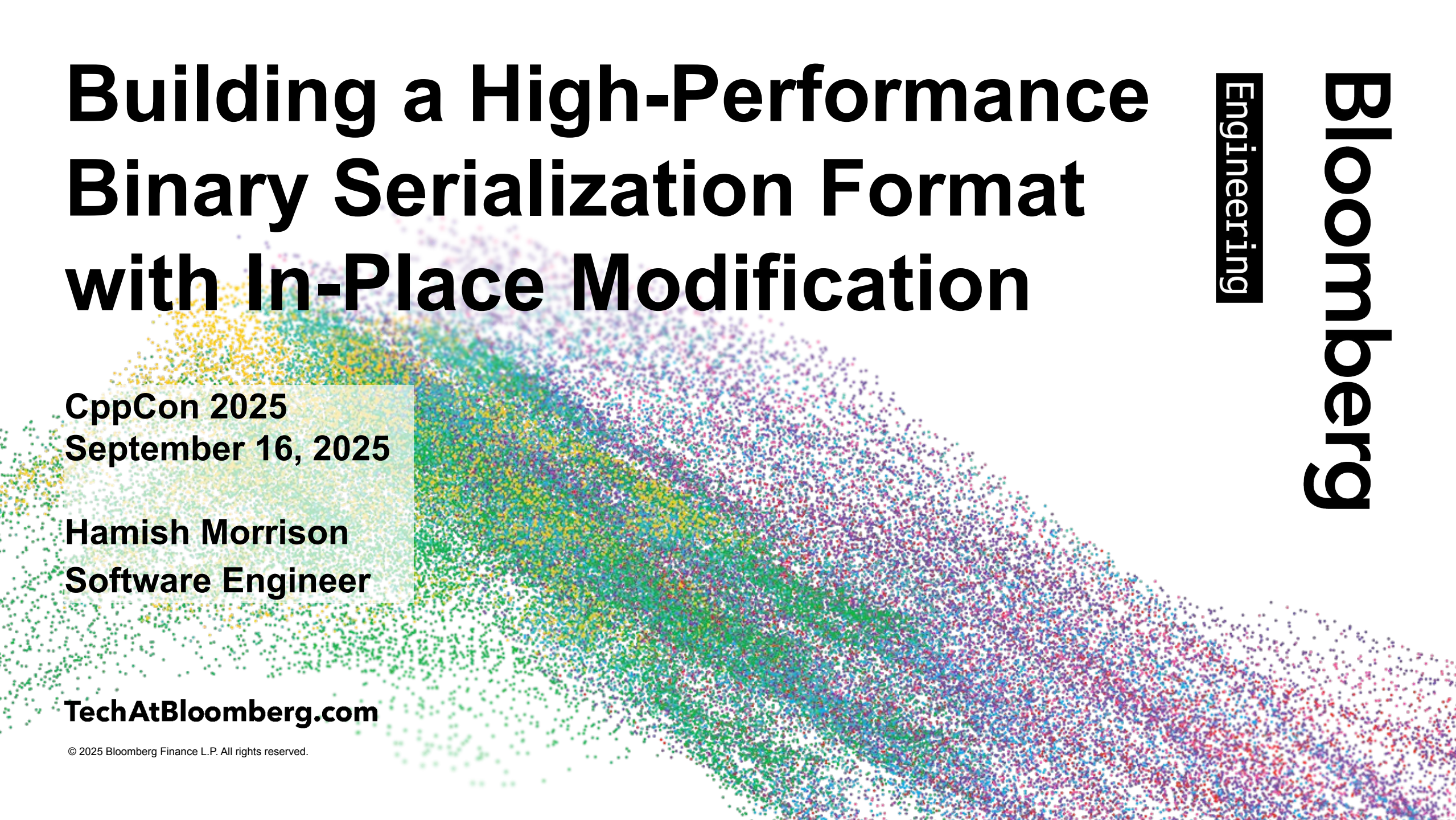
Bloomberg

Engineering

CppCon 2025
September 16, 2025

Hamish Morrison
Software Engineer

TechAtBloomberg.com



The Basics – What is Serialization?

- I have some data structure and I want to…
  - Send it over the network
  - Persist it somewhere
  - Share it between processes

- To do this, I need to turn it into a sequence of bytes

- And I need to be able to turn those bytes back into original data structure

The Basics – What is Serialization?

- Example data structure:

```
struct Trade {
    unsigned      price;
    unsigned      volume;
    unsigned long timestamp;
    char          tradeId[20];
    char          buyBroker[5];
    char          sellBroker[5];
    unsigned      flags;
};
```


Bloomberg

Engineering

The Basics – What is Serialization?

- How do I serialize this?

```
struct Trade {
    unsigned      price;
    unsigned      volume;
    unsigned long timestamp;
    char          tradeId[20];
    char          buyBroker[5];
    char          sellBroker[5];
    unsigned      flags;
};
```


Bloomberg

Engineering

Serializing our Trade struct

- How about this:

```
size_t serialize(char *buffer, const Trade& trade)
{
    memcpy(buffer, &trade, sizeof(trade));
    return sizeof(Trade);
}
```

Serializing our Trade struct

- How about this:

```
size_t serialize(char *buffer, const Trade& trade)
{
    memcpy(buffer, &trade, sizeof(trade));
    return sizeof(Trade);
}
```

- This is no good

Serializing our Trade struct

- How about this:

```
size_t serialize(char *buffer, const Trade& trade)
{
    memcpy(buffer, &trade, sizeof(trade));
    return sizeof(Trade);
}
```

- This is no good
- Why?

Bloomberg

Engineering

Serializing our Trade struct

- The problems with using memcpy:

```
struct Trade {
    unsigned      price;
    unsigned      volume;
    unsigned long timestamp;
    char          tradeId[20];
    char          buyBroker[5];
    char          sellBroker[5];
    unsigned      flags;
};
```

Endianness

Size of types

Padding between types

TechAtBloomberg.com

Bloomberg

Engineering

Serializing our Trade struct

- A better version:

```
void serialize(char *buffer, const Trade& trade)
{
  uint32_t price = htonl(trade.price);
  memcpy(buffer, &price, sizeof(price));
  buffer += sizeof(price);
  uint32_t volume = htonl(trade.size);
  memcpy(buffer, &size, sizeof(size));
  buffer += sizeof(volume);
  ...
}
```

Types with defined sizes

Swap everything to network order

Consistent padding

Serializing our `Trade` struct

- It’s better, but it’s still not great
  - We have to write a lot of boilerplate
  - The sending and receiving code have to agree exactly on this format
  - Evolving the schema is not possible
  - It’s not easy to see how to encode more complex nested structures

How Do We Improve on This?

- Two broad approaches:
  - Schema-based
  - Schemaless

In a schema-based format, the receiver has to know something about the shape of the data before it can decode it meaningfully.

In schemaless formats, it doesn’t need to know anything.

Schema-Based Formats

- In schema-based formats, the user defines a schema, usually in a DSL
- Often, code is generated from the schema to encode and decode messages
- e.g., in protobufs:

```
message Trade {
    optional uint32 price     = 1;
    optional uint32 volume    = 2;
    optional uint64 timestamp = 3;
    optional string tradeId   = 4;
    ...

};
```

Bloomberg

Engineering

Schema-Based Formats

- Sender and receiver need to agree on the schema, to an extent

```
message Trade {
    optional uint32 price     = 1;
    optional uint32 volume    = 2;
    optional uint64 timestamp = 3;
    optional string tradeId   = 4;
    ...

};
```

Bloomberg

Engineering

Schema-Based Formats

- Sender and receiver need to agree on the schema, to an extent
- But the schema is allowed to evolve in some ways

```
message Trade {
    optional uint32 price     = 1;
    optional uint32 volume    = 2;
    optional uint64 timestamp = 3;
    optional string tradeId   = 4;
    ...

};
```

Bloomberg

Engineering

Schema-Based Formats

- Sender and receiver need to agree on the schema, to an extent
- But the schema is allowed to evolve in some ways
- e.g., a new field can be added

```
message Trade {
    optional uint32 price     = 1;
    optional uint32 volume    = 2;
    optional uint64 timestamp = 3;
    optional string tradeId   = 4;
    optional string brokerId  = 5;
    ...
};
```

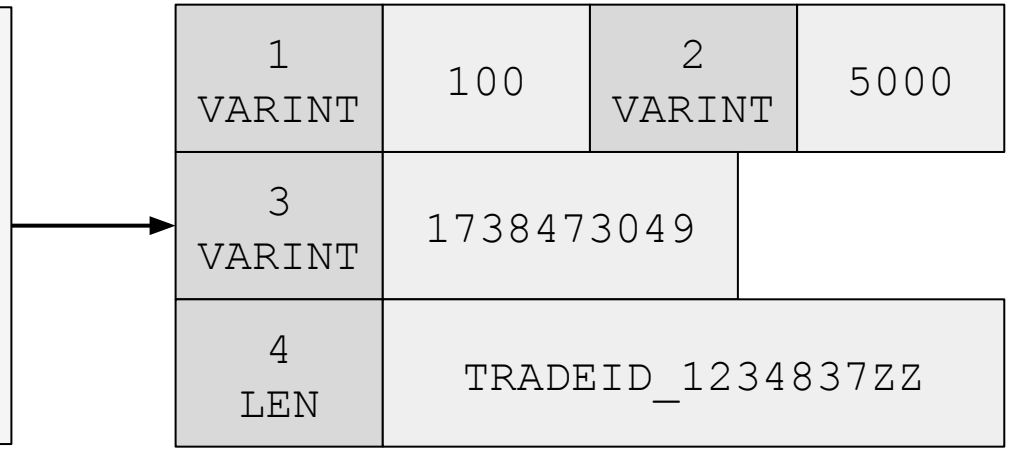
Bloomberg

Engineering

Schema-Based Formats

- On the wire, the protobuf message is encoded as a sequence of pairs
- Pairs of:
  - Tag – which is a field number and wire type
  - And value

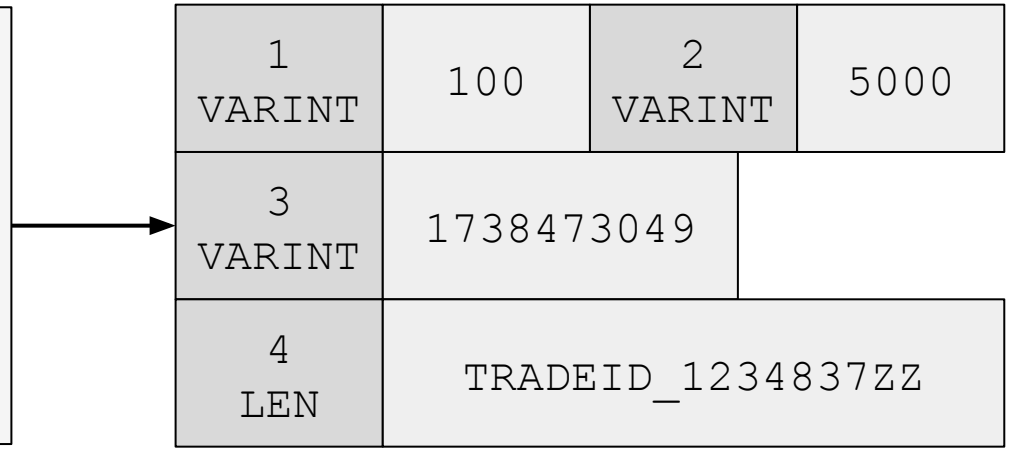
```
message Trade {
    optional uint32 price     = 1;
    optional uint32 volume    = 2;
    optional uint64 timestamp = 3;
    optional string tradeId   = 4;
    ...
};
```



Schema-Based Formats

- This encoding is somewhat self-describing
- I can traverse through it, even if I don’t know the schema
- But I need the schema to make sense of it (e.g. to know field names, etc)

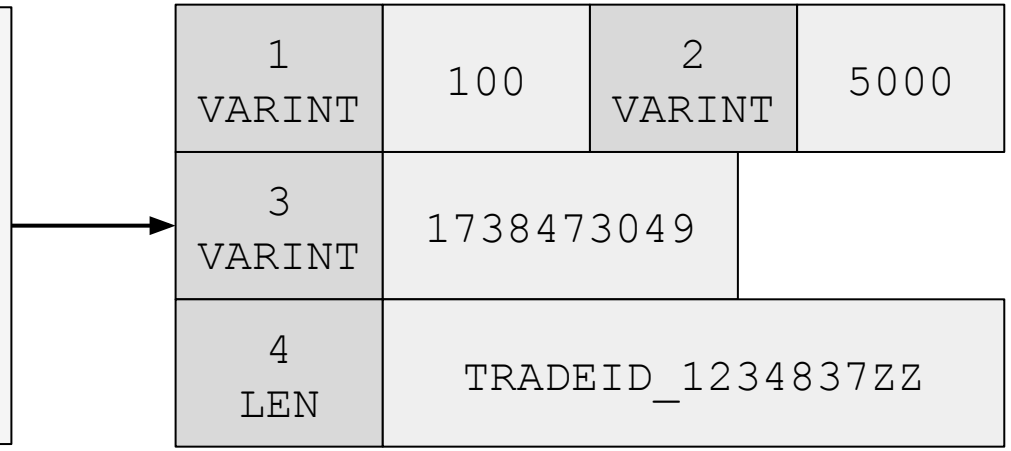
```
message Trade {
    optional uint32 price     = 1;
    optional uint32 volume    = 2;
    optional uint64 timestamp = 3;
    optional string tradeId   = 4;
    ...
};
```



Schema-Based Formats

- What if I don’t know the shape of my data up front?
- What if I want to encode more complex, arbitrarily nested documents of maps and arrays?

```
message Trade {
    optional uint32 price     = 1;
    optional uint32 volume    = 2;
    optional uint64 timestamp = 3;
    optional string tradeId   = 4;
    ...
};
```



Schemaless Formats

- These formats generally let you encode anything, without having to agree on a schema between sender and receiver

- The user often gets a dictionary-style API to interact with, something like:

```
class Value {
  int    asInt() const;
  string asString() const;
  Value  get(std::string key);
  void   set(std::string key, int value);
  void   set(std::string key, string value);
};
```


Bloomberg

Engineering

Schemaless Formats

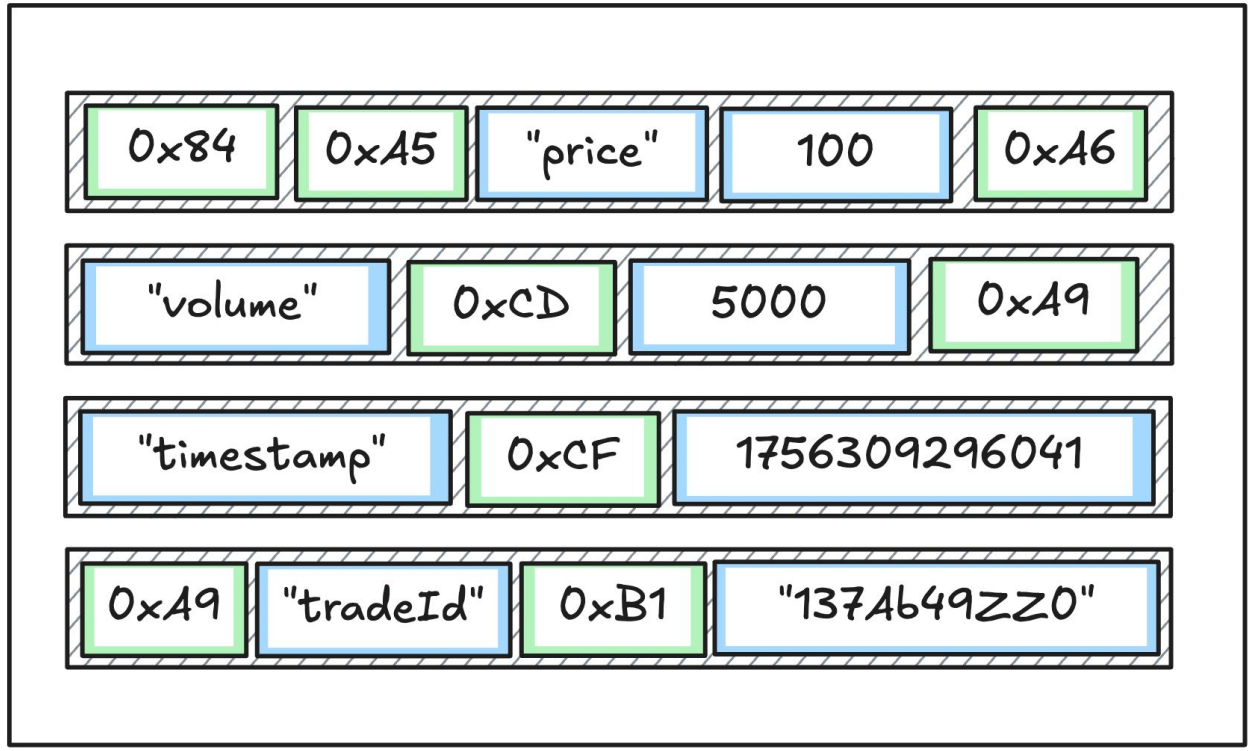
- Examples of schemaless formats include:
  - JSON
  - BSON
  - MsgPack
  - CBOR
  - FlexBuffers

- The binary formats are usually encoded as a sequence of tag, value pairs

Schemaless Formats

- An example document, encoded in both JSON and MsgPack:

```
{
  "price": 100,
  "volume": 5000,
  "timestamp": 1756309296041,
  "tradeId": "137A649ZZ0"
}
```



TechAtBloomberg.com

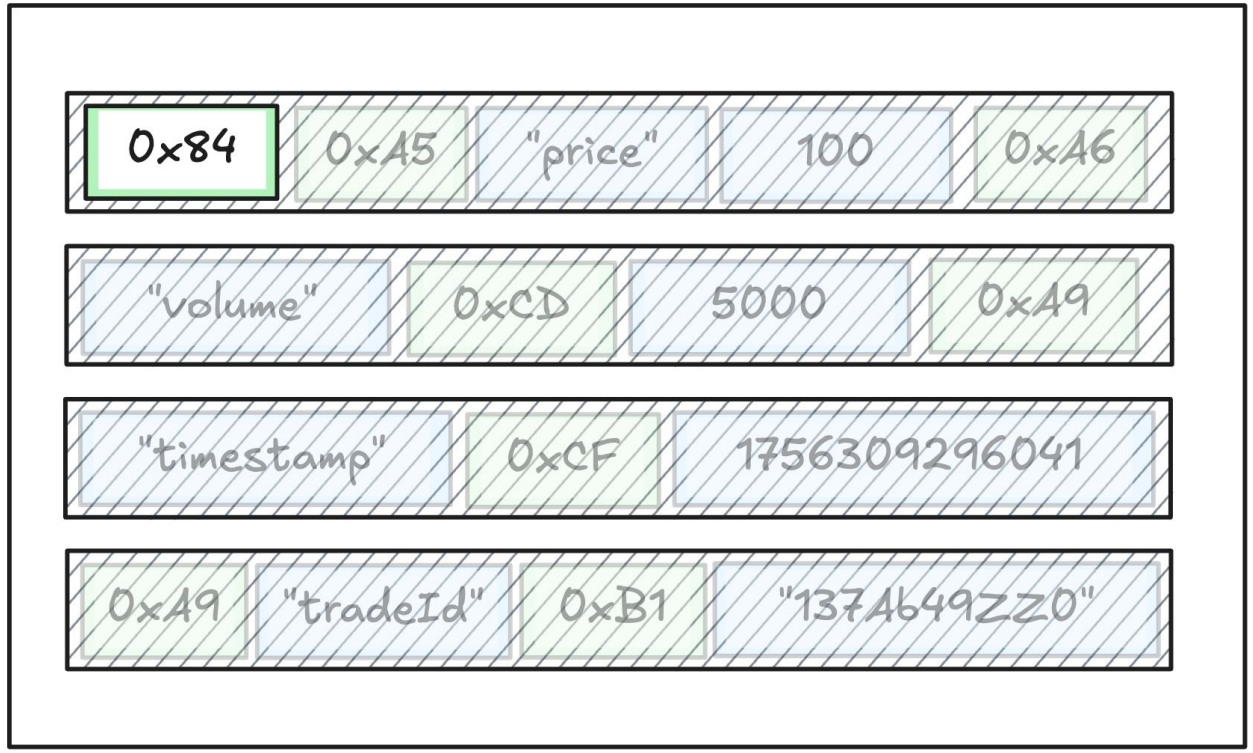
Bloomberg

Engineering

Schemaless Formats

- An example document, encoded in both JSON and MsgPack:

```
{
  "price": 100,
  "volume": 5000,
  "timestamp": 1756309296041,
  "tradeId": "137A649ZZ0"
}
```



TechAtBloomberg.com

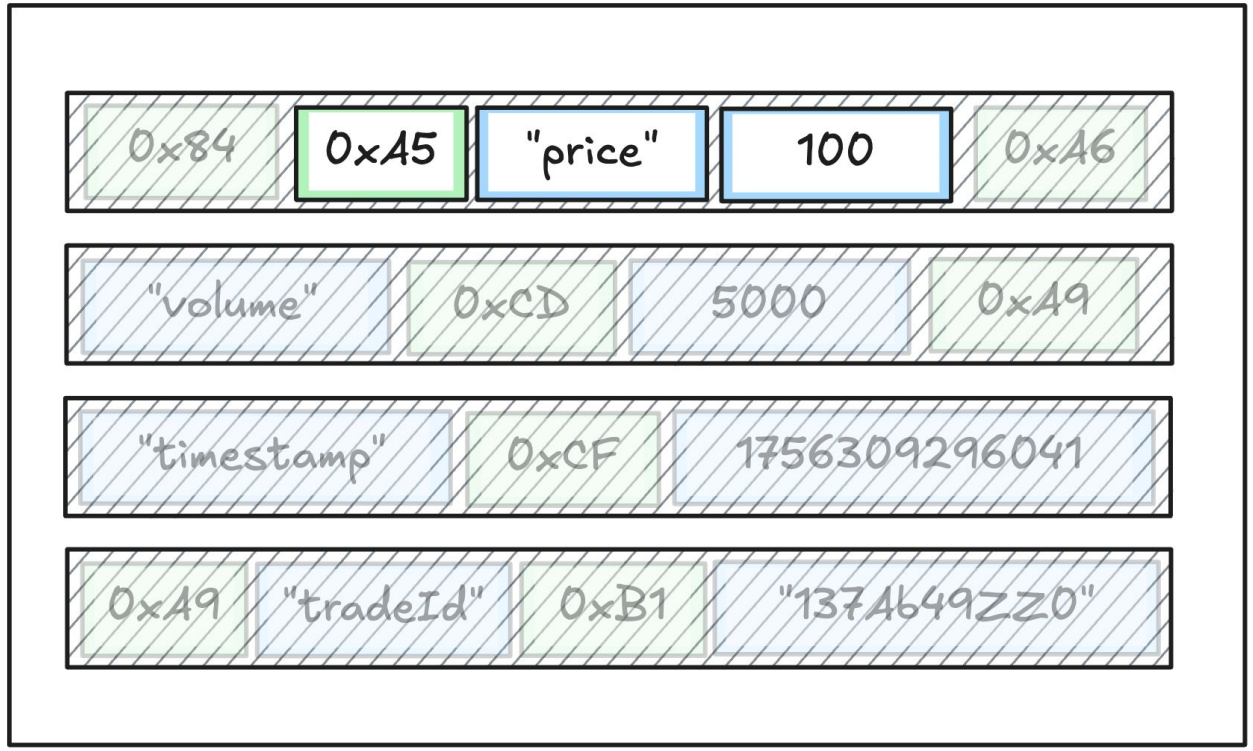
Bloomberg

Engineering

Schemaless Formats

- An example document, encoded in both JSON and MsgPack:

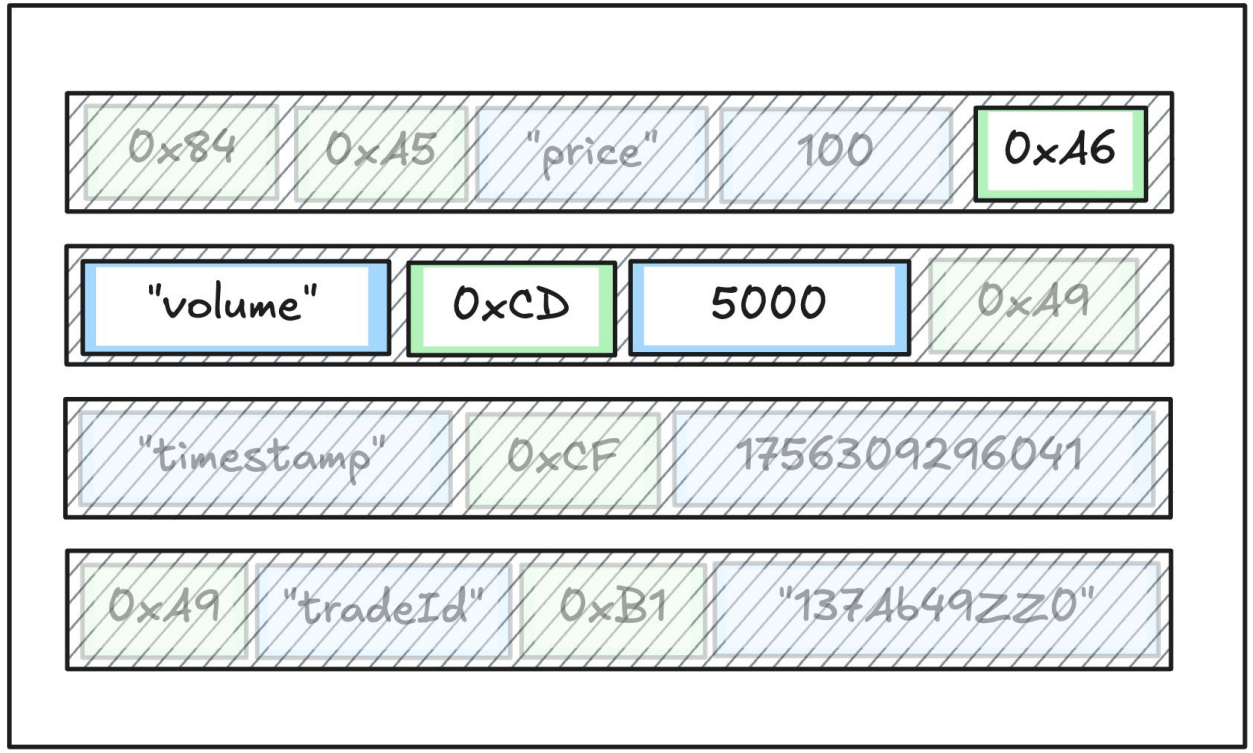
```
{
  "price": 100,
  "volume": 5000,
  "timestamp": 1756309296041,
  "tradeId": "137A649ZZ0"
}
```



Schemaless Formats

- An example document, encoded in both JSON and MsgPack:

```
{
  "price": 100,
  "volume": 5000,
  "timestamp": 1756309296041,
  "tradeId": "137A649ZZ0"
}
```



TechAtBloomberg.com

Bloomberg

Engineering

Schemaless Formats

- The tagged formats often get creative about packing small values
- In MsgPack, there are various values that do not need a tag byte:
  - `true` is encoded as `0xc2` and `false` as `0xc3`
  - Small integer values (less than 128) are simply encoded as themselves, without a tag
- As a result, binary schemaless formats are compact and elegant

Schemaless Formats

- What is the cost of using a schemaless format?
- In order to read an encoded message, it typically gets decoded into some in-memory data structure
- Processing is then done on the in-memory data structure
- It is then encoded back into the serialized form

Schemaless Formats

- What if I want to avoid the cost of always re-serializing the data?
- Maybe I want to pull a document out of a database and make some small, targeted modifications to it, and then write it back
- Or maybe I have some messages I’m receiving from the network, and I just want to pull one field out of each of them

Schemaless Formats

- Can we build a schemaless serialization format that supports fast traversal and modification in-place?

- **Let’s give it a try!**

Baseline Serialization Format

- First of all, let’s create a simple, unoptimized tagged serialization format
  - Similar to MsgPack, CBOR, etc.
- This will serve as a baseline for our benchmarks
- We’ll build on top of it to create our more advanced versions

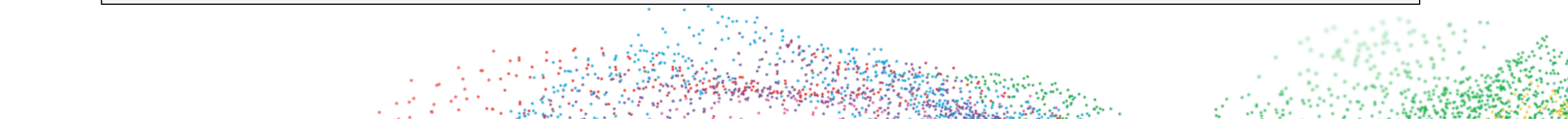
Baseline Serialization Format

- Baseline serialization format:
  - Supports strings, ints, floats, maps, and arrays
  - Encoded as a sequence of tag, value pairs
  - Tag will encode the type (and the size, if necessary)
  - Values will be in little-endian (to be friendly to modern CPUs)
  - Maps will be a sequence of key, value pairs
  - Arrays will be a sequence of values

Baseline Serialization Format

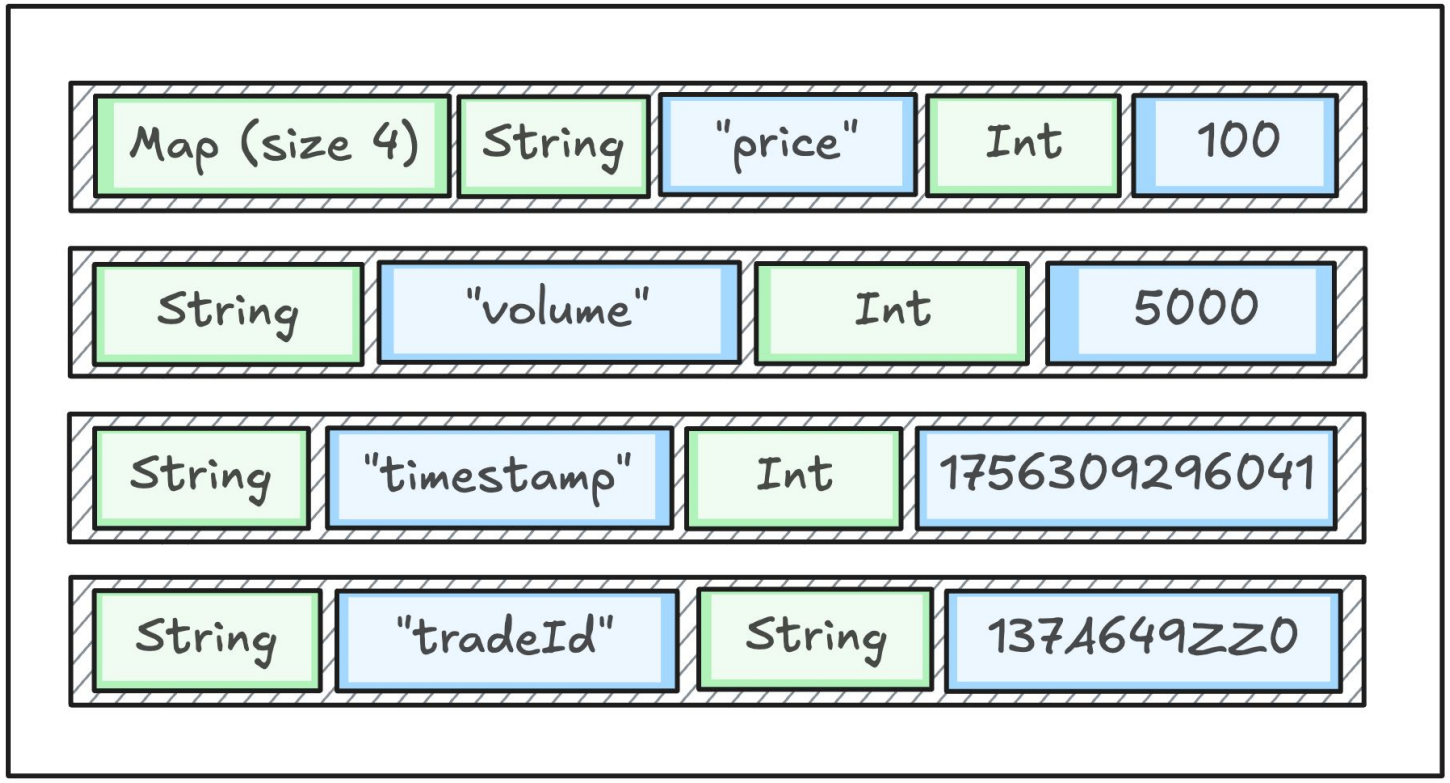
- Our interface will look something like this:

```
class Value {
    float asFloat() const;                          // get this value as a float
    int   asInt() const;                            // get this value as an int
    ...                                             // etc...

    void makeArray();                               // make this value an array
    void makeMap();                                 // make this value a map

    Value operator=(auto value);                    // assign the given value
    Value append(auto value);                       // append a value to this array

    Value set(std::string_view key, auto value);    // set key to value in map

    Value operator[](std::string_view key) const;   // get value for key in map
    Value operator[](int index) const;              // get value at index in array
};
```



Baseline Serialization Format

- To implement read operations, we will do a linear search through the document, looking for the right key



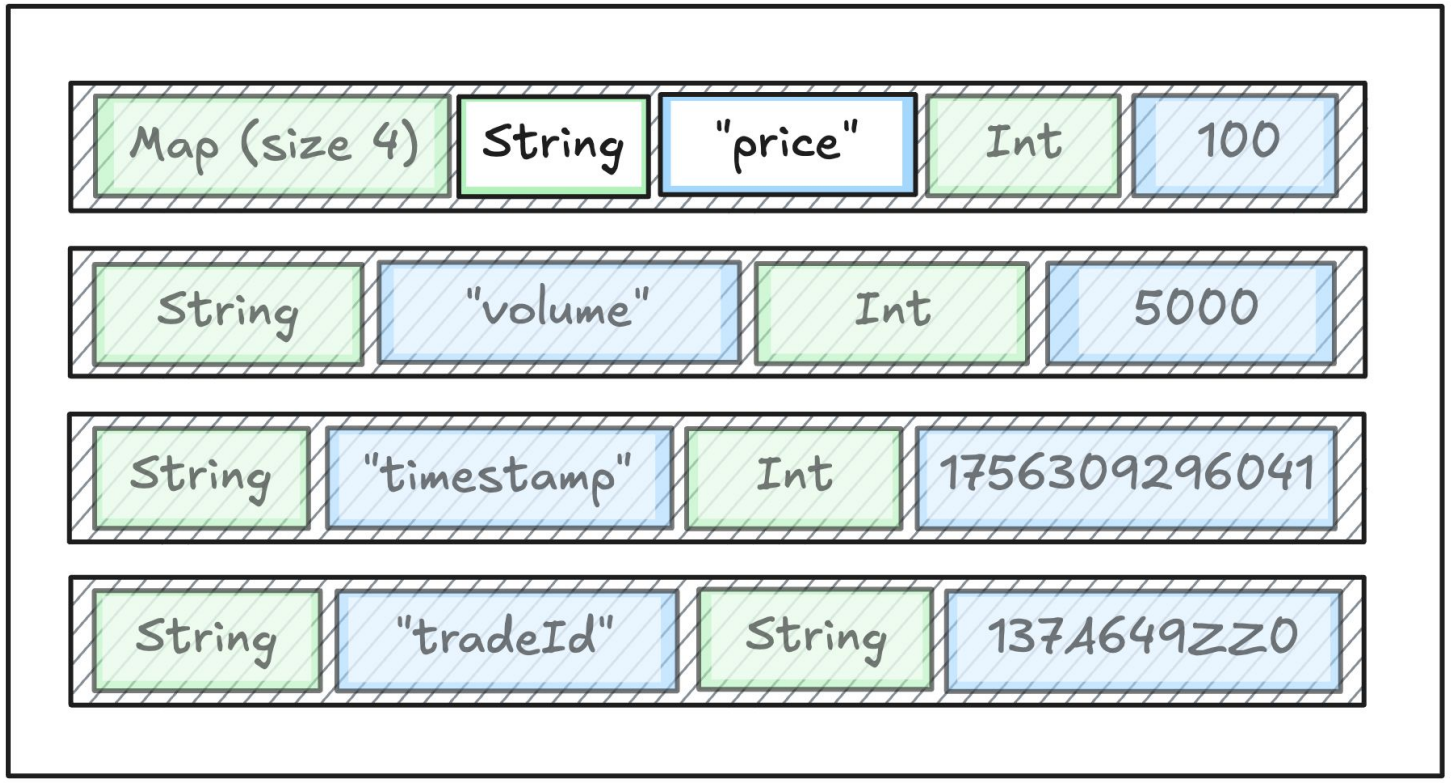

```
Value doc(buf);
Value ts = doc["timestamp"];
int64_t value = ts.asInt();
```

Bloomberg

Engineering

Baseline Serialization Format

- To implement read operations, we will do a linear search through the document, looking for the right key



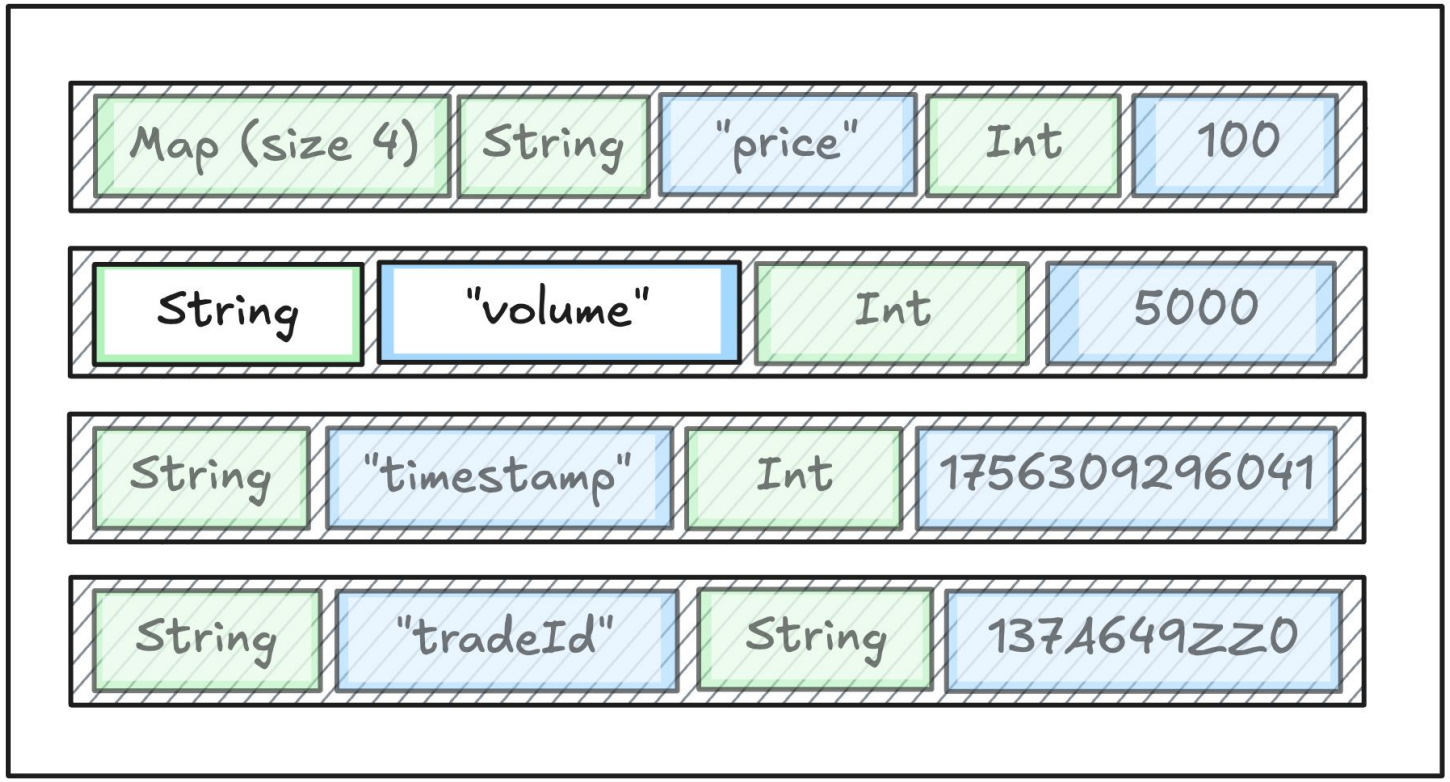

```
Value doc(buf);
Value ts = doc["timestamp"];
int64_t value = ts.asInt();
```

Bloomberg

Engineering

Baseline Serialization Format

- To implement read operations, we will do a linear search through the document, looking for the right key



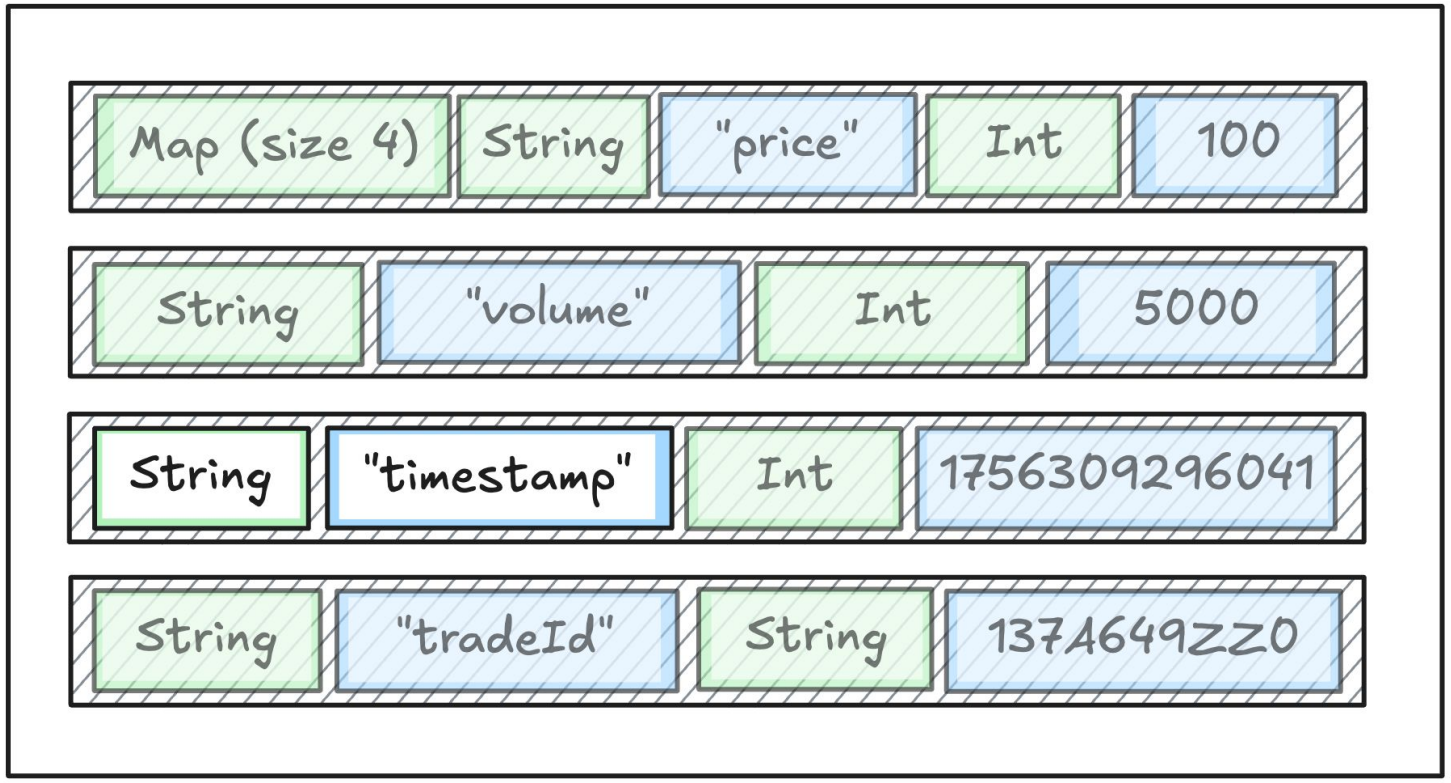

```
Value doc(buf);
Value ts = doc["timestamp"];
int64_t value = ts.asInt();
```

Bloomberg

Engineering

Baseline Serialization Format

- To implement read operations, we will do a linear search through the document, looking for the right key



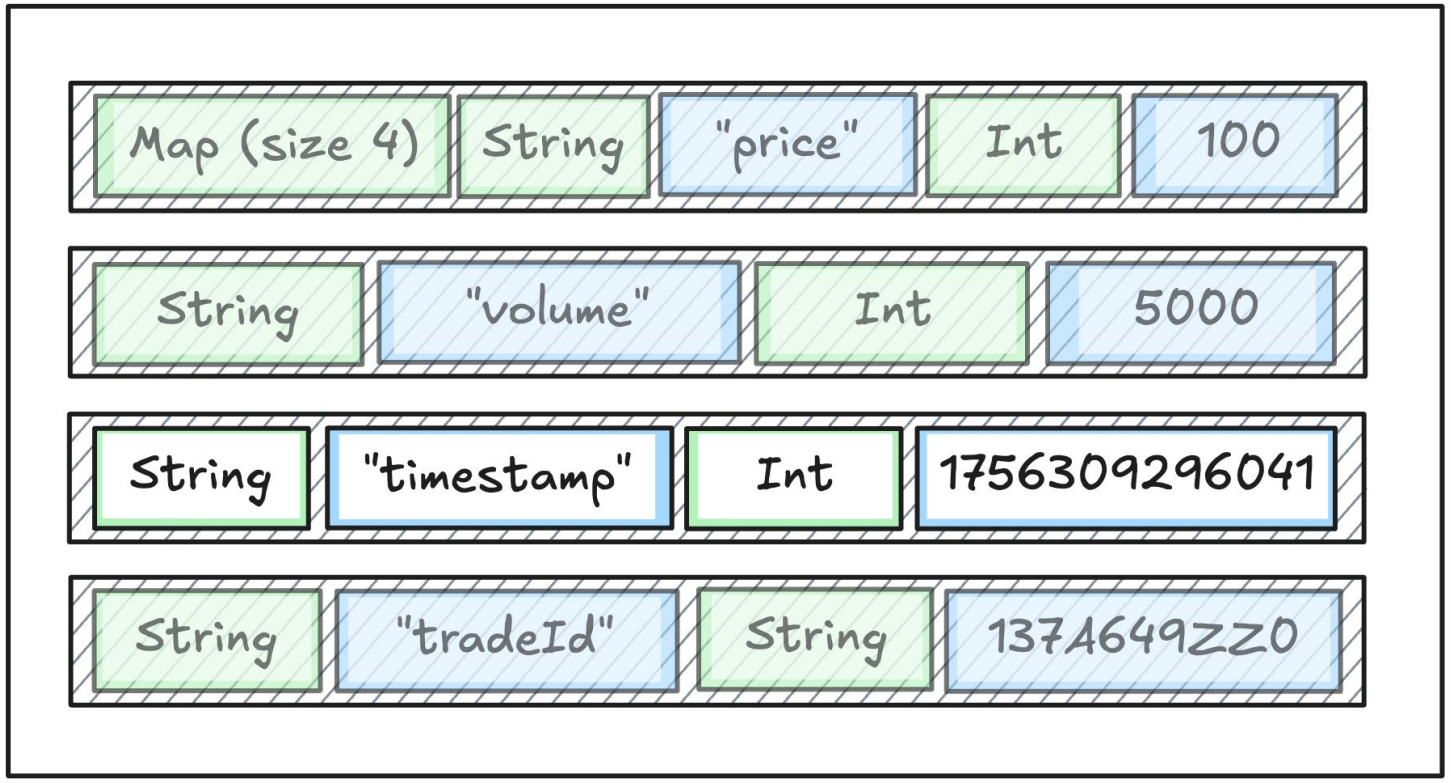

```
Value doc(buf);
Value ts = doc["timestamp"];
int64_t value = ts.asInt();
```

Bloomberg

Engineering

Baseline Serialization Format

- To implement read operations, we will do a linear search through the document, looking for the right key



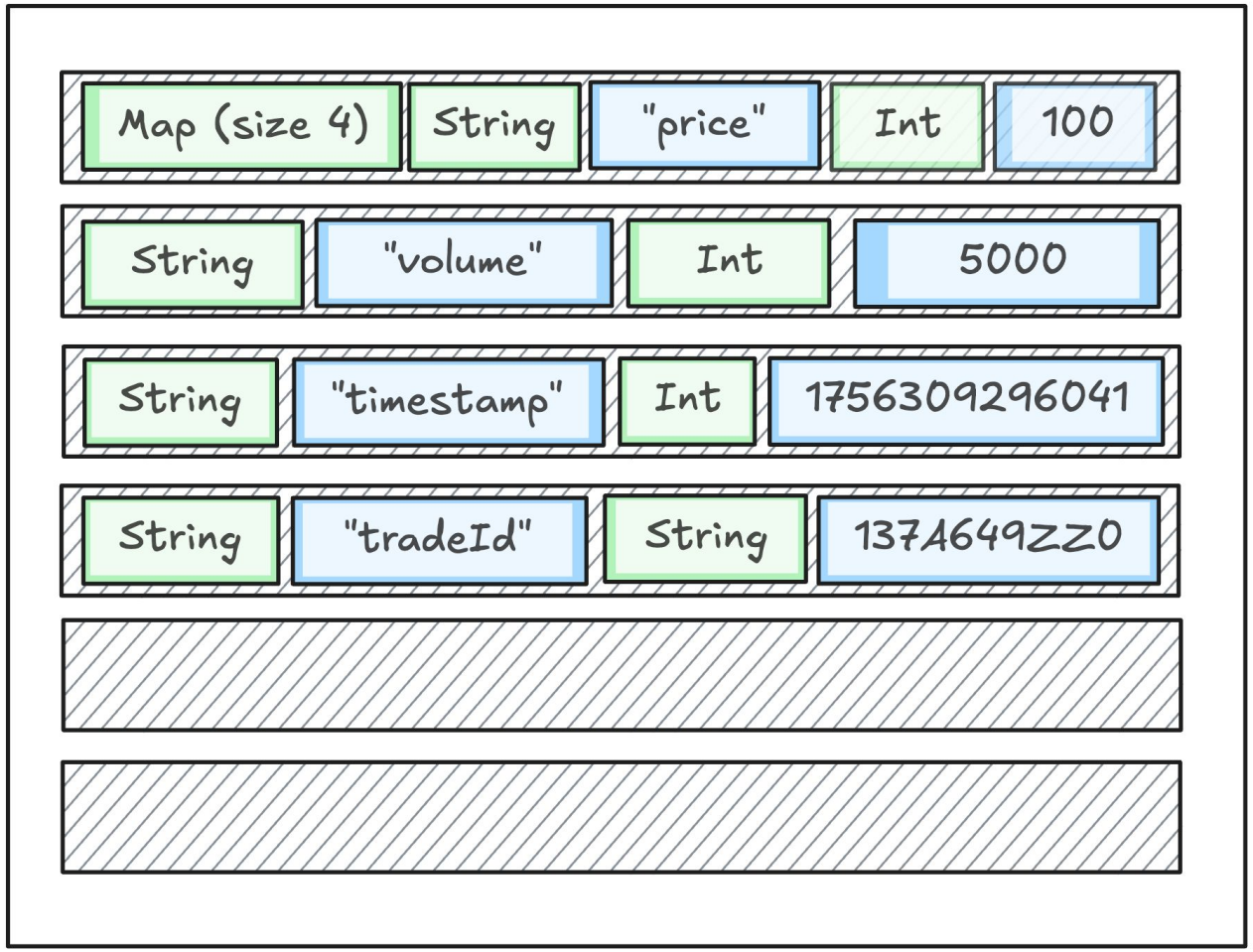

```
Value doc(buf);
Value ts = doc["timestamp"];
int64_t value = ts.asInt();
```

Bloomberg

Engineering

Baseline Serialization Format

- To make modifications, we will modify in-place, and move everything that comes after to make space



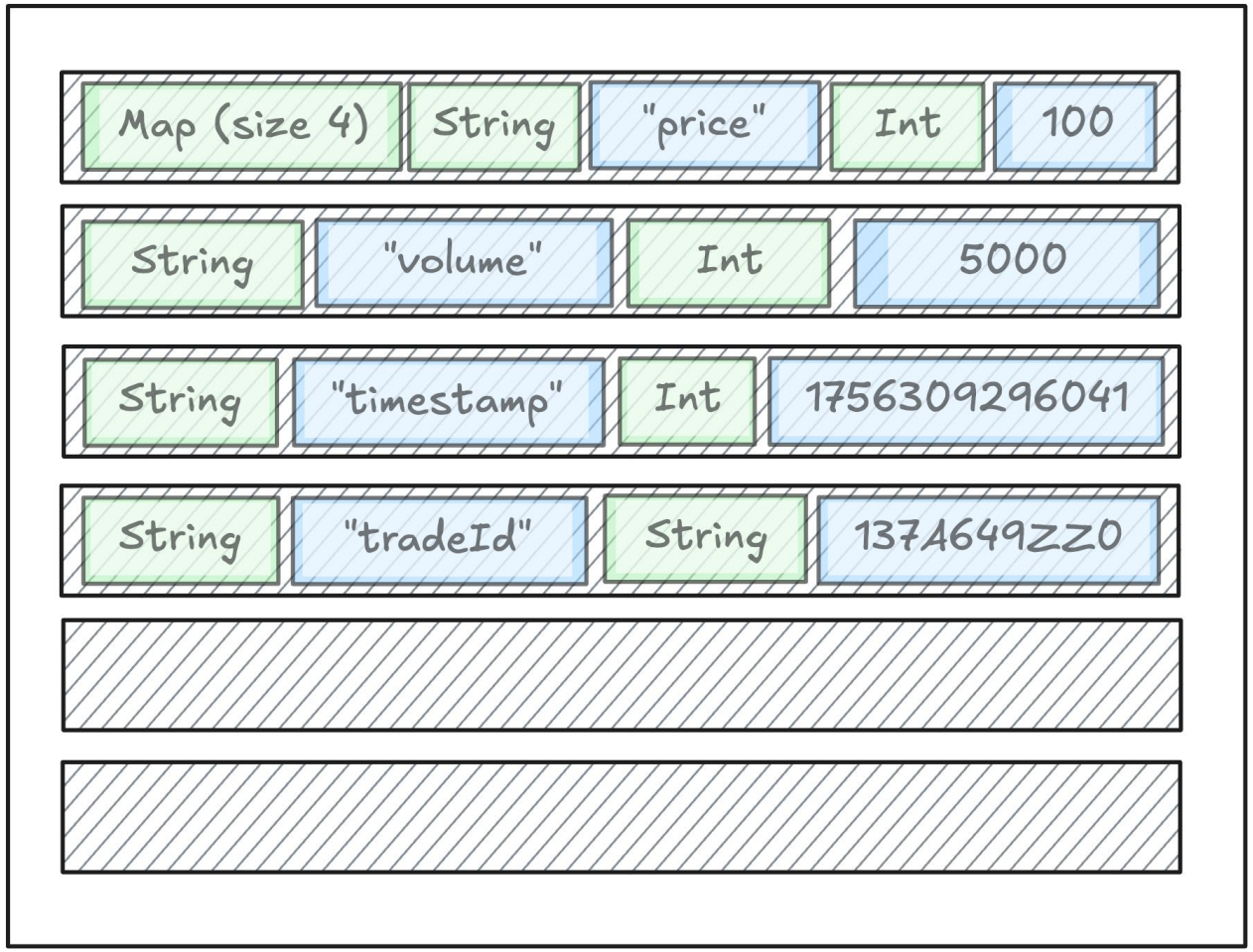

```
Value doc(buf);
Value price = doc["price"];
price.makeMap();
price.set("mantissa", 100);
price.set("exponent", -2);
```

Bloomberg

Engineering

Baseline Serialization Format

- To make modifications, we will modify in-place, and move everything that comes after to make space



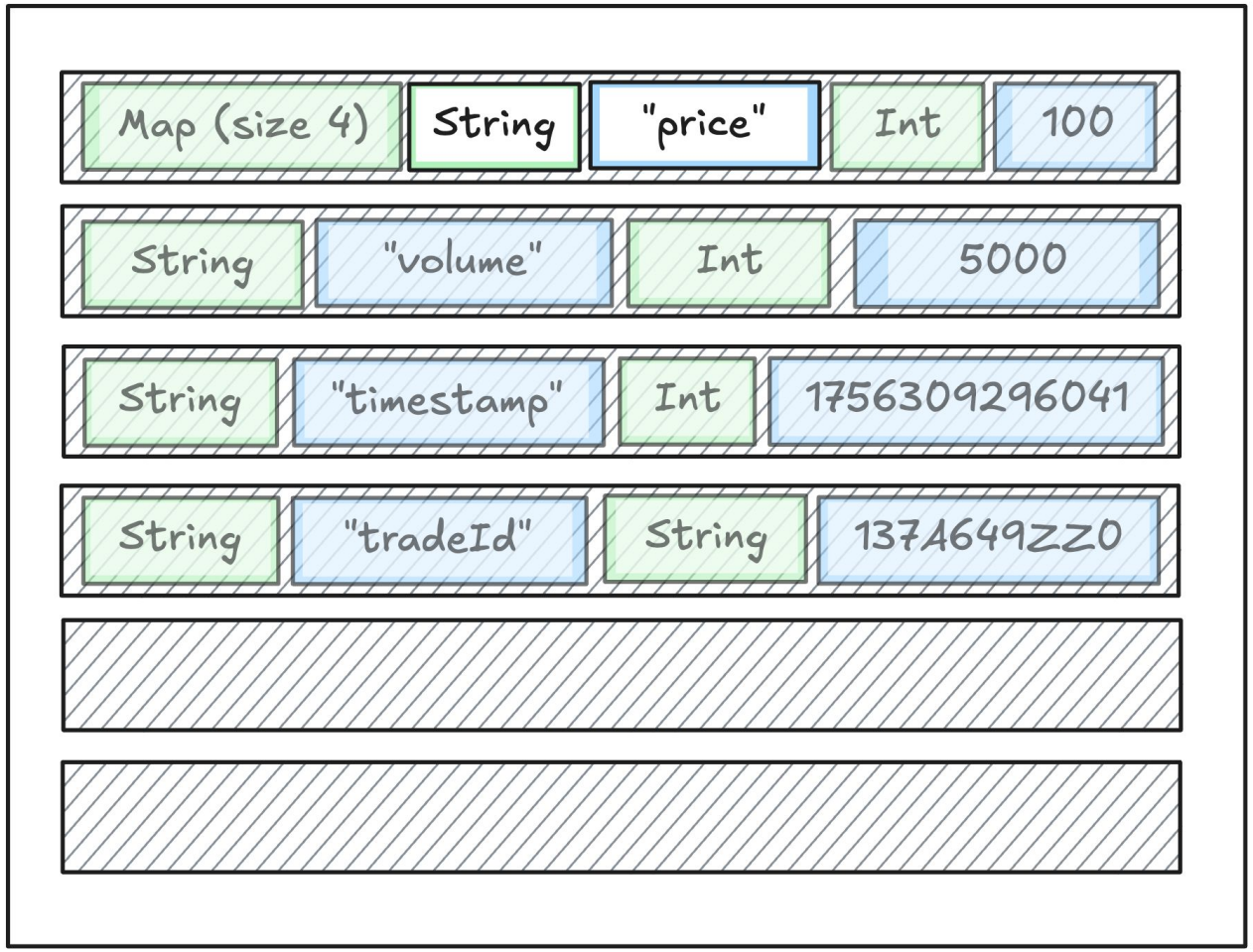

```
Value doc(buf);
Value price = doc["price"];
price.makeMap();
price.set("mantissa", 100);
price.set("exponent", -2);
```

Bloomberg

Engineering

Baseline Serialization Format

- To make modifications, we will modify in-place, and move everything that comes after to make space



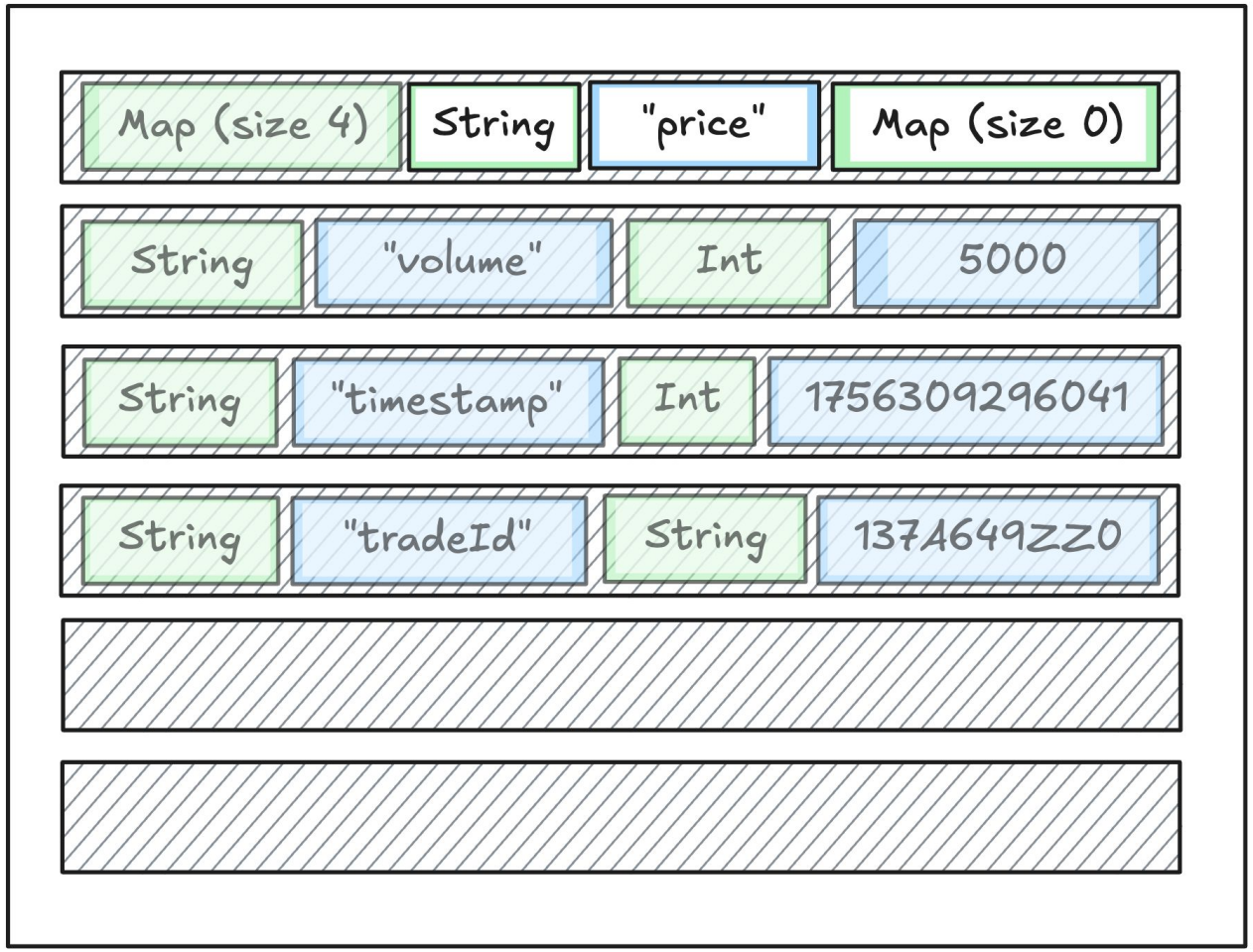

```
Value doc(buf);
Value price = doc["price"];
price.makeMap();
price.set("mantissa", 100);
price.set("exponent", -2);
```

Bloomberg

Engineering

Baseline Serialization Format

- To make modifications, we will modify in-place, and move everything that comes after to make space



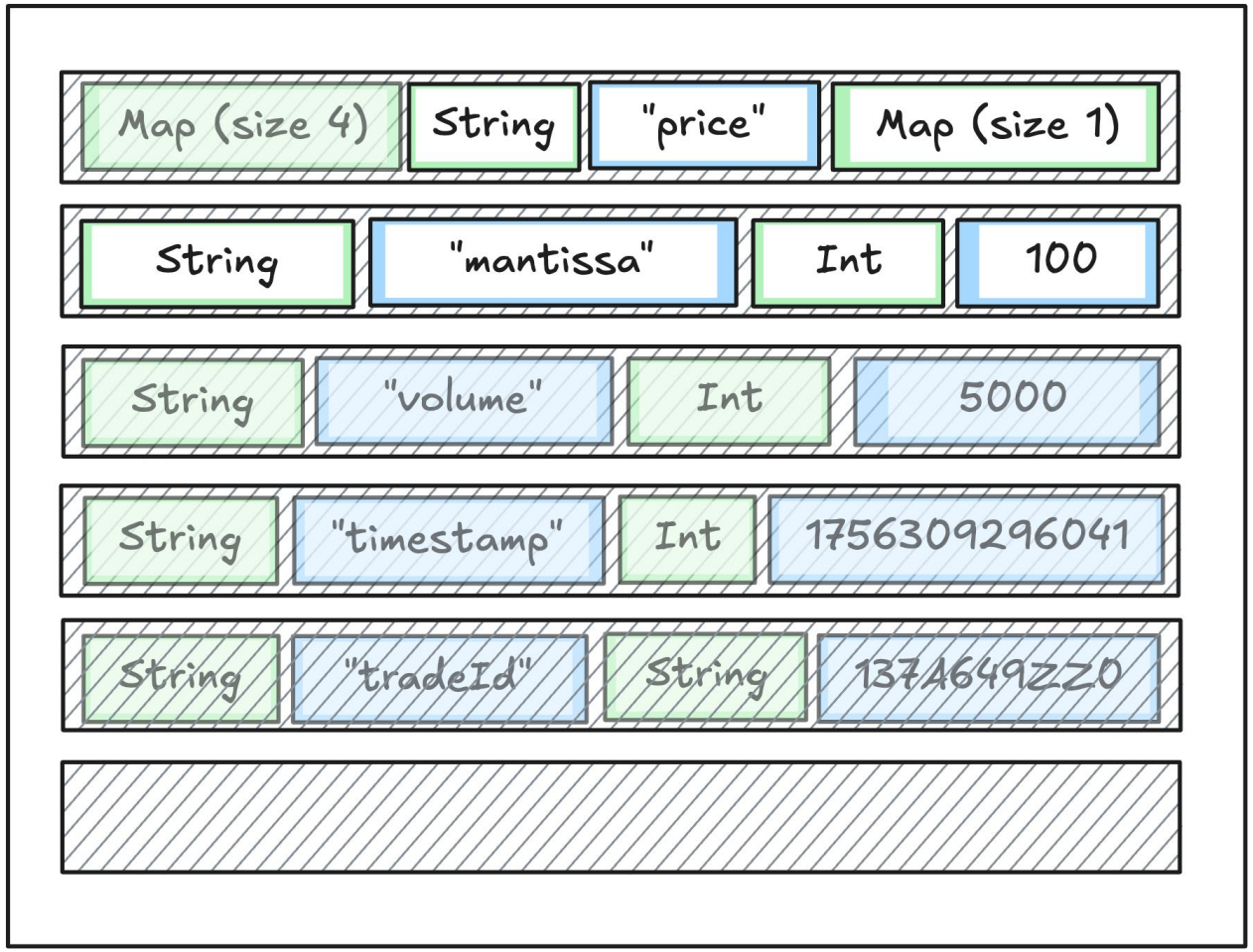

```
Value doc(buf);
Value price = doc["price"];
price.makeMap();
price.set("mantissa", 100);
price.set("exponent", -2);
```

Bloomberg

Engineering

Baseline Serialization Format

- To make modifications, we will modify in-place, and move everything that comes after to make space



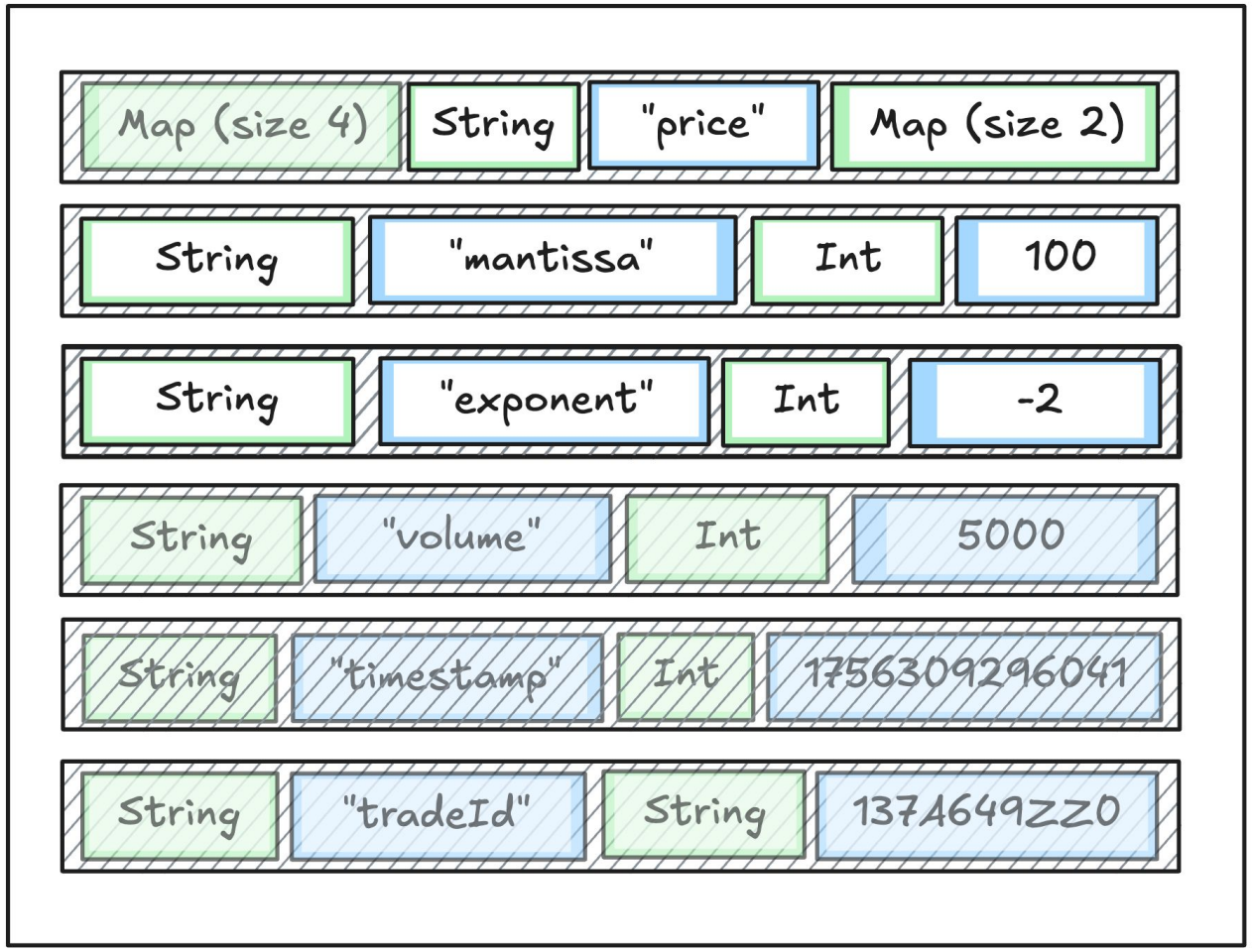

```
Value doc(buf);
Value price = doc["price"];
price.makeMap();
price.set("mantissa", 100);
price.set("exponent", -2);
```

Bloomberg

Engineering

Baseline Serialization Format

- To make modifications, we will modify in-place, and move everything that comes after to make space




```
Value doc(buf);
Value price = doc["price"];
price.makeMap();
price.set("mantissa", 100);
price.set("exponent", -2);
```

Bloomberg

Engineering

Baseline Serialization Format – Benchmarks

- So, how does this perform?
- Let’s create a simple benchmark
- We’ll parameterize our tests on:
  - Number of fields in document
  - Number of operations to perform
  - Percentage of operations that are:
    - Reads
    - Writes to existing values
    - Structural modifications (add or remove value)

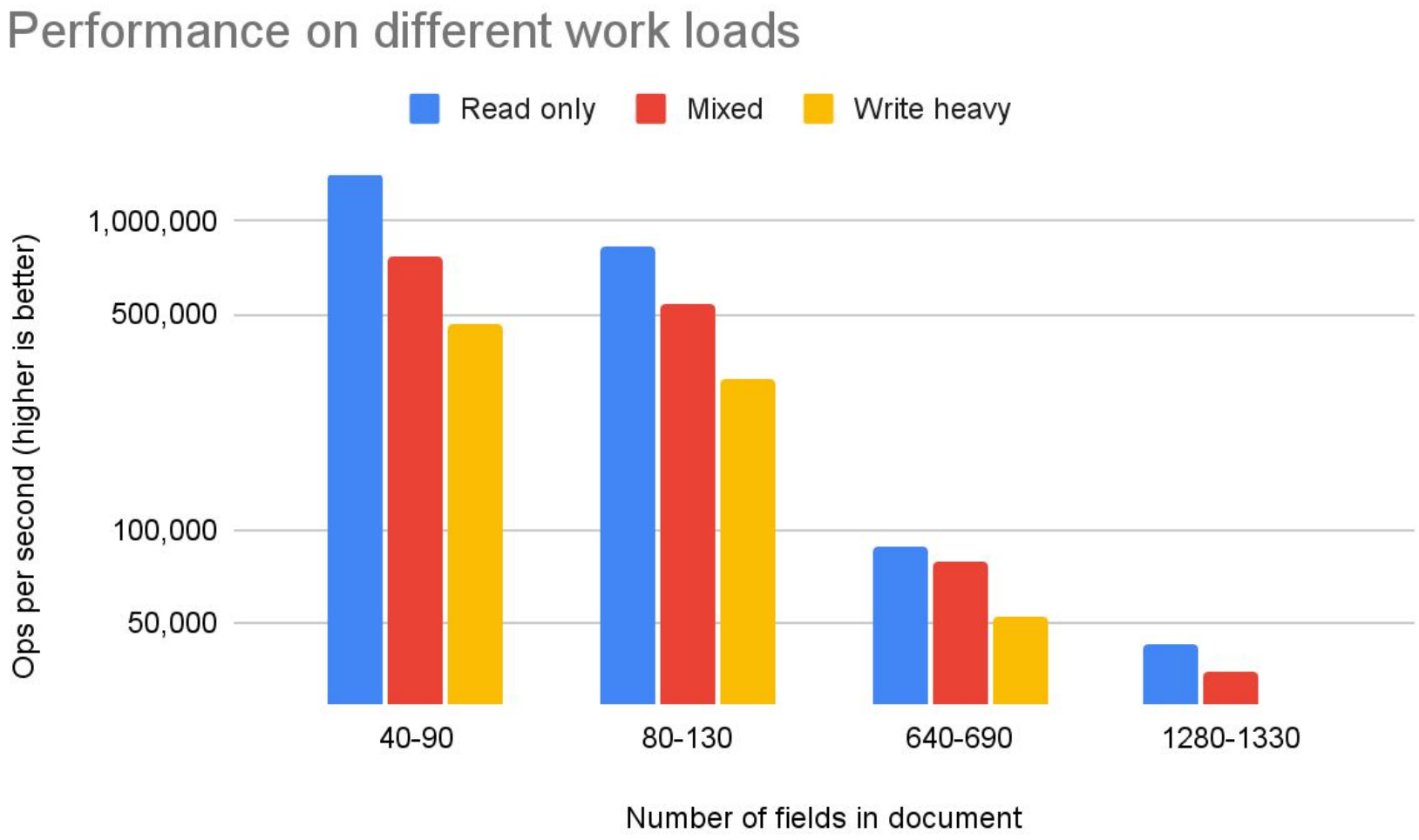
Benchmarking Setup

- Benchmark environment:
  - Linux 3.10
  - 72-core Intel Xeon Gold 6240 @ 2.6 GHz (Cascade Lake)
  - gcc 13.1.1 20230614 (Red Hat 13.1.1-4)
  - `-O2 -march=native -std=c++23`

- Machine load < 1%
- These benchmarks are synthetic and don’t necessarily represent the results you’d get in a production workload

Baseline Serialization Format – Benchmarks

- Results

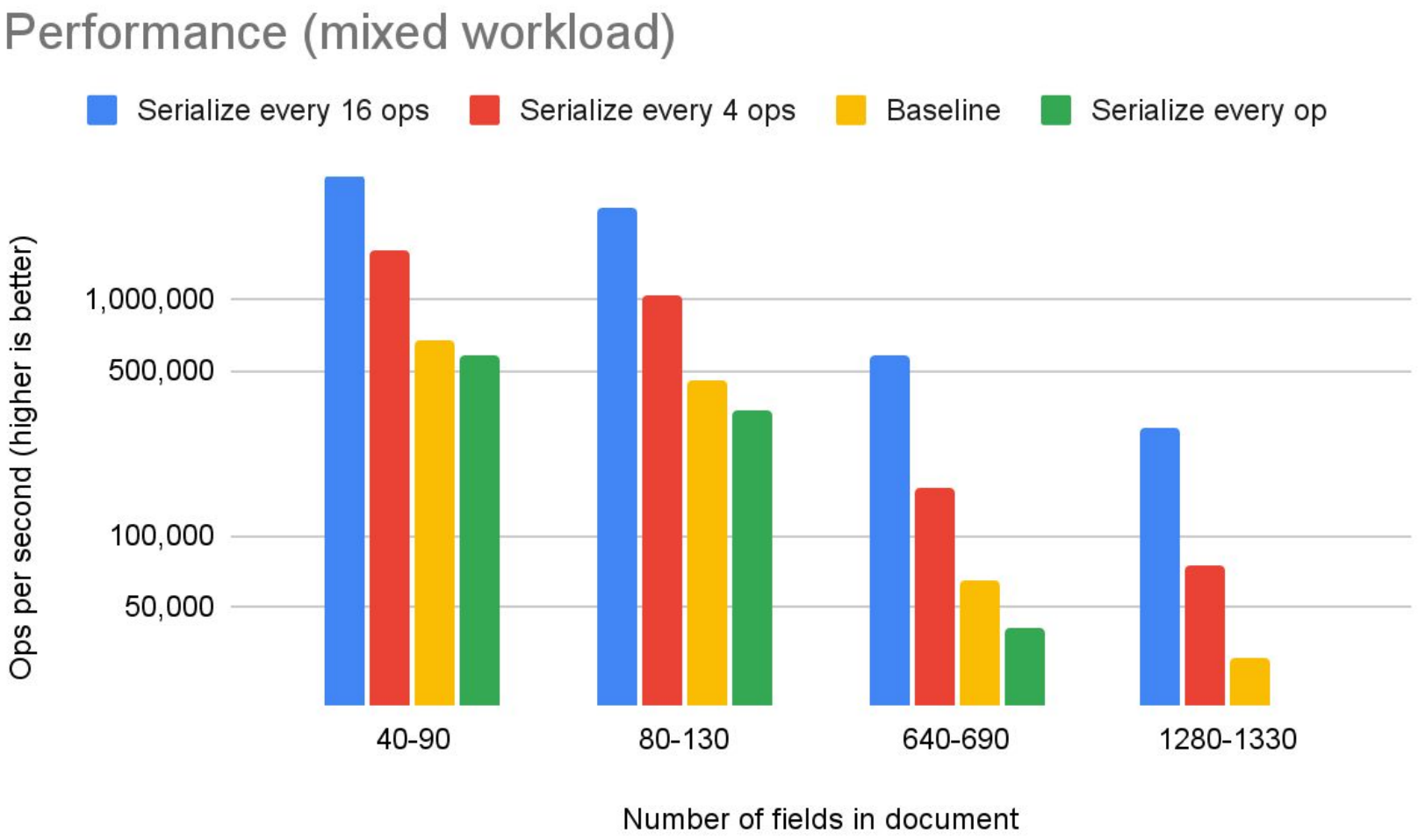


Bloomberg

Engineering

Baseline Serialization Format – Benchmarks

- Results vs. operating on an unordered_map and re-serializing



Bloomberg

Engineering

Faster Searches

How do we make searches faster?

TechAtBloomberg.com

Bloomberg

Engineering

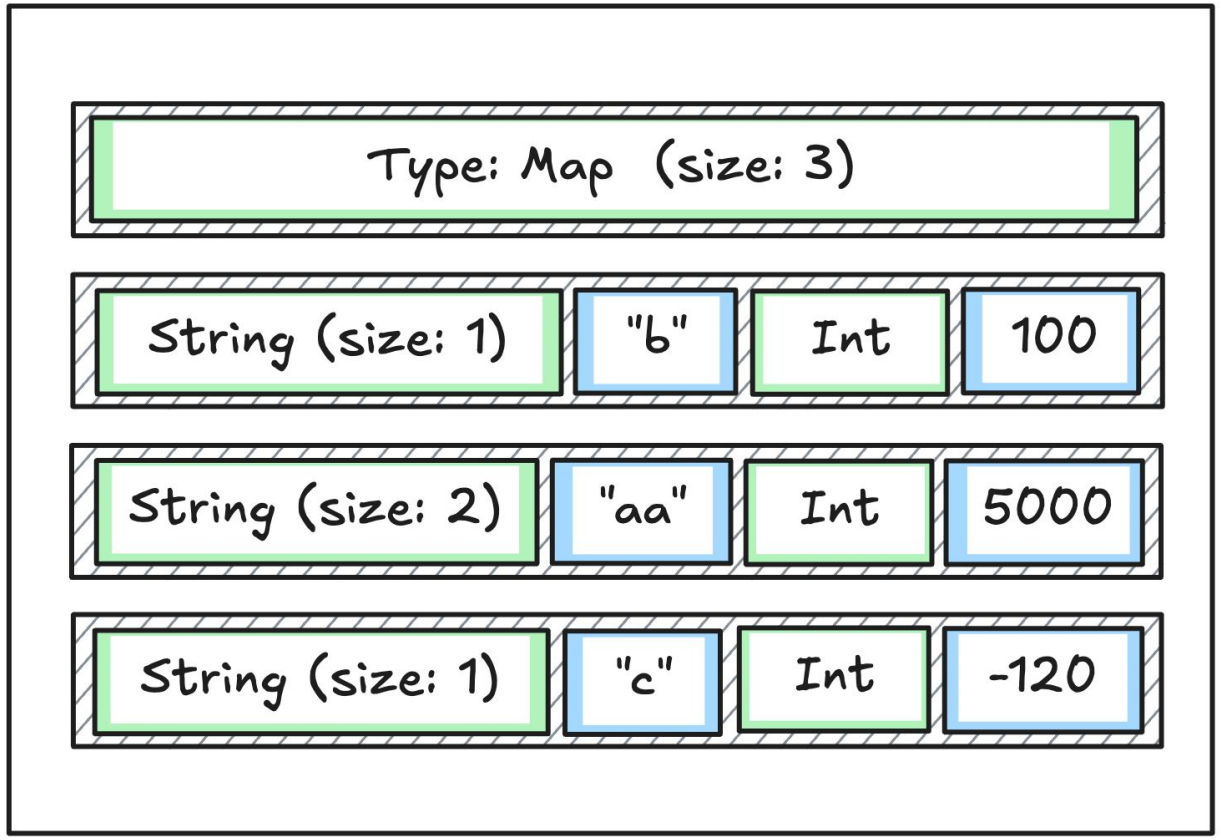
Profiling

- Profiling reveals that most of the time is spent checking the size of fields while searching for elements of interest:

```
Samples: 18K of event 'cycles:uppp', Event count (approx.): 15146430238
Overhead  Command          Shared Object             Symbol
  41.85%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::tagged::TaggedUtil::sizeOfMap
  29.72%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::tagged::TaggedUtil::sizeOfValue
   7.89%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::tagged::TaggedUtil::mapElementAt
   3.15%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::tagged::Value::operator[]
   3.05%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::adv::container::performOperationC
   1.82%  adv_container_b  libc-2.17.so              [.] __memcmp_sse4_1
   1.75%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::adv::ContainerBenchmark::prepareC
   1.67%  adv_container_b  adv_container_benchmarks  [.] std::mersenne_twister_engine<unsigned long, 32ul, 62
   1.16%  adv_container_b  adv_container_benchmarks  [.] std::uniform_int_distribution<char>::operator()<std:
   0.92%  adv_container_b  libc-2.17.so              [.] __memmove_ssse3_back
   0.82%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::adv::ContainerBenchmark::setupIni
   0.53%  adv_container_b  libc-2.17.so              [.] _int_malloc
   0.49%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::adv::ContainerBenchmark::getUnuse
   0.48%  adv_container_b  adv_container_benchmarks  [.] experimental::ser::adv::ContainerBenchmark::generate
```

Faster Searches

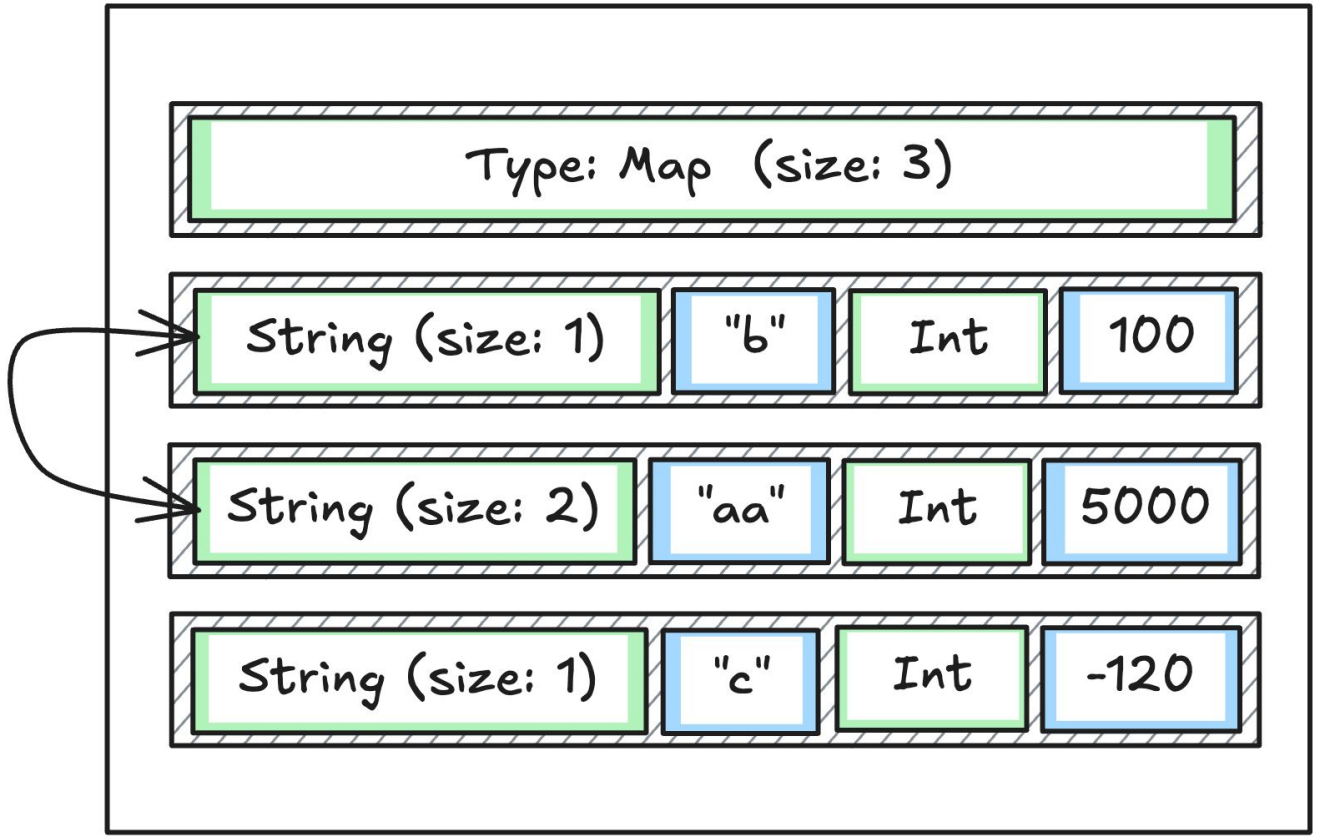
- A lot of time is spent skipping over elements
- The key we are searching for could appear anywhere in the map
- What if we put our keys in sorted order?



TechAtBloomberg.com

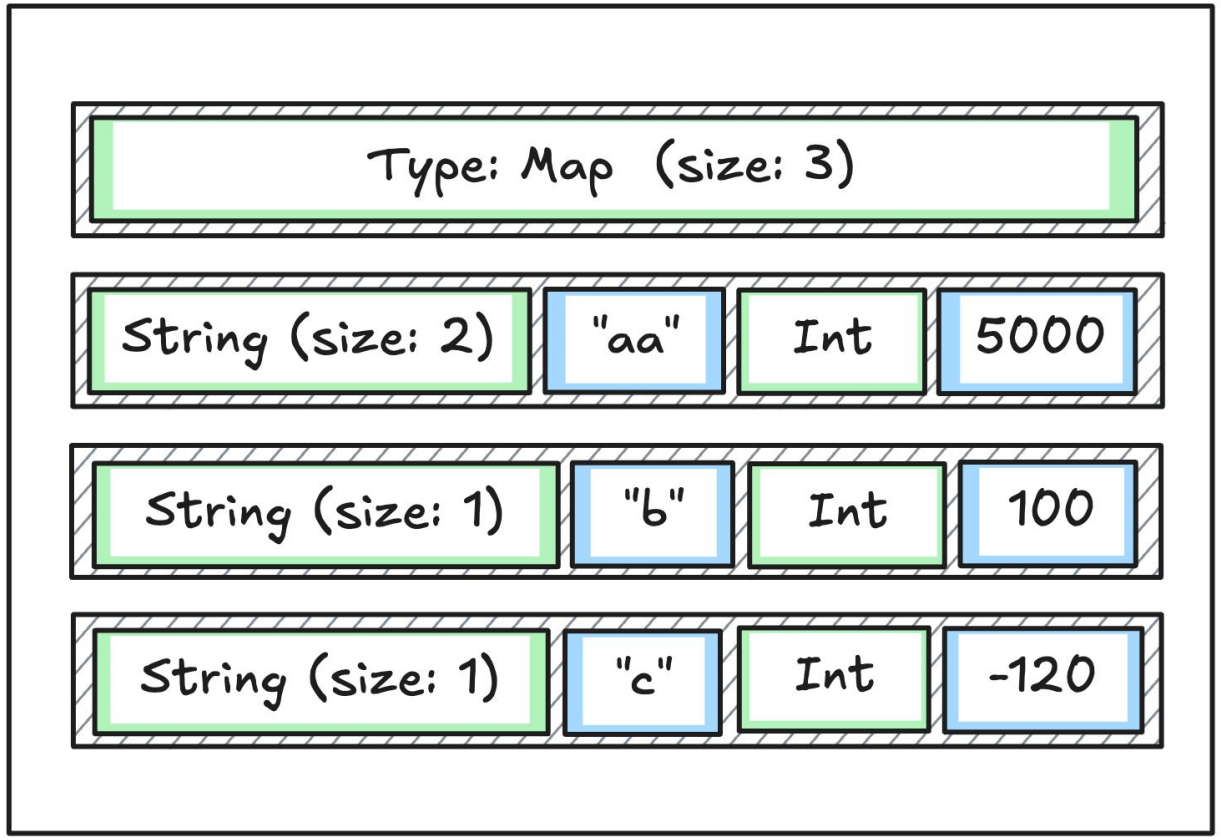
Faster Searches

- A lot of time is spent skipping over elements
- The key we are searching for could appear anywhere in the map
- What if we put our keys in sorted order?



Faster Searches

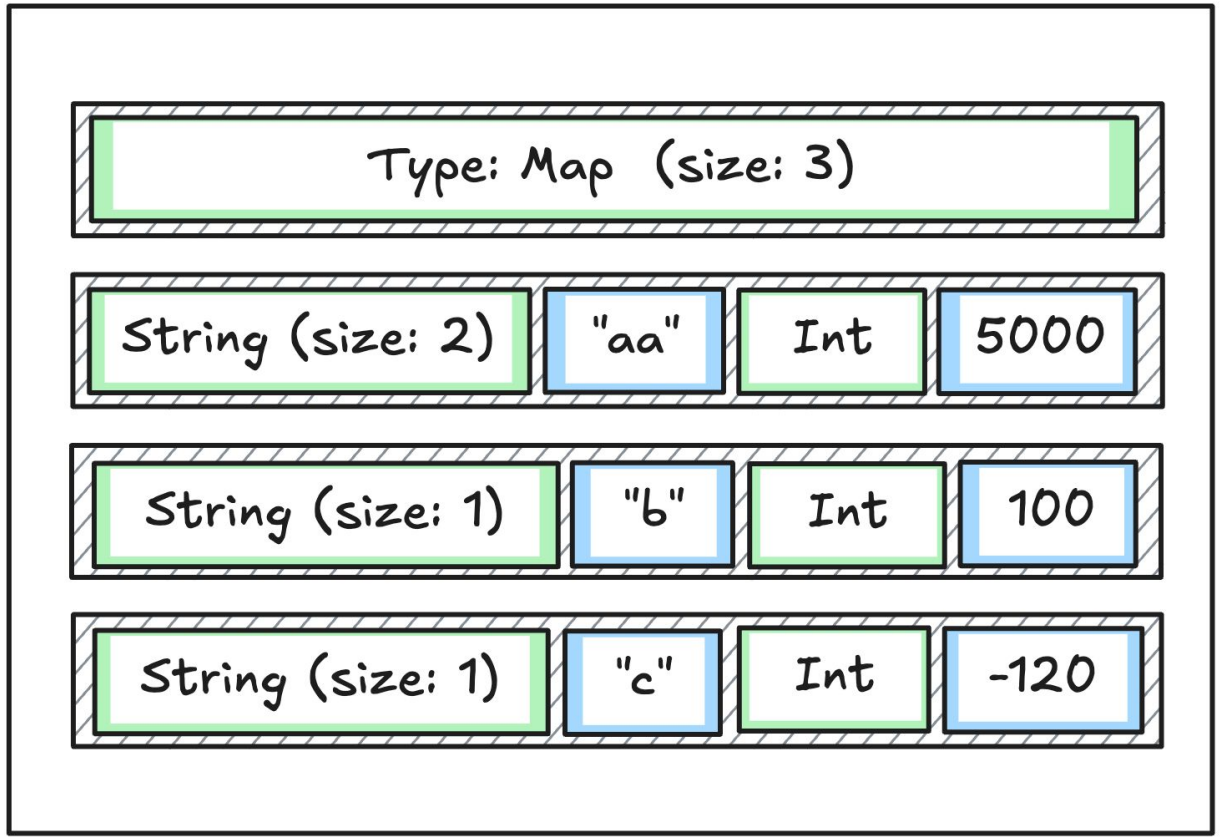
- A lot of time is spent skipping over elements
- The key we are searching for could appear anywhere in the map
- What if we put our keys in sorted order?



TechAtBloomberg.com

Faster Searches

- A lot of time is spent skipping over elements
- The key we are searching for could appear anywhere in the map
- What if we put our keys in sorted order?
- Now can we do binary search?
  - Not quite, because our keys and values are variable sized
  - This makes it hard to jump to the middle key



TechAtBloomberg.com

Faster Searches

Idea 1: Store a table of offsets at the start of the map

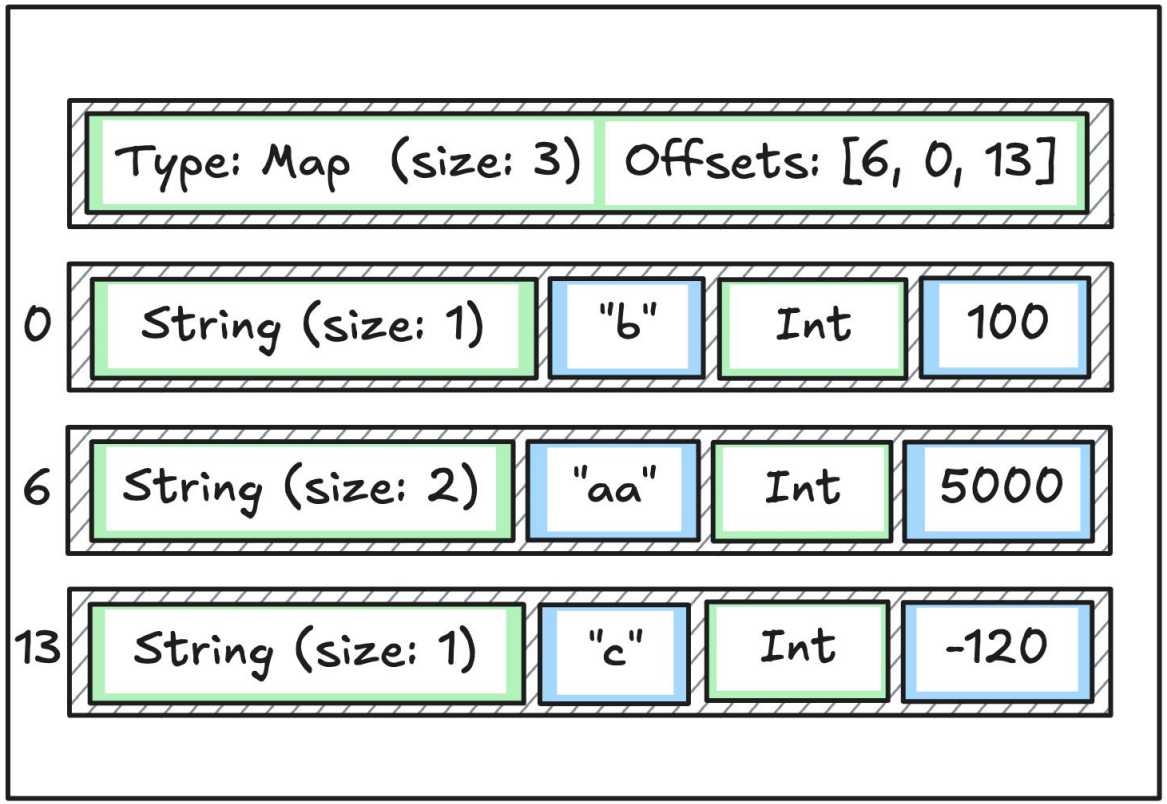
TechAtBloomberg.com

Bloomberg

Engineering

Faster Searches

- Insert an offset table at the start of the map
- This makes it easy to jump to certain elements
- The actual elements don’t need to be in sorted order, only the offsets

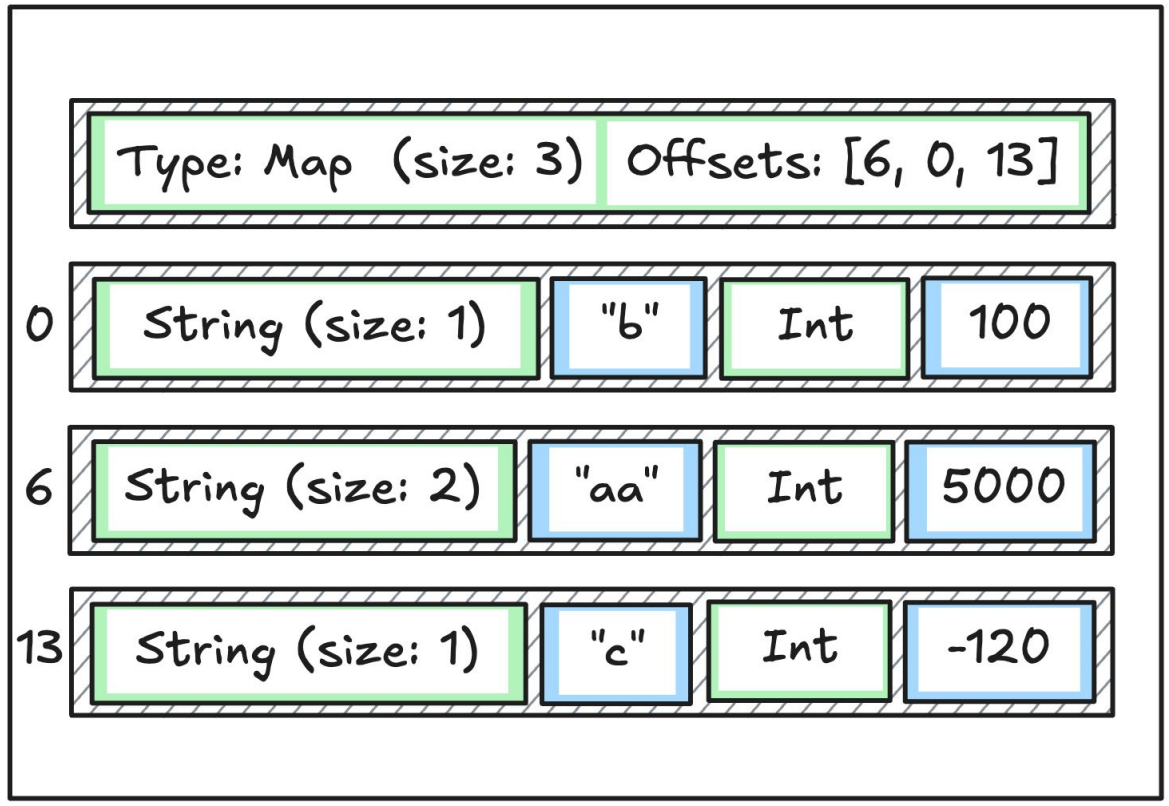


TechAtBloomberg.com

Faster Searches

- Insert an offset table at the start of the map
- This makes it easy to jump to certain elements
- The actual elements don’t need to be in sorted order, only the offsets

- In this example, the offsets tell us that key “aa” comes first, followed by “b”, followed by “c”

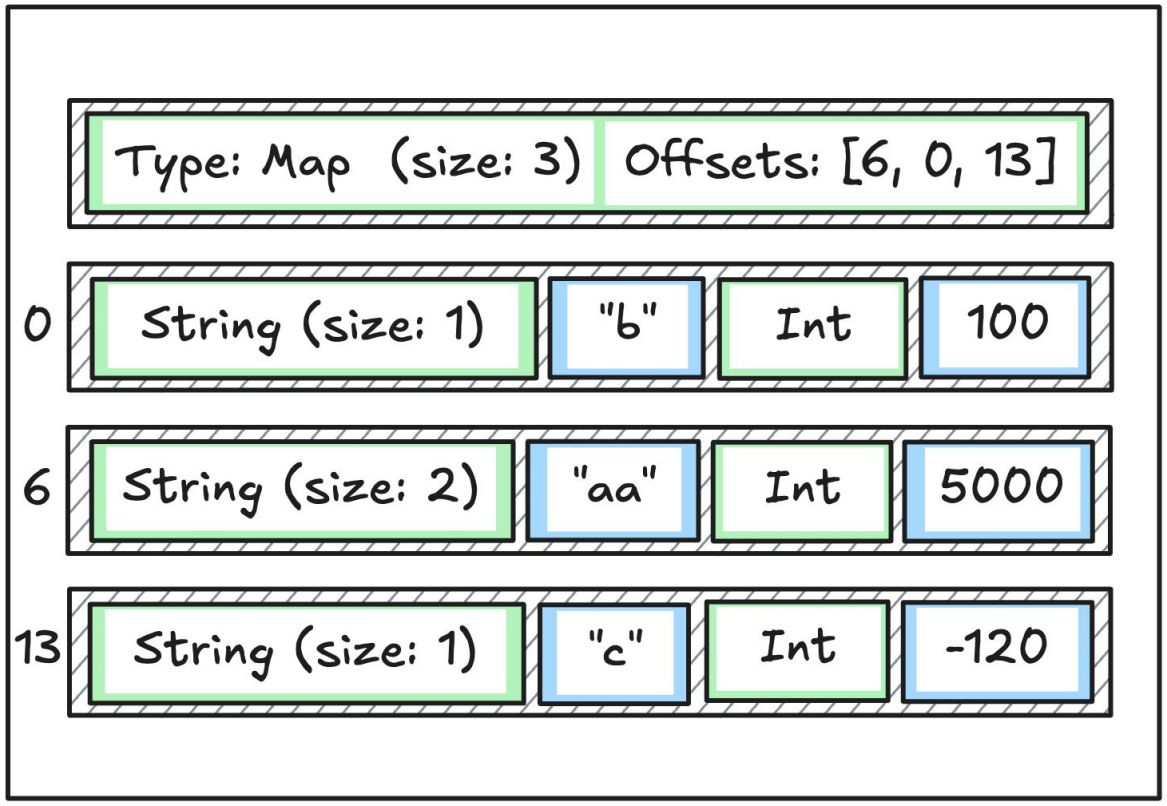


TechAtBloomberg.com

Faster Searches

- Insert an offset table at the start of the map
- This makes it easy to jump to certain elements
- The actual elements don’t need to be in sorted order, only the offsets

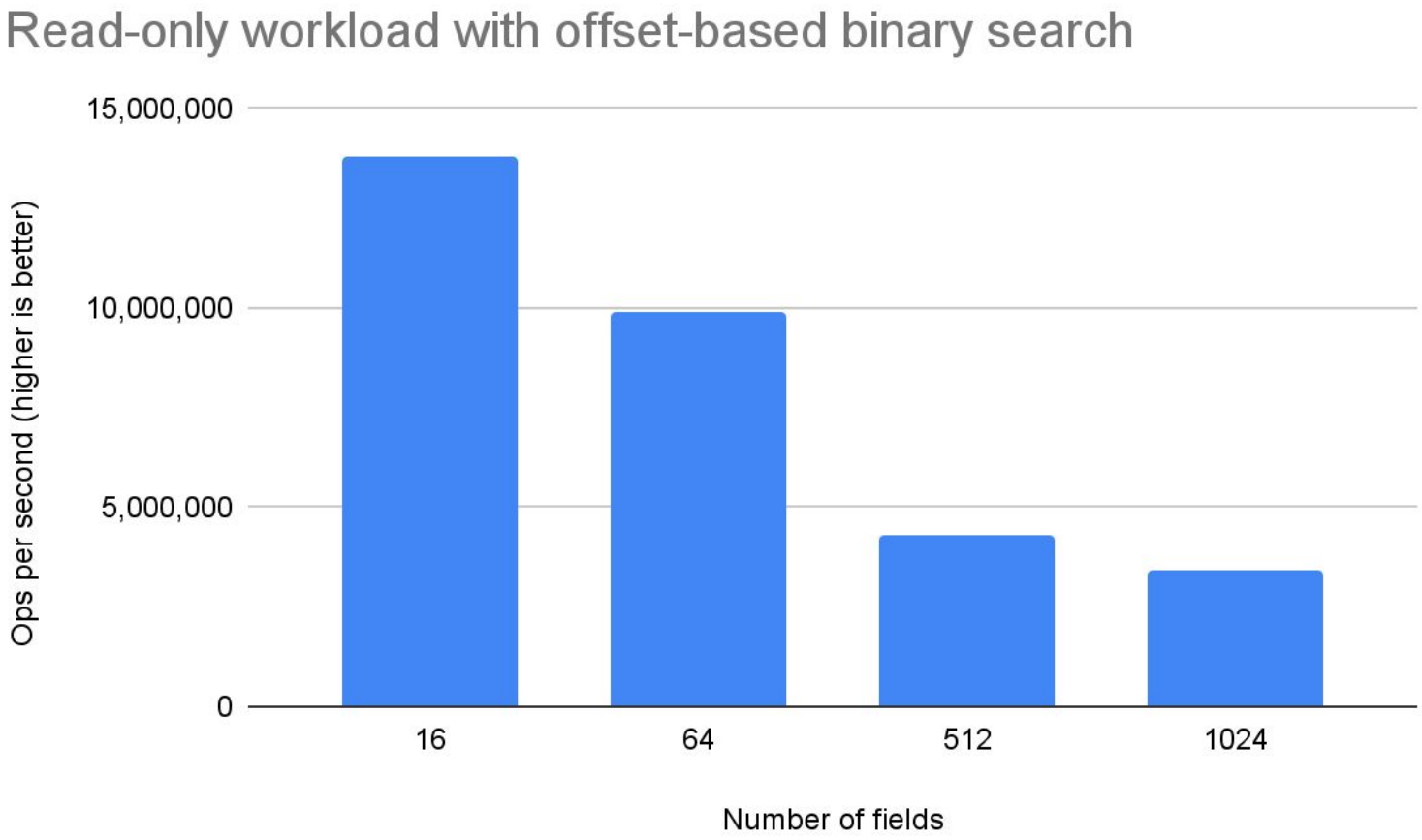
- In this example, the offsets tell us that key “aa” comes first, followed by “b”, followed by “c”
- Now it’s easier to do binary search



TechAtBloomberg.com

Faster Searches – Benchmark Results

- Benchmark results for binary search on offsets:



Bloomberg

Engineering

Faster Searches

Idea 1: Store a table of offsets at the start of the map
Idea 2: Store hashes of keys at the start of the map too

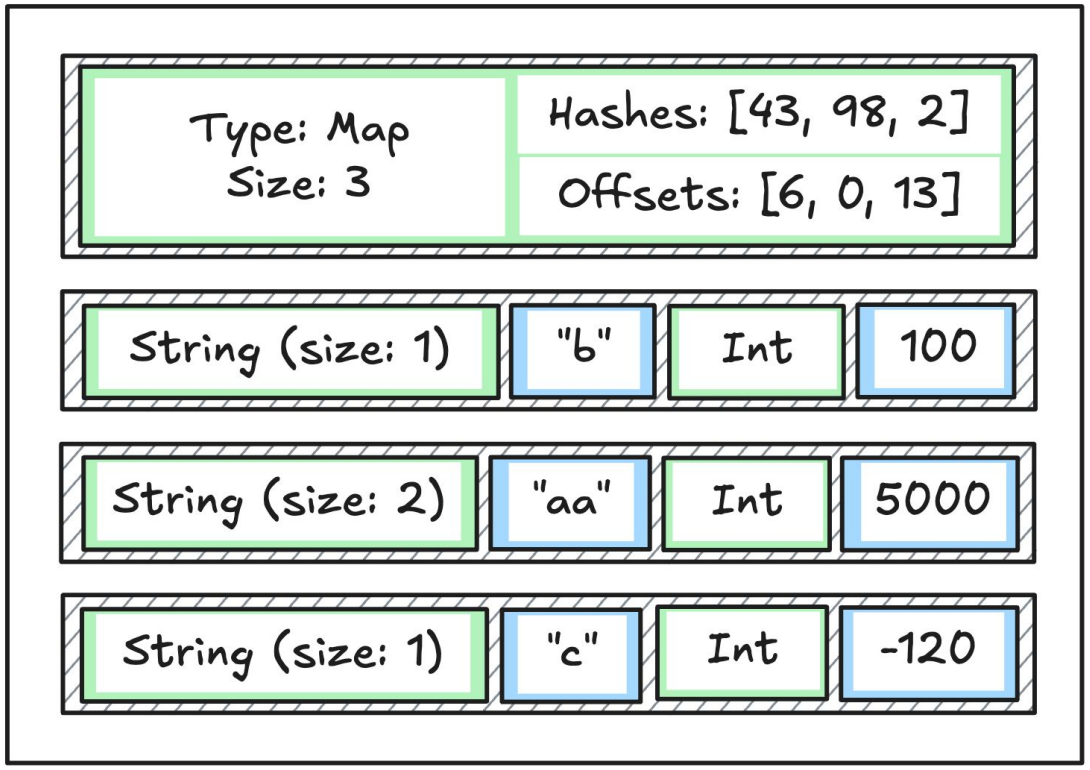
TechAtBloomberg.com

Bloomberg

Engineering

Faster Searches

- A further enhancement:
  - Store an array of hashes at the start of our map
  - Search these first, before checking actual keys
    - Comparing small integers is faster than comparing strings
    - Hashes are contiguous in memory, which is more cache-friendly

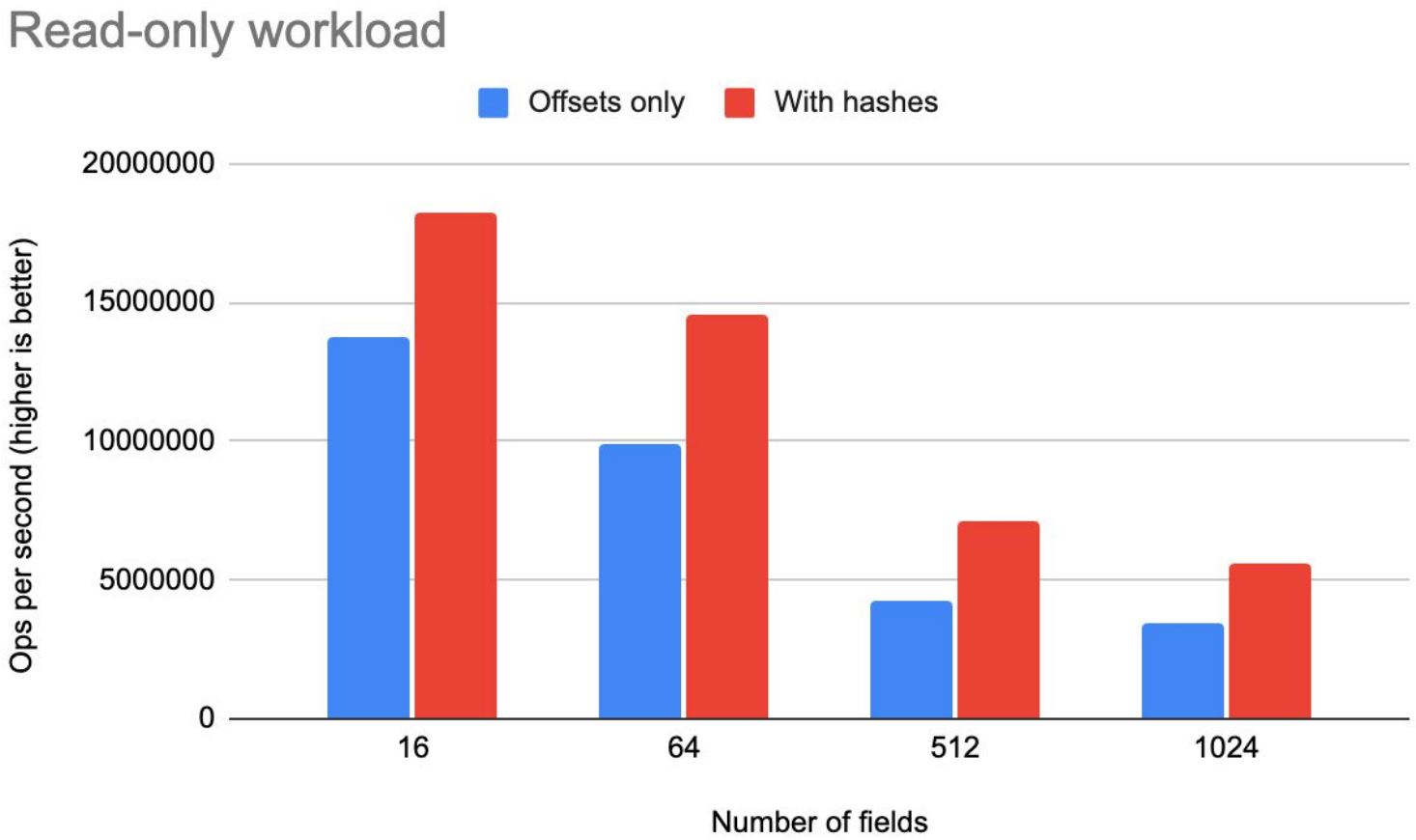


Bloomberg

Engineering

Faster Searches – Benchmark Results

- How does this scheme compare to the previous one?



Bloomberg

Engineering

Faster Searches – Benchmark Results

- Can we do even faster than that?
- perf shows that a lot of the cost is in this if condition in std::lower_bound
- **`sub`** is cheap, so this probably indicates the previous branch is mispredicted

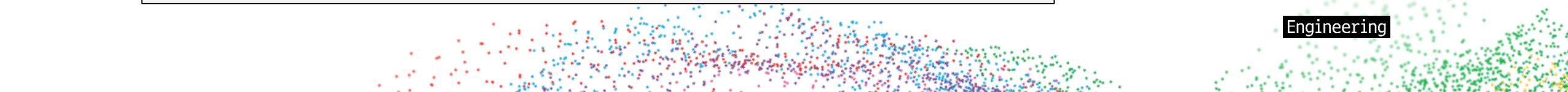
```
                _ZSt13__lower_boundIPKtiN9__gnu_cxx5__ops14_Iter_less_valEET_S5_S5_RKT0_T1_():
                          _ForwardIterator __middle = __first;
                          std::advance(__middle, __half);
                          if (__comp(__middle, __val))
  2.14                      cmp    %edi,%r15d
  2.63                    → jle    239838 <experimental::ser::adv::sortedstring::Container::search(std::basic_s
                              {
                                __first = __middle;
                                ++__first;
                                __len = __len - __half - 1;
 22.64                        sub    %rsi,%rax
                                ++__first;
  0.05                        lea    0x2(%rdx),%rbx
                                __len = __len - __half - 1;
  0.09                        dec    %rax
                          while (__len > 0)
  0.33                        test   %rax,%rax
```

Binary Search – Mispredicted Branches

- The mispredicted branch is the one where we decide whether to look at the lower or upper half of the sub-array
- This makes sense – we really can't predict whether the value will be in the upper or lower half each time around the loop

```
const T *lower_bound(const T *begin, const T *end, T value) {
    int64_t len = end - begin;
    while (len > 0) {
        const int64_t step = count / 2;
        if (begin[step] < value) {
            begin = &begin[step + 1];
            len -= step + 1;
        }
        else len = step;
    }
    return begin;
}
```

This branch is often mispredicted



Bloomberg

Engineering

Binary Search – Mispredicted Branches

- To fix this, we can remove the branching and use a conditional move instruction instead
- This is sometimes called “branchless” programming

Faster Searches

Idea 1: Store a table of offsets at the start of the map
Idea 2: Store hashes of keys at the start of the map too
Idea 3: Use branchless binary search

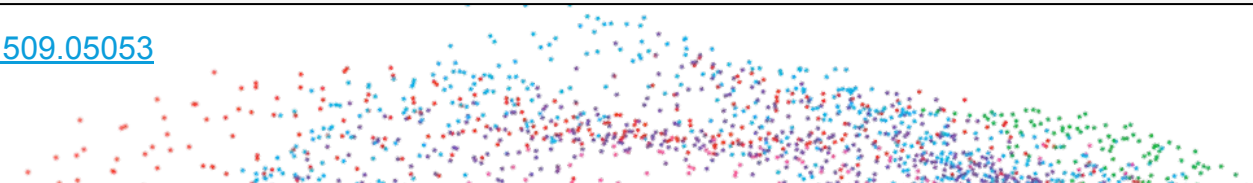
Branchless Binary Search

- “Array Layouts for Comparison-Based Searching” by Khuong and Morin [1] has a good implementation of branchless binary search
- It looks something like this:

```
const T *search(const T *begin, const T *end, T key)
{
    const T *base = begin;
    int64_t len = end - begin;
    while (len > 1) {
        const int64_t half = len / 2;
        if (base[half] < key) {
            base += half;
        }
        len -= half;
    }
    return *base < key ? base + 1 : base;
}
```

Doing only one assignment in the if statement makes the compiler more likely to generate a CMOV instruction

[1]: https://arxiv.org/abs/1509.05053



Bloomberg

Engineering

Branchless Binary Search – Perf Report

- Our comparison and assignment in the loop is done using cmov:

```
                _ZNK12experimental3ser3adv12sortedstring9Container10binsearch3EPKtS5_t():
 0.01             cmp    $0x2,%r9
                → jbe    2490b0 <int experimental::ser::adv::performOperationGeneric<experim
                  mov    $0xffffffffffffffff,%rax
 0.22             nop
 7.47             mov    %r13,%rdx
 2.91             sar    %rdx
 0.03             lea    (%rdx,%rax,1),%rsi
18.68             cmp    %r11w,(%r14,%rsi,2)
 4.33             cmovb  %rsi,%rax
                  sub    %rdx,%r13
                  cmp    $0x1,%r13
 0.28             jg     248dc8 <int experimental::ser::adv::performOperationGeneric<experim
 2.03             lea    0x2(%r14,%rax,2),%r13
                _ZNK12experimental3ser3adv12sortedstring9Container6searchESt17basic_string_v
 1.22             cmp    %r13,%r8
                → je     249130 <int experimental::ser::adv::performOperationGeneric<experim
```

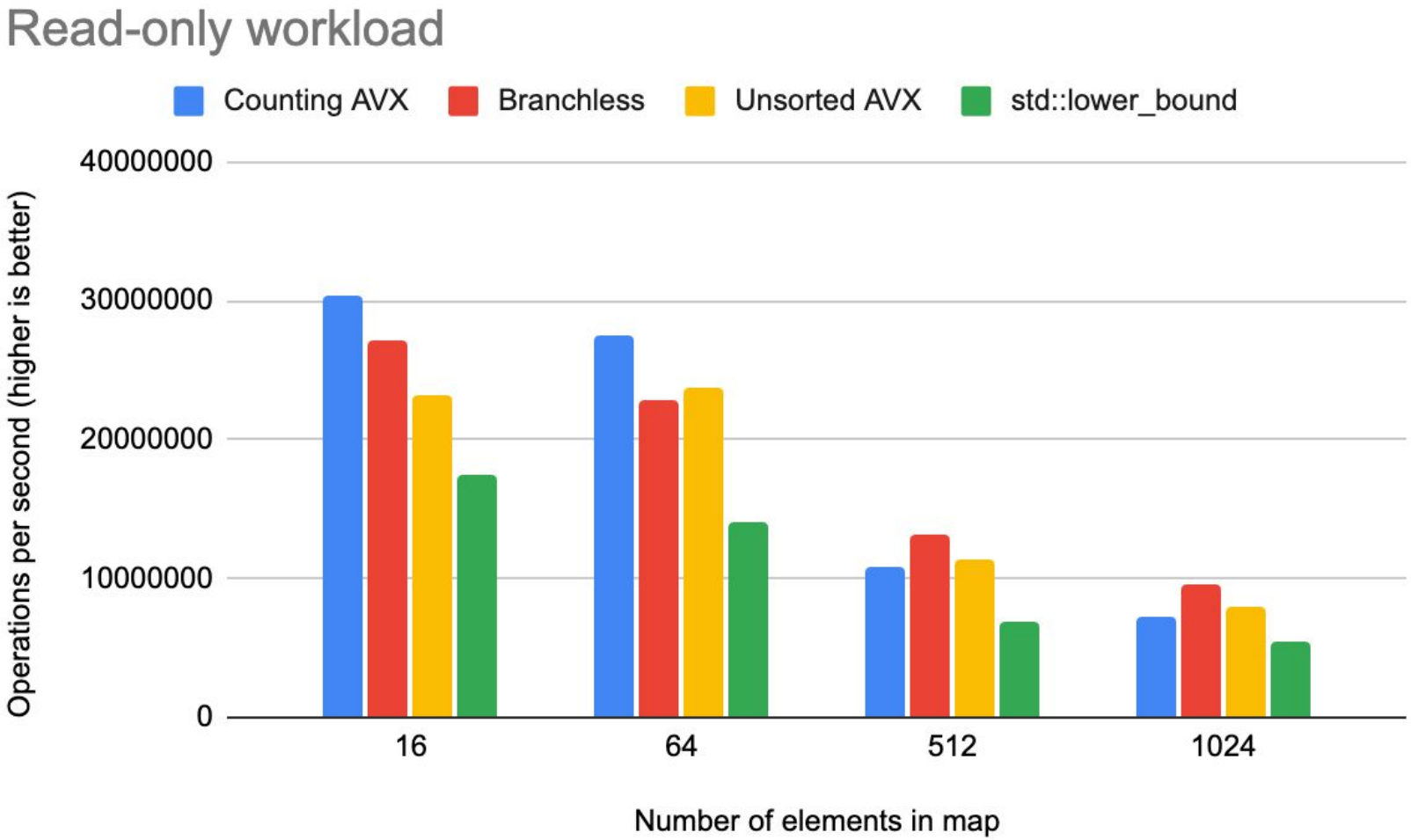
Other Search Algorithms – Side Note

- I went down a bit of a rabbit hole and tried other accelerated search algorithms:
  - Vectorized counting linear search on a sorted array
  - Vectorized standard linear search on an unsorted array

- For a good article on these and more, see this blog post: https://dirtyhandscoding.github.io/posts/performance-comparison-linear-search-vs-binary-search.html

Faster Searches – Benchmark Results

- Comparison of all hash-search methods:



Bloomberg

Engineering

Faster Searches

Idea 1: Store a table of offsets at the start of the map
Idea 2: Store hashes of keys at the start of the map too
Idea 3: Use branchless binary search
I’m pretty happy with search performance now!

Faster Modifications

How do we make modifications faster?

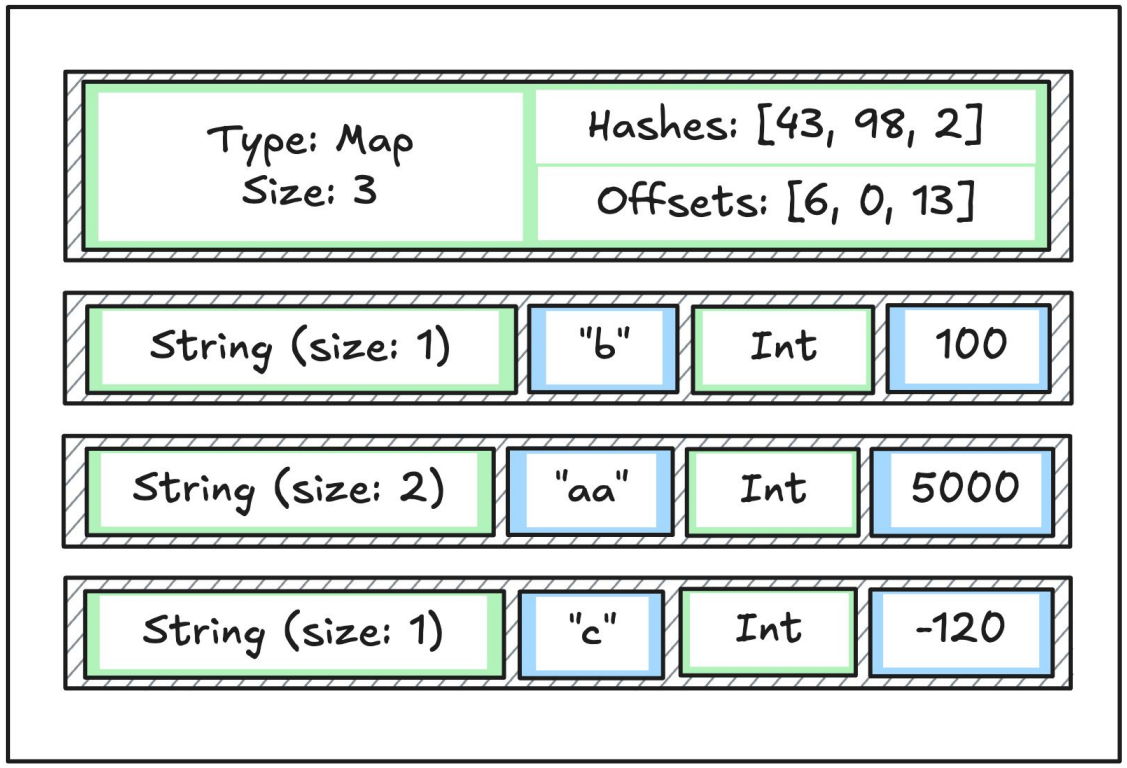
TechAtBloomberg.com

Bloomberg

Engineering

Making our Format Modifiable

- In our baseline format, we performed modifications by just moving everything around
- Now that we've introduced offsets, this is problematic
- Why?

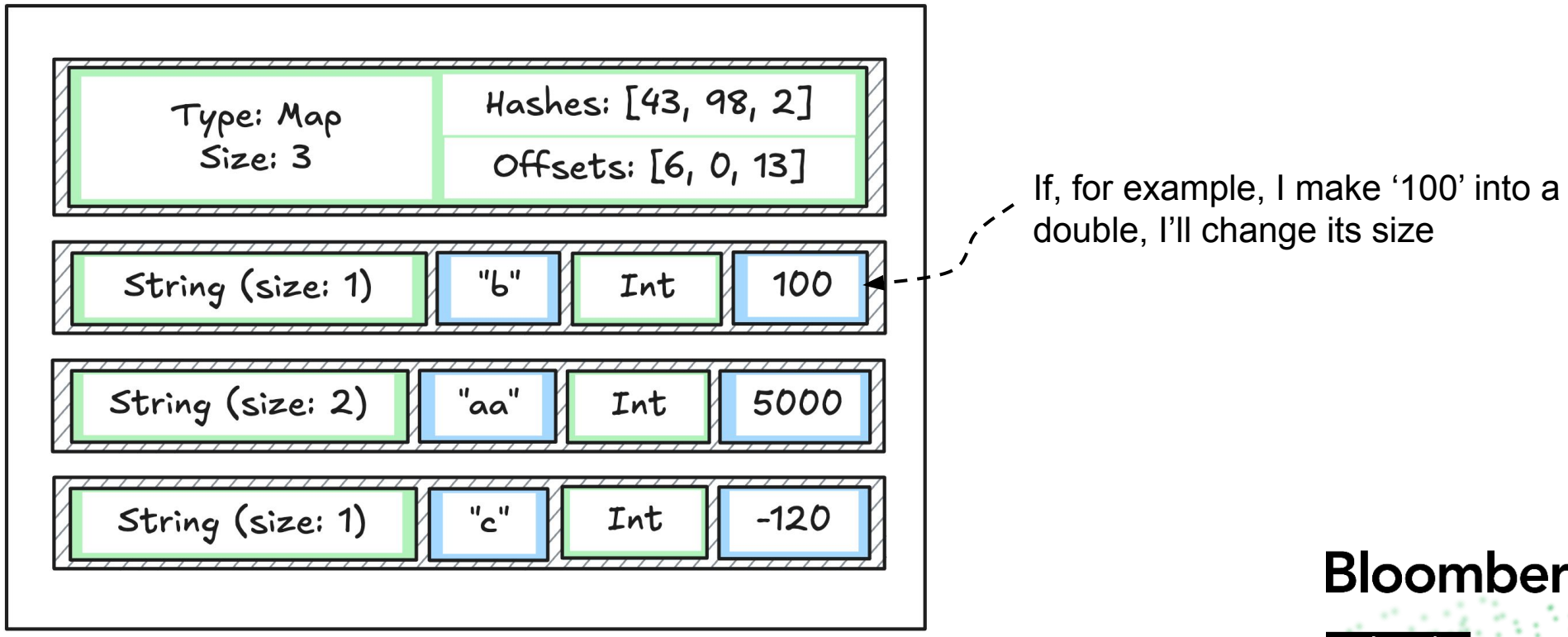


Bloomberg

Engineering

Making our Format Modifiable

- In our baseline format, we performed modifications by just moving everything around
- Now that we've introduced offsets, this is problematic
- Why?

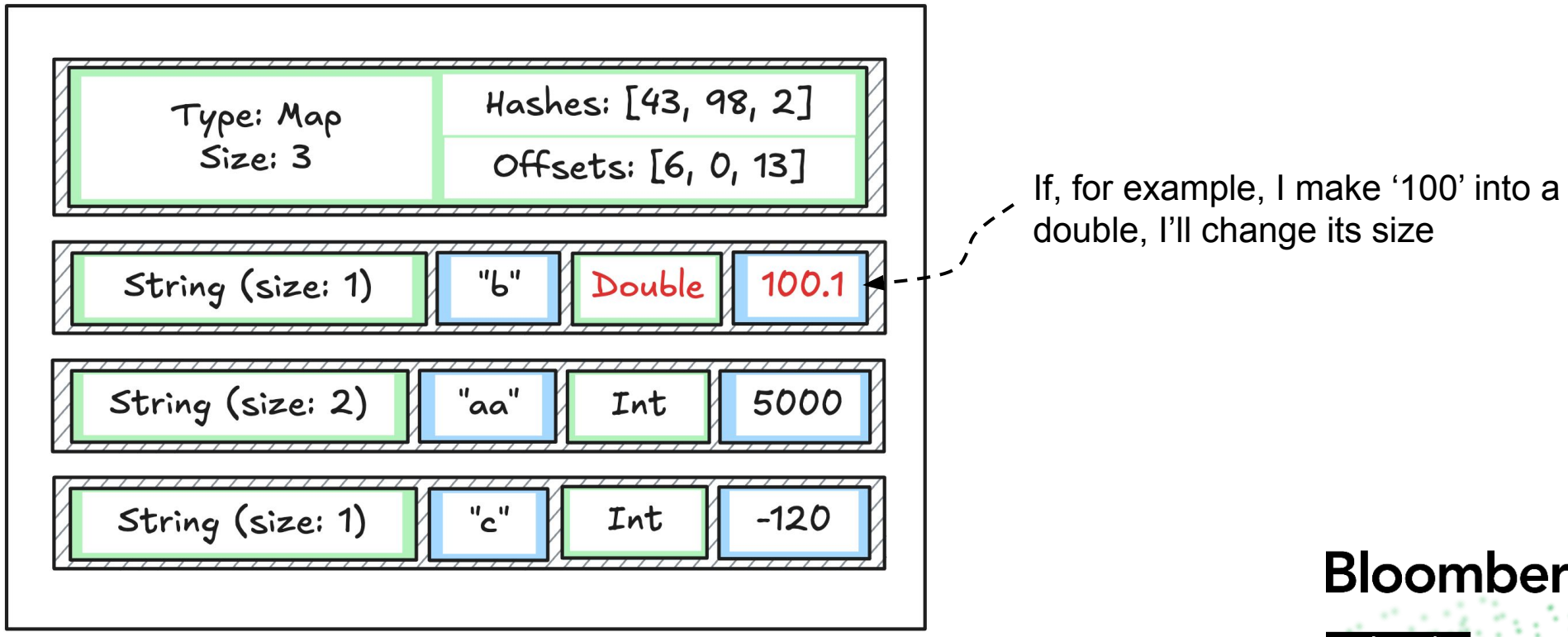


Bloomberg

Engineering

Making our Format Modifiable

- In our baseline format, we performed modifications by just moving everything around
- Now that we've introduced offsets, this is problematic
- Why?

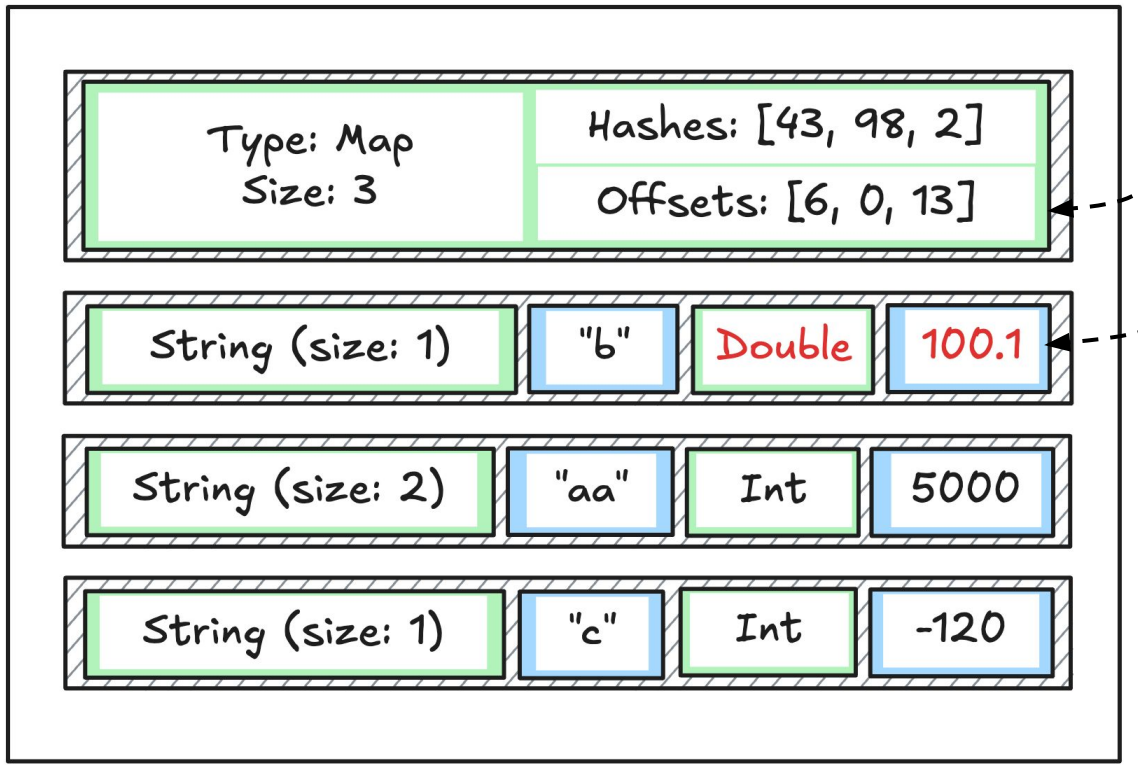


Bloomberg

Engineering

Making our Format Modifiable

- In our baseline format, we performed modifications by just moving everything around
- Now that we've introduced offsets, this is problematic
- Why?

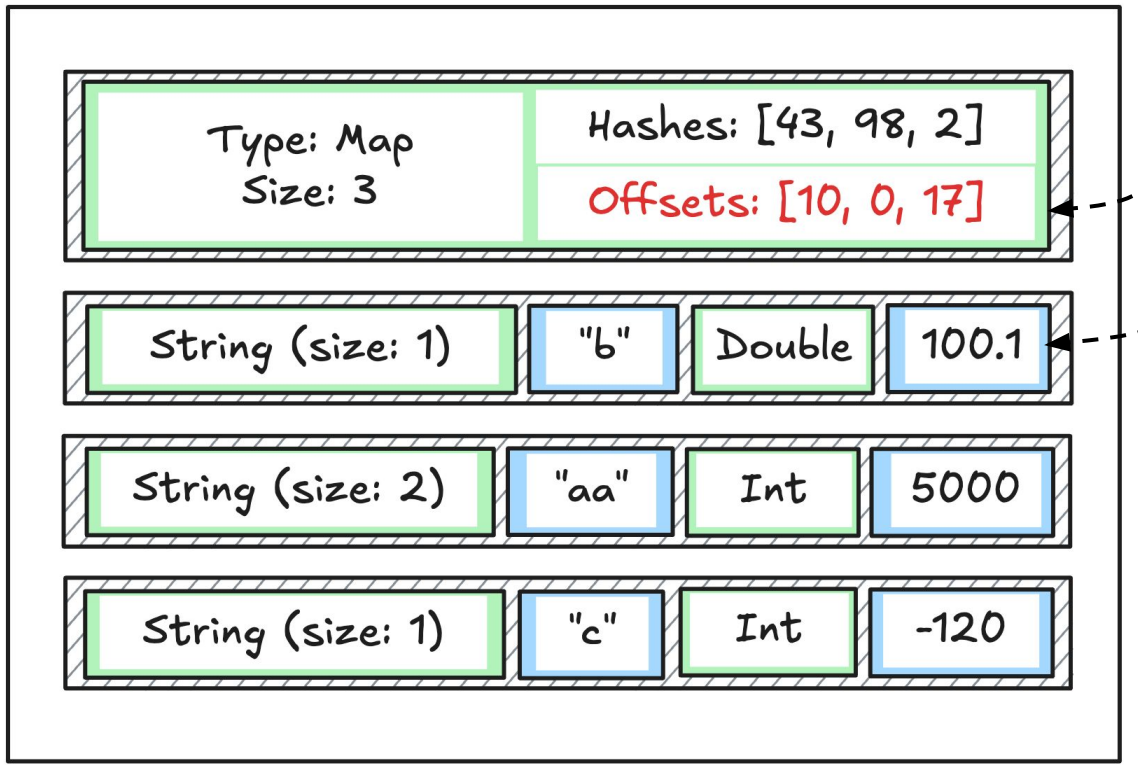


Now offsets of later elements need to be updated

If, for example, I make '100' into a double, I'll change its size

Making our Format Modifiable

- In our baseline format, we performed modifications by just moving everything around
- Now that we've introduced offsets, this is problematic
- Why?



Now offsets of later elements need to be updated

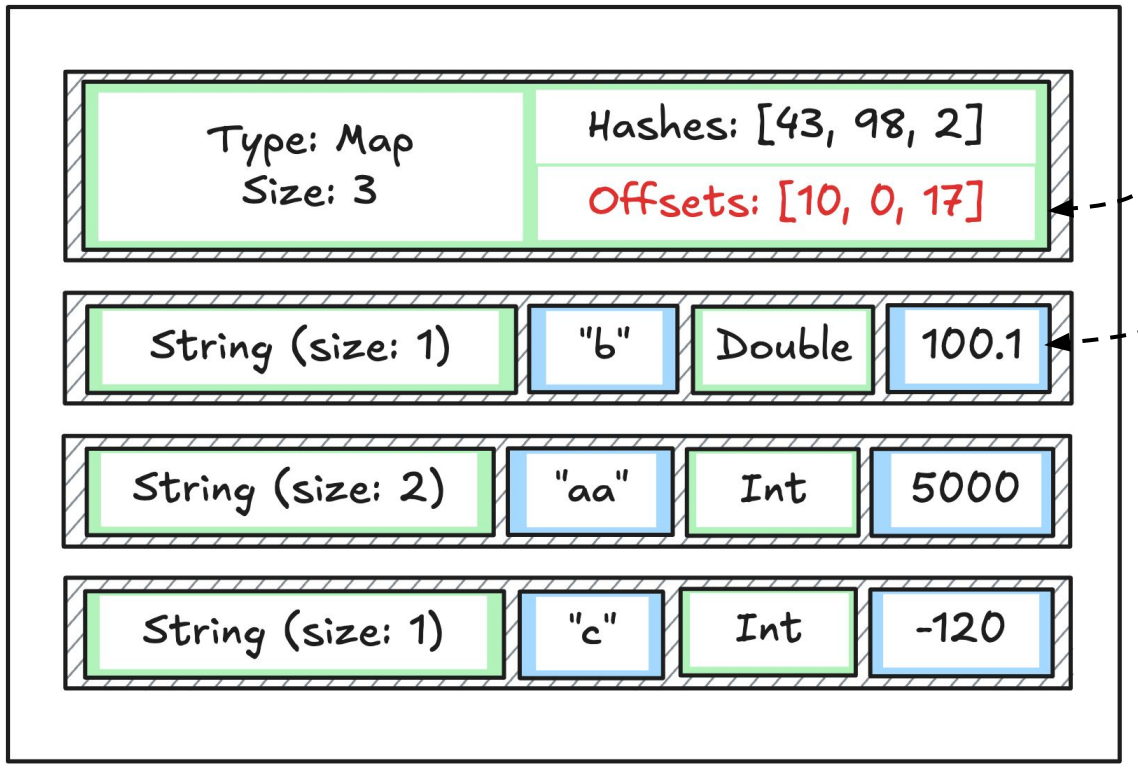
If, for example, I make '100' into a double, I'll change its size

Bloomberg

Engineering

Making our Format Modifiable

- Is this a big deal?
- On the face of it, no
- But the real problem comes when this map is nested within others



Now offsets of later elements need to be updated

If, for example, I make '100' into a double, I'll change its size

Bloomberg

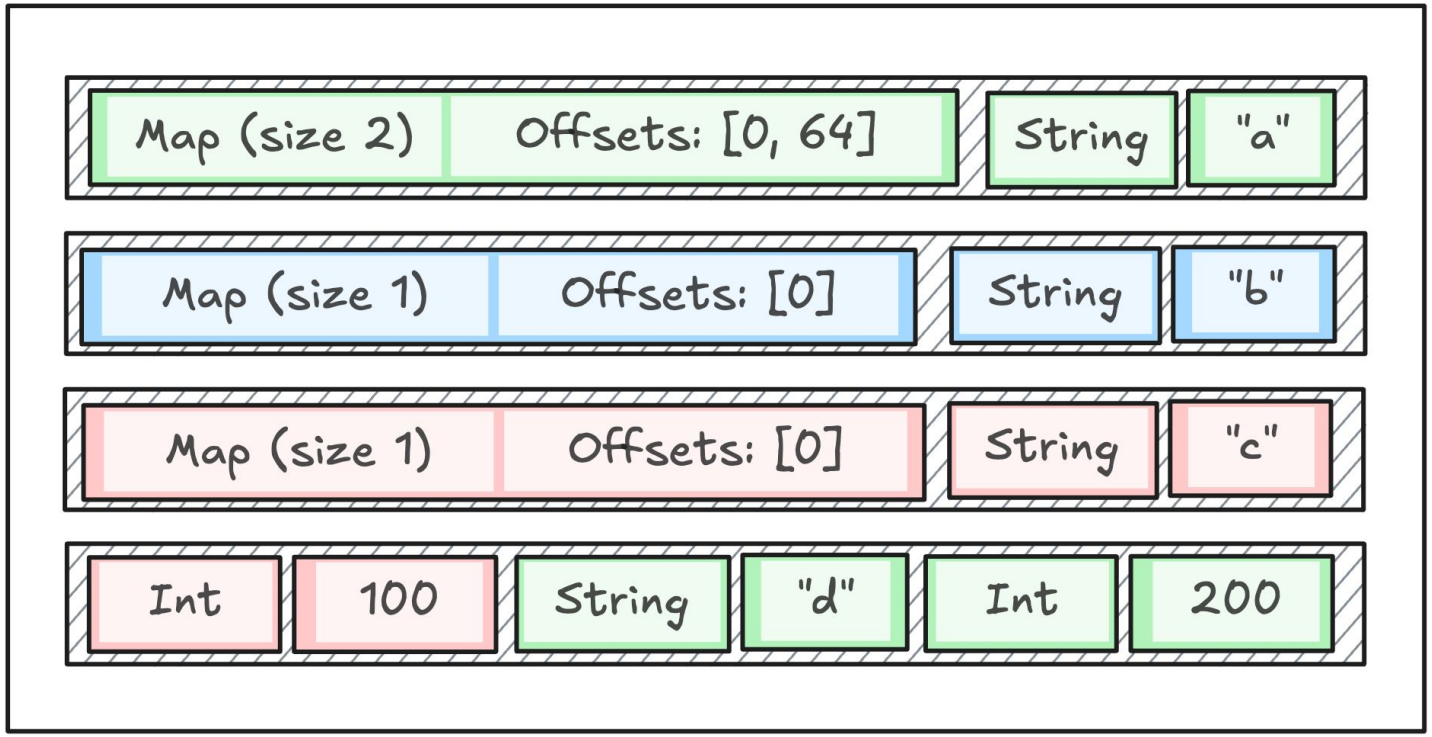
Engineering

The Problem of Nested Maps / Arrays

- If this map is nested in others, then moving elements within it will also move elements that come after this in *containing* maps

- To understand why this is a problem, consider the following document:

```
{
  "a":
  {
    "b":
    {
      "c": 100
    }
  }
  "d": 200
}
```

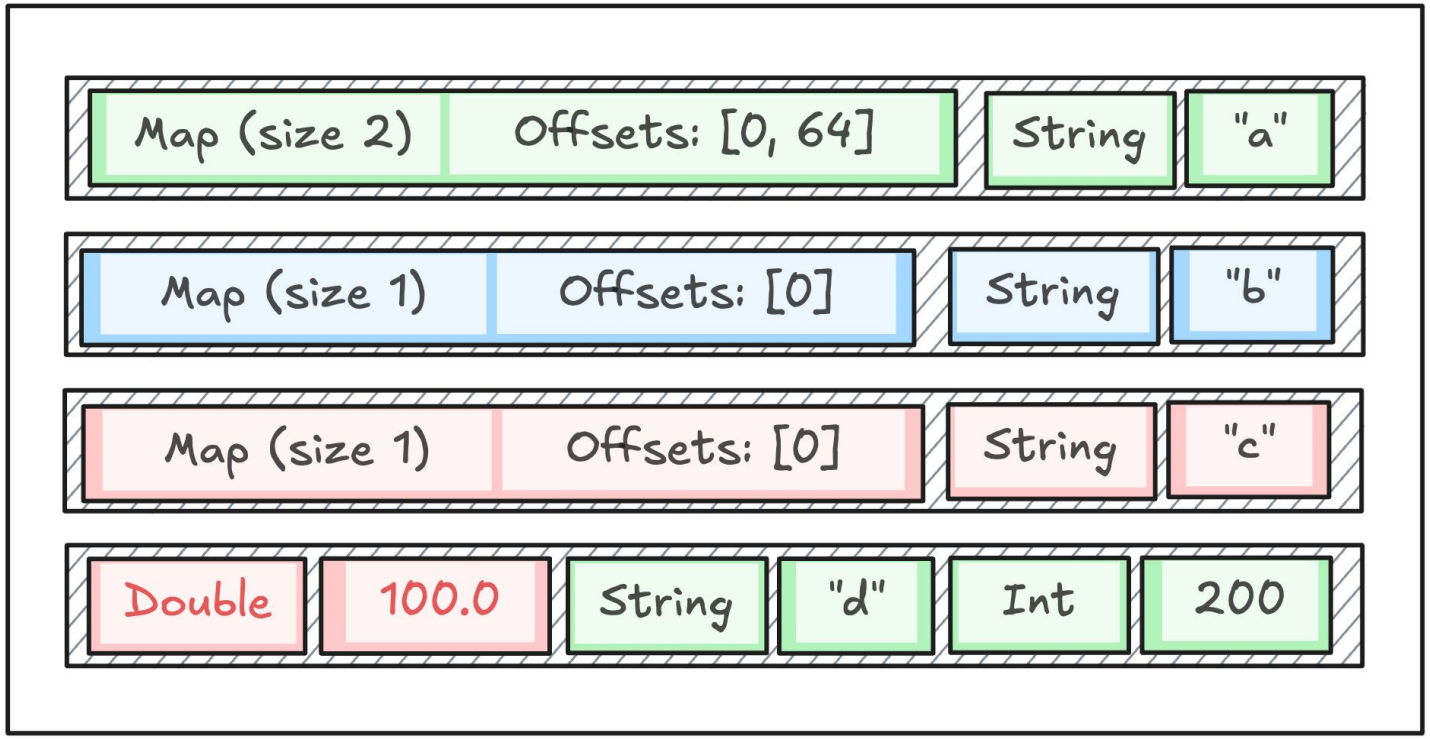


The Problem of Nested Maps / Arrays

- If this map is nested in others, then moving elements within it will also move elements that come after this in *containing* maps

- To understand why this is a problem, consider the following document:

```
{
  "a":
  {
    "b":
    {
      "c": 100.0
    }
  }
  "d": 200
}
```

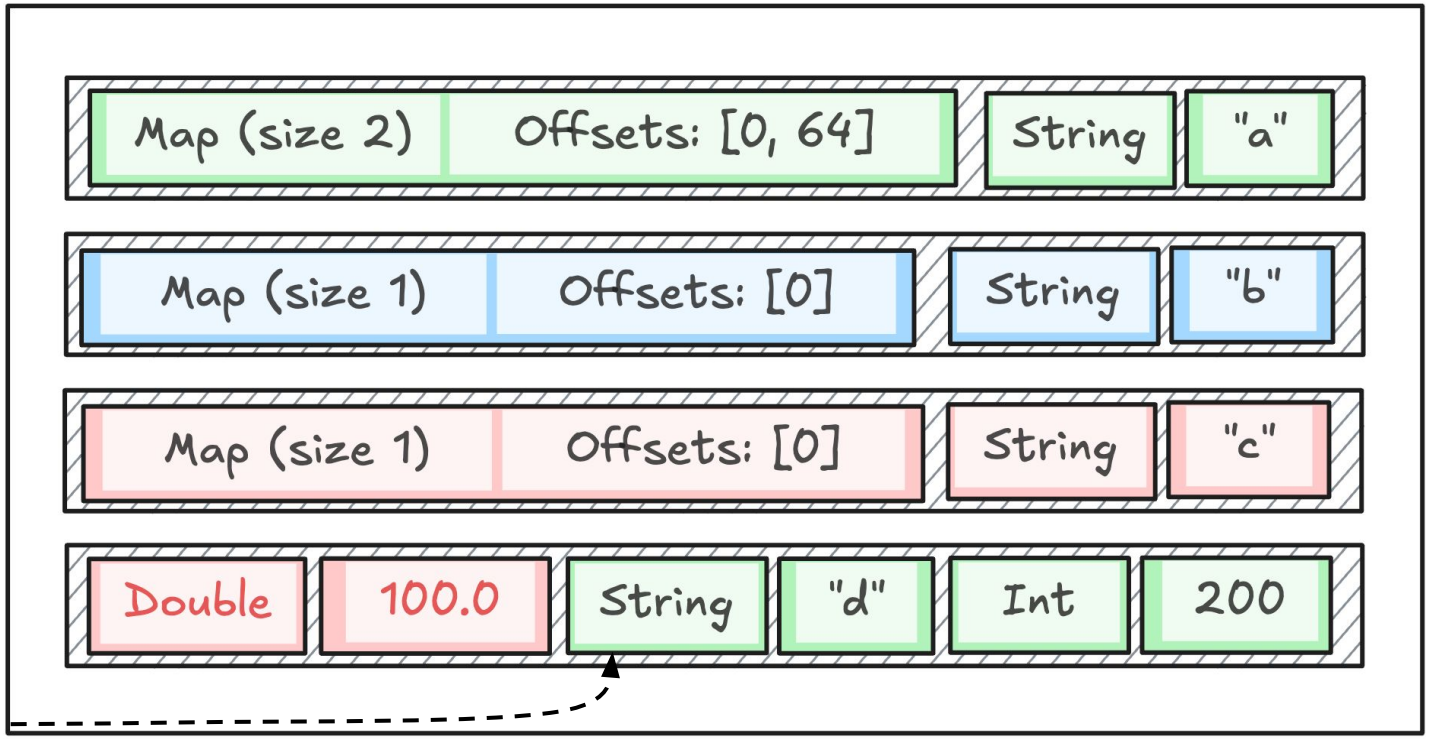


The Problem of Nested Maps / Arrays

- If this map is nested in others, then moving elements within it will also move elements that come after this in *containing* maps

- To understand why this is a problem, consider the following document:

```
{
  "a":
  {
    "b":
    {
      "c": 100.0
    }
  }
  "d": 200
}
```

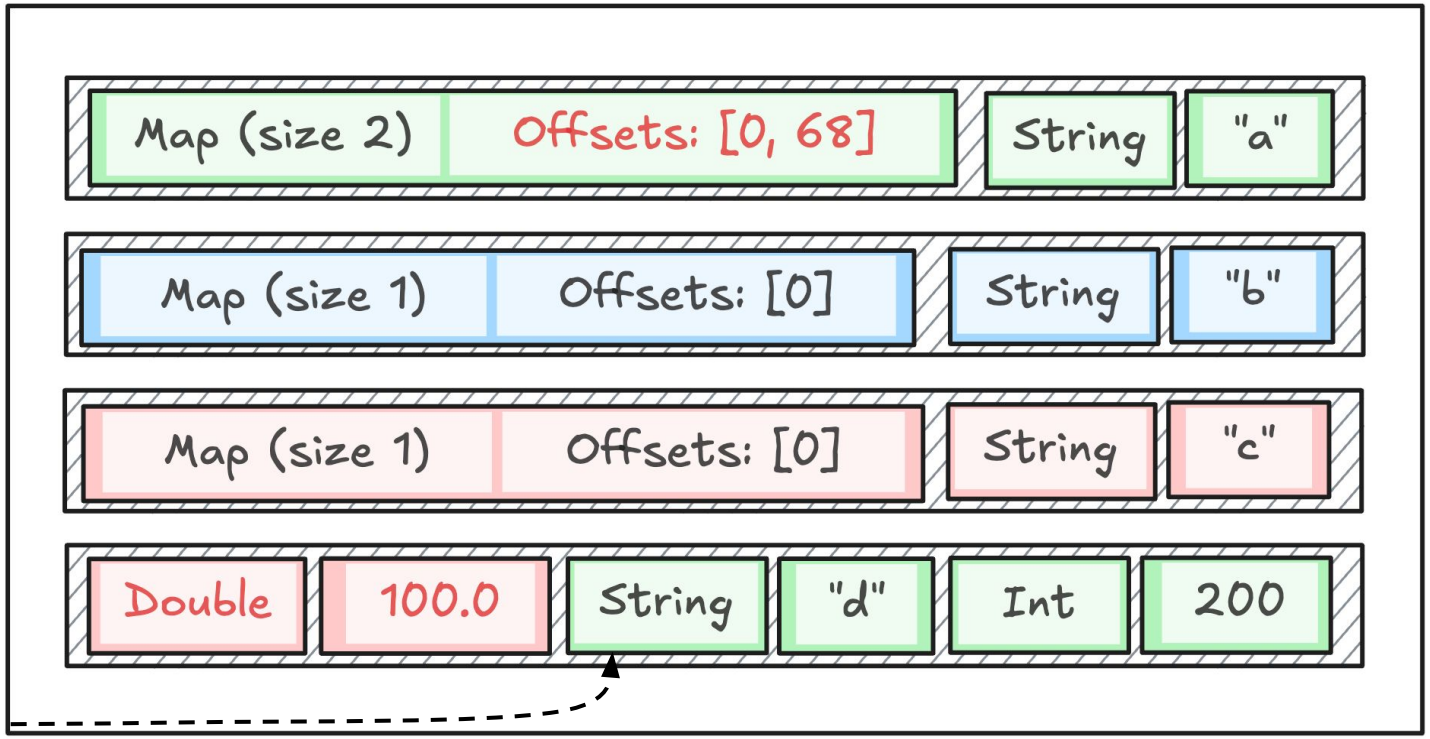


The Problem of Nested Maps / Arrays

- If this map is nested in others, then moving elements within it will also move elements that come after this in *containing* maps

- To understand why this is a problem, consider the following document:

```
{
  "a":
  {
    "b":
    {
      "c": 100.0
    }
  }
  "d": 200
}
```

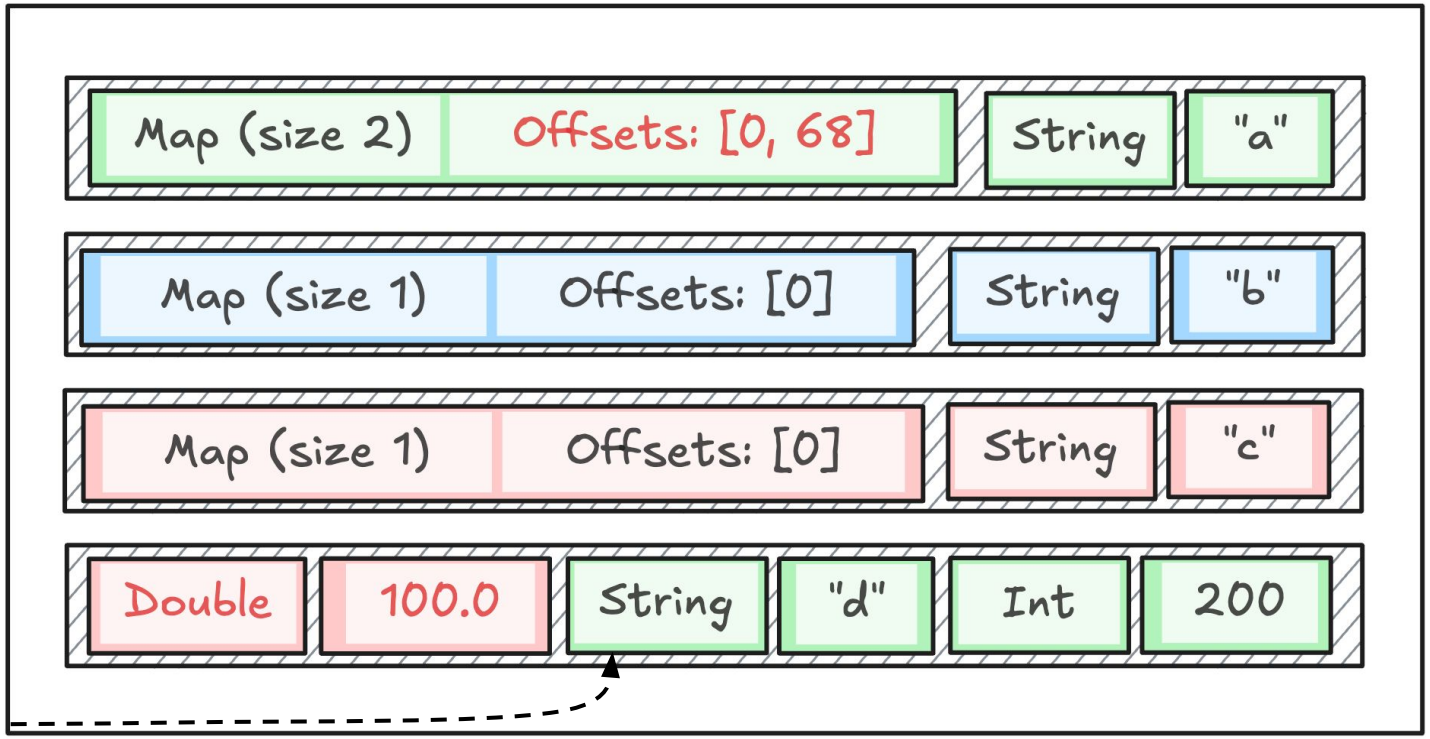


The offset of “d” now changes!

The Problem of Nested Maps / Arrays

- If I have a large amount of nesting, this will become quite expensive
- I need to track which parents I've traversed through, or store parent pointers, both of which add complexity

```
{
  "a":
  {
    "b":
    {
      "c": 100.0
    }
  }
  "d": 200
}
```



The offset of "d" now changes!

Faster Modifications

Idea 1: Move items “out-of-line” of their containing object

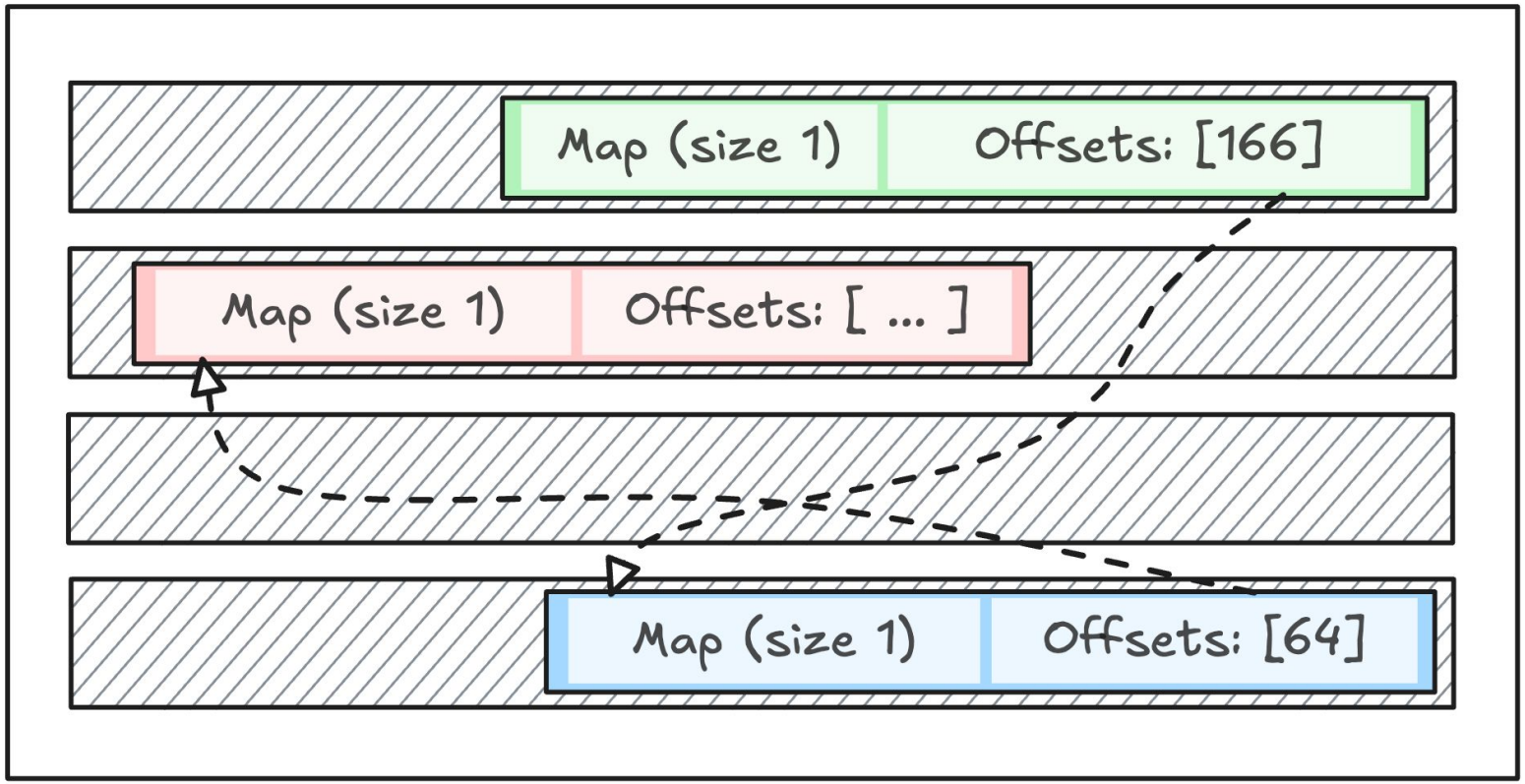
TechAtBloomberg.com

Bloomberg

Engineering

Move Items “Out-of-Line” of Containing Object

- That means that offsets can now point anywhere else in the buffer
- Our nested maps could now look like this:

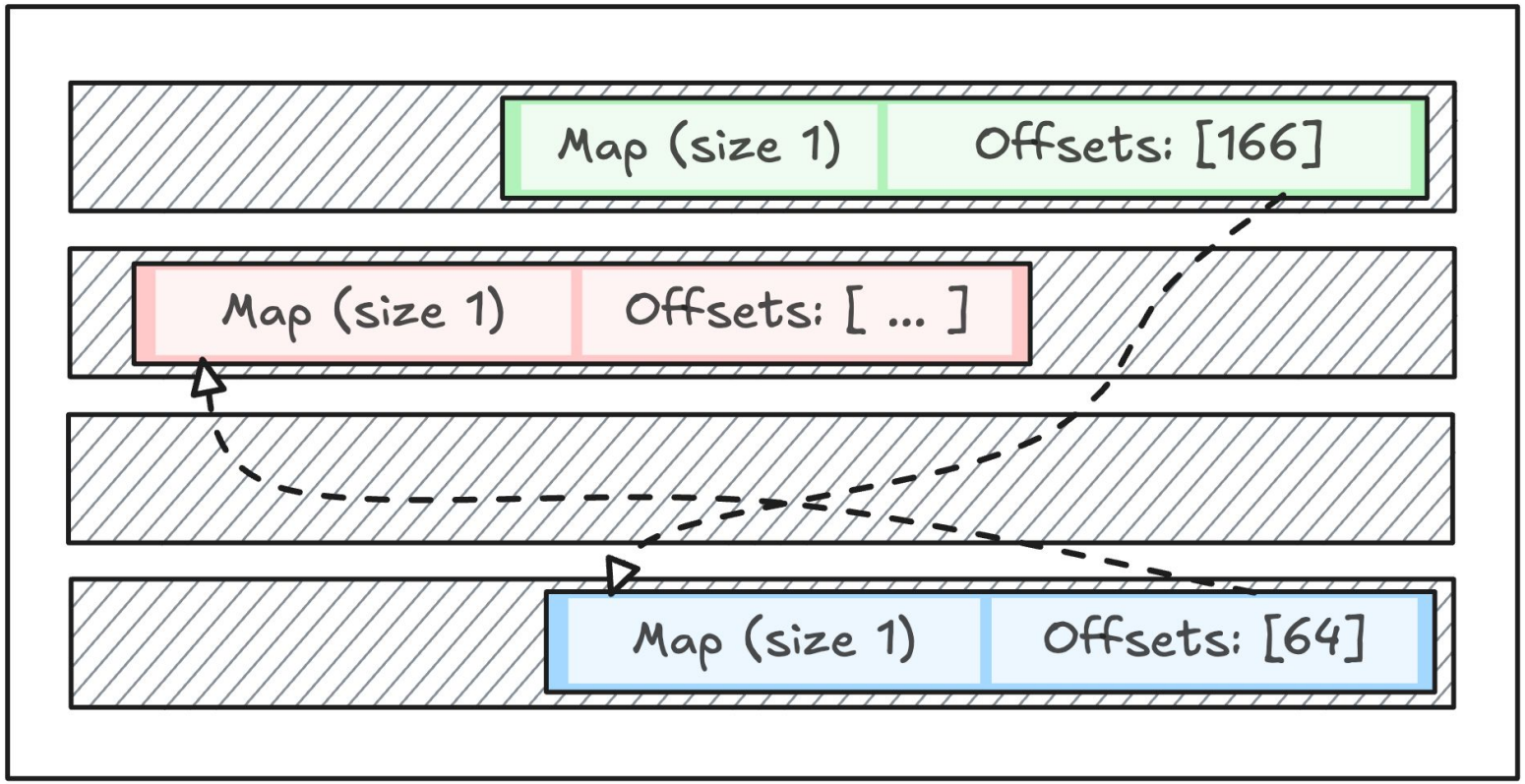


Bloomberg

Engineering

Move Items “Out-of-Line” of Containing Object

- Our offsets have essentially become small pointers
- The nesting is now expressed by maps “pointing to” other maps (and keys/values/etc.) via offsets

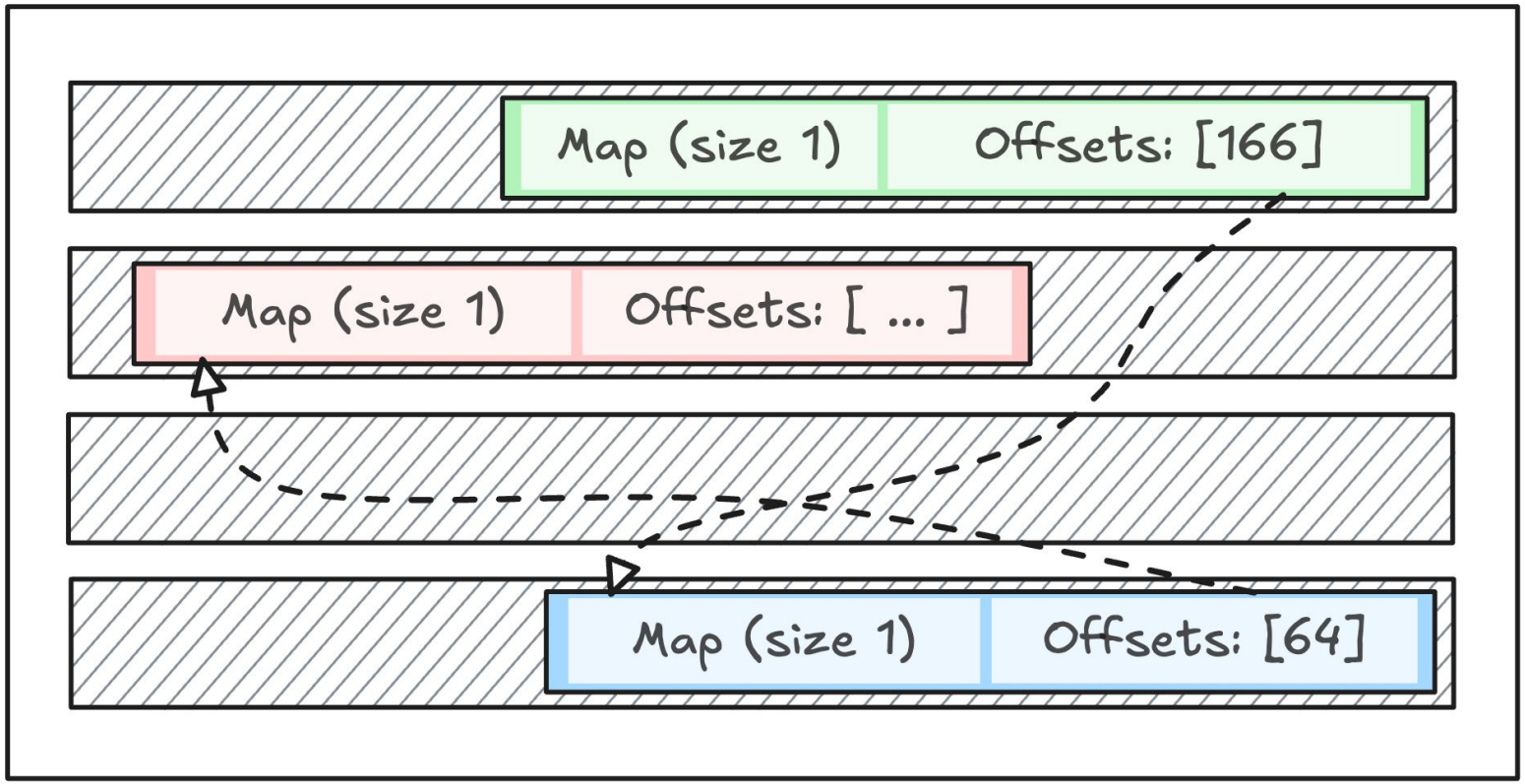


Bloomberg

Engineering

Allocating Space for Elements

- Now we have another problem:
- How do we allocate space for elements within our buffer?
  - … And how do we free up space?



Bloomberg

Engineering

Faster Modifications

Idea 1: Move items “out-of-line” of their containing object
Idea 2: Implement a bump allocator

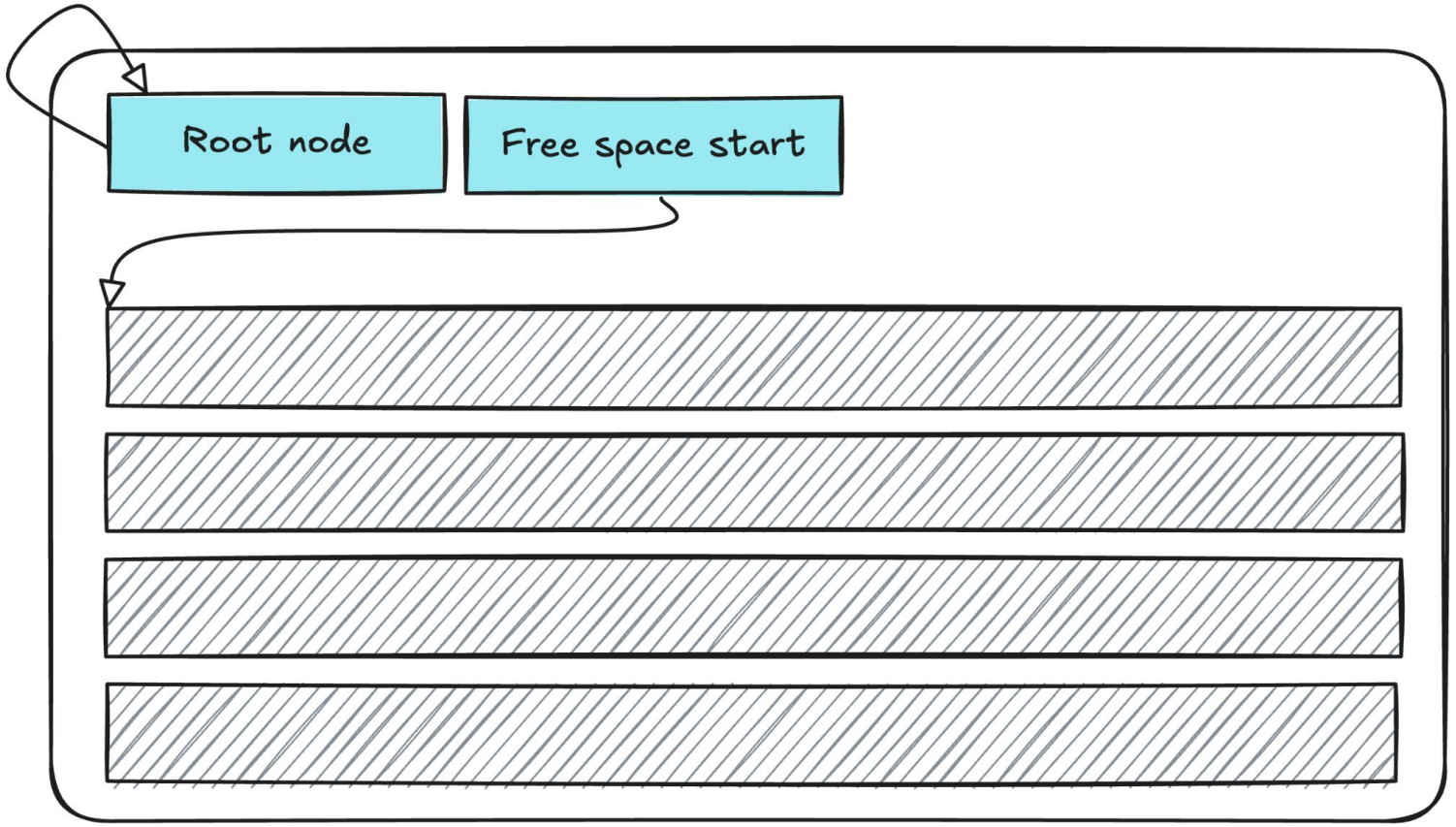
TechAtBloomberg.com

Bloomberg

Engineering

Bump Allocator

- The most simple option is a “bump allocator”
- In this, we keep an offset to the start of the free space within our buffer
- When we want to allocate something, we just bump this offset

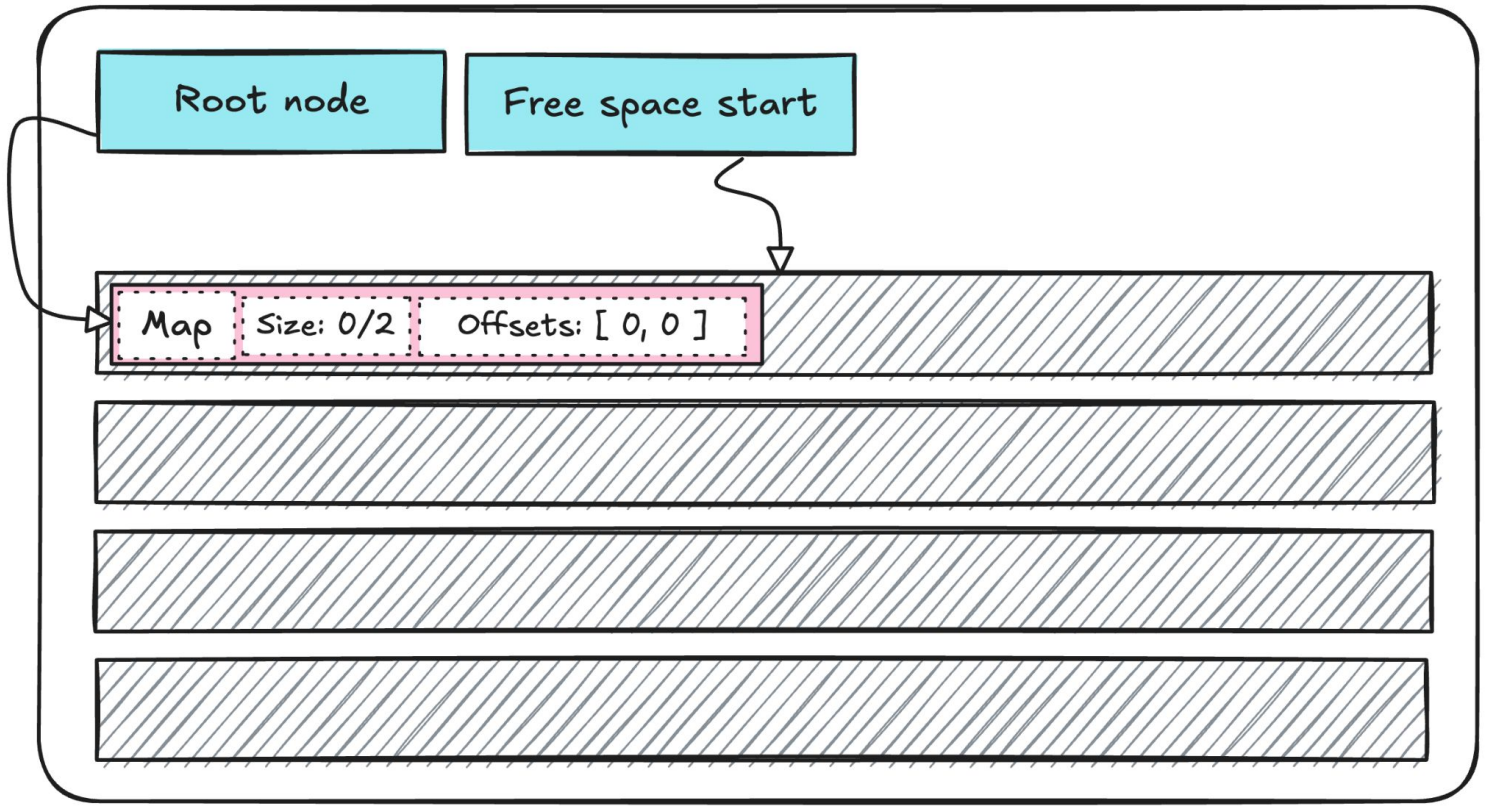


Bloomberg

Engineering

Bump Allocator

- First, we'll turn our root node into a map

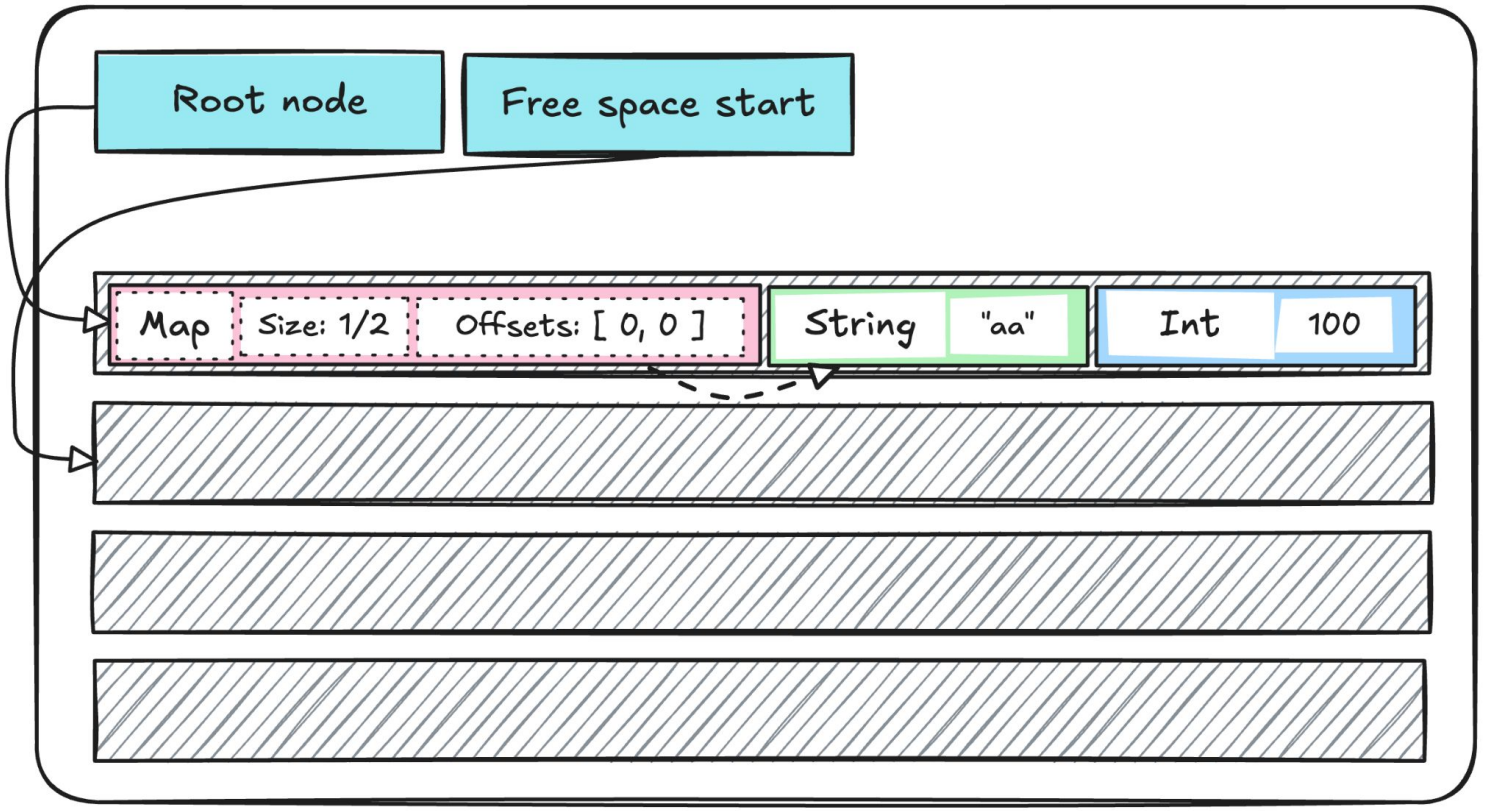


Bloomberg

Engineering

Bump Allocator

- Now, let’s add some keys and values to our map

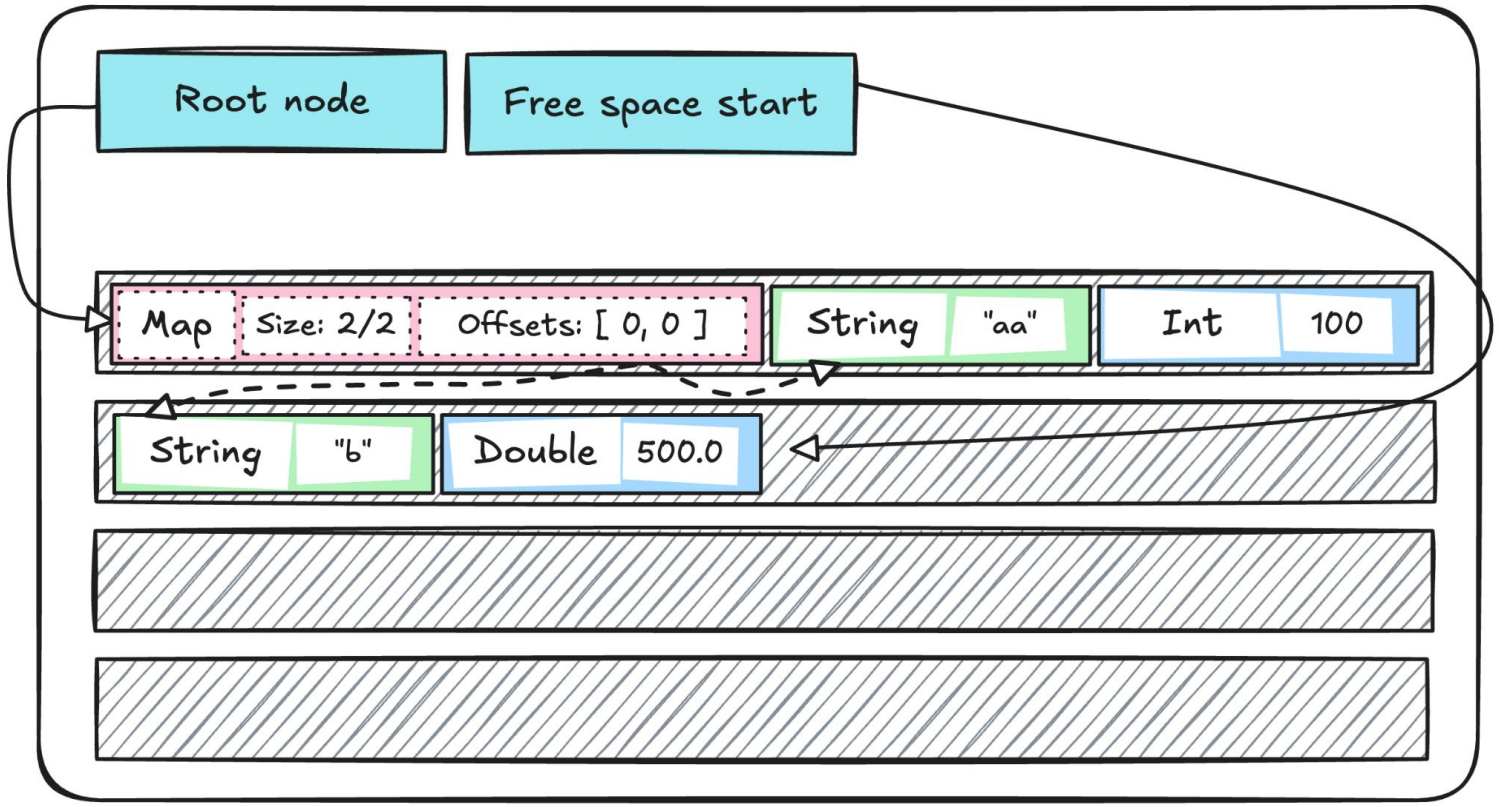


Bloomberg

Engineering

Bump Allocator

- Now, let’s add some keys and values to our map

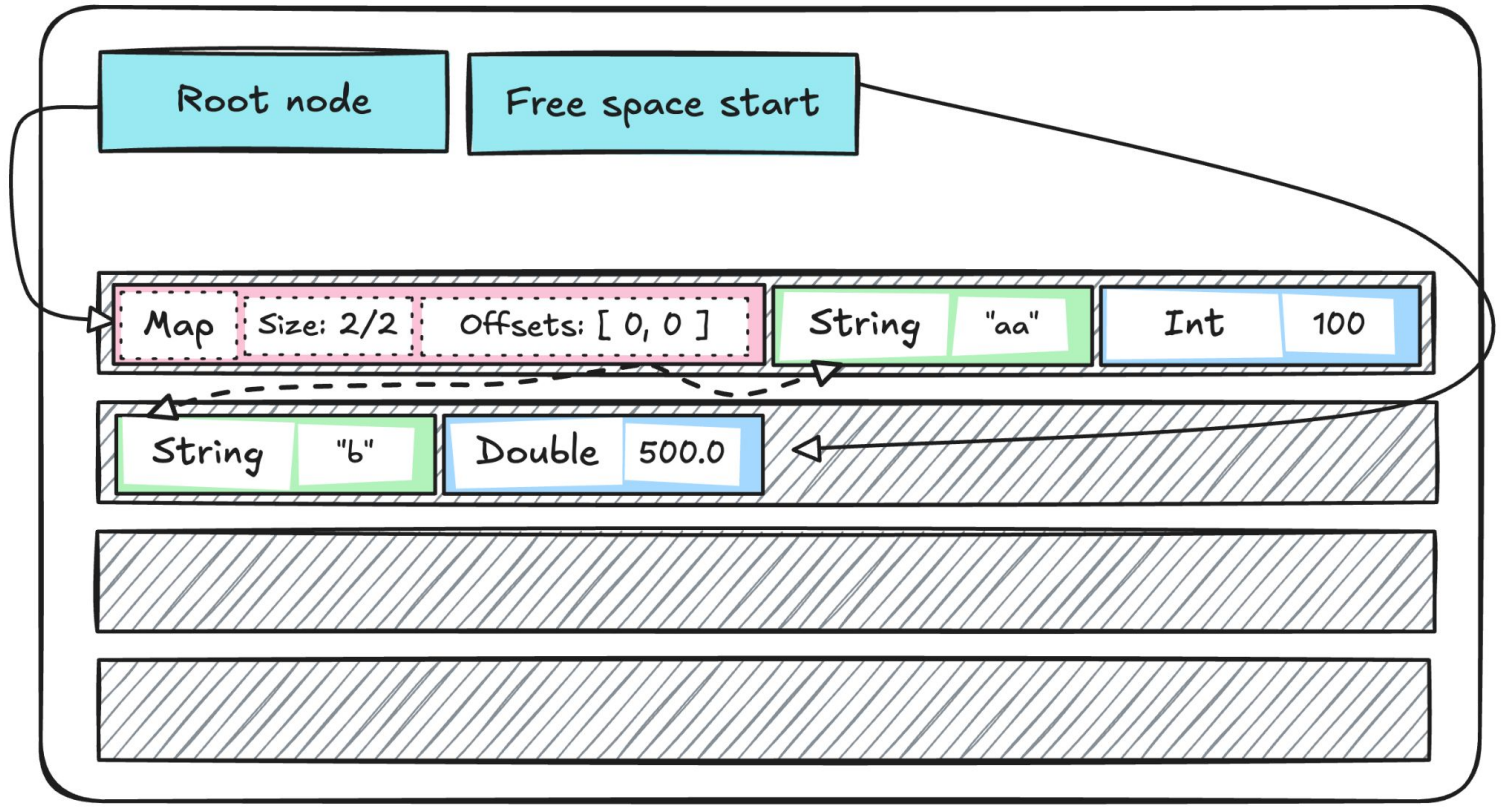


Bloomberg

Engineering

Bump Allocator

- Now I want to add another key and value, but I'm out of space in my map
- So I need to allocate more space for a larger map

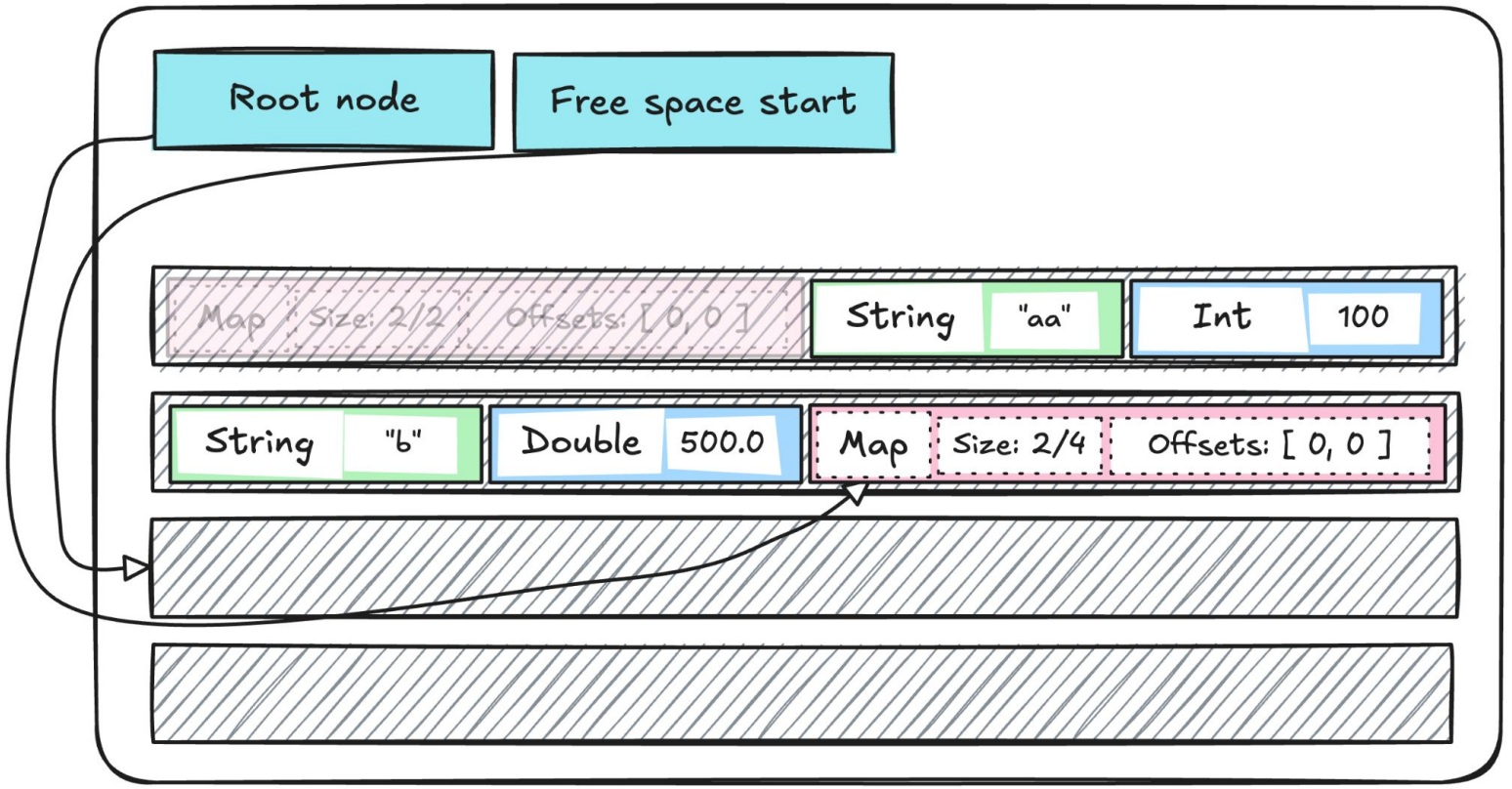


Bloomberg

Engineering

Bump Allocator

- Now I want to add another key and value, but I'm out of space in my map
- So I need to allocate more space for a larger map

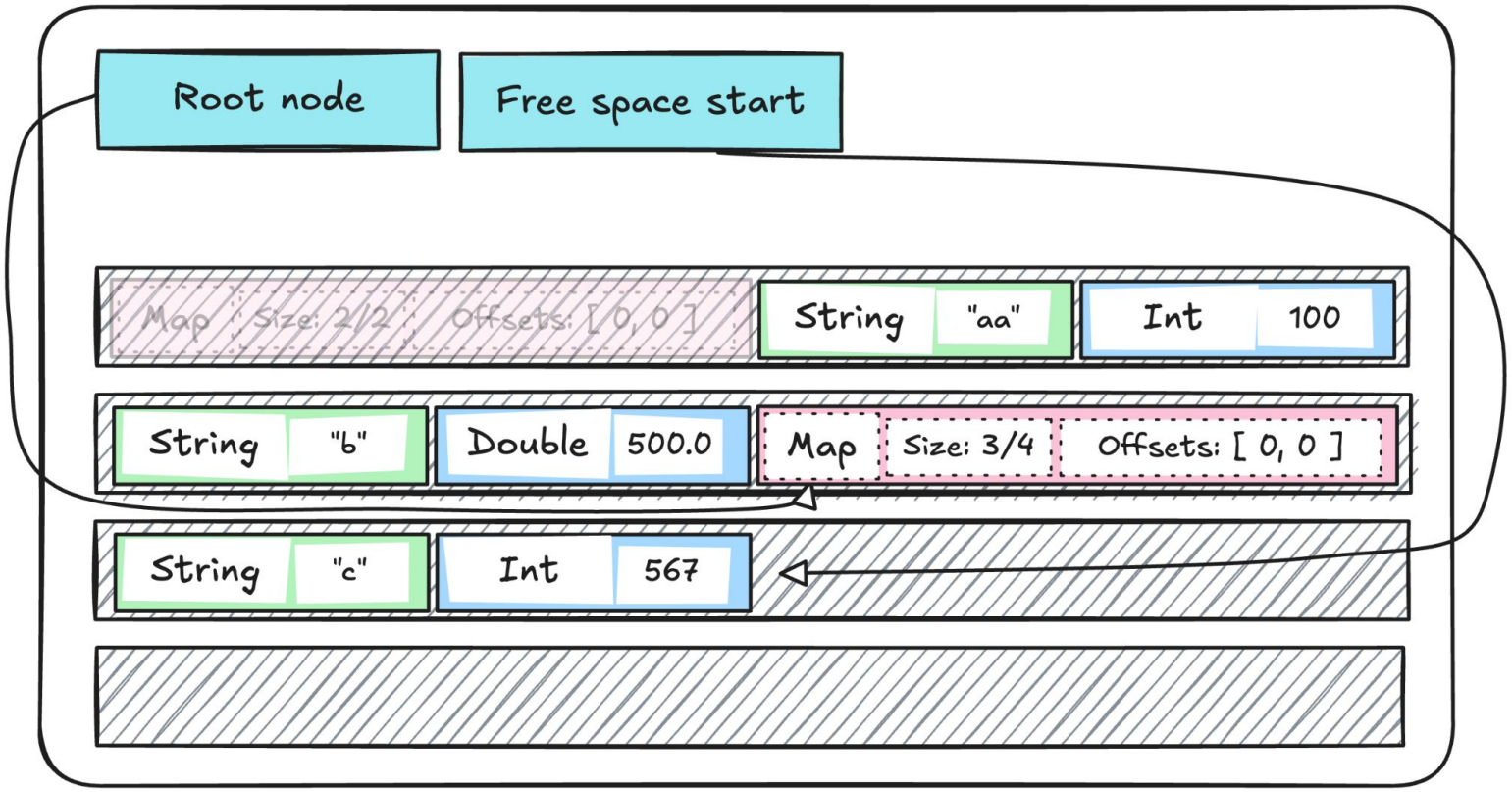


Bloomberg

Engineering

Bump Allocator

- Now I want to add another key and value, but I'm out of space in my map
- So I need to allocate more space for a larger map
- And now I can add my value

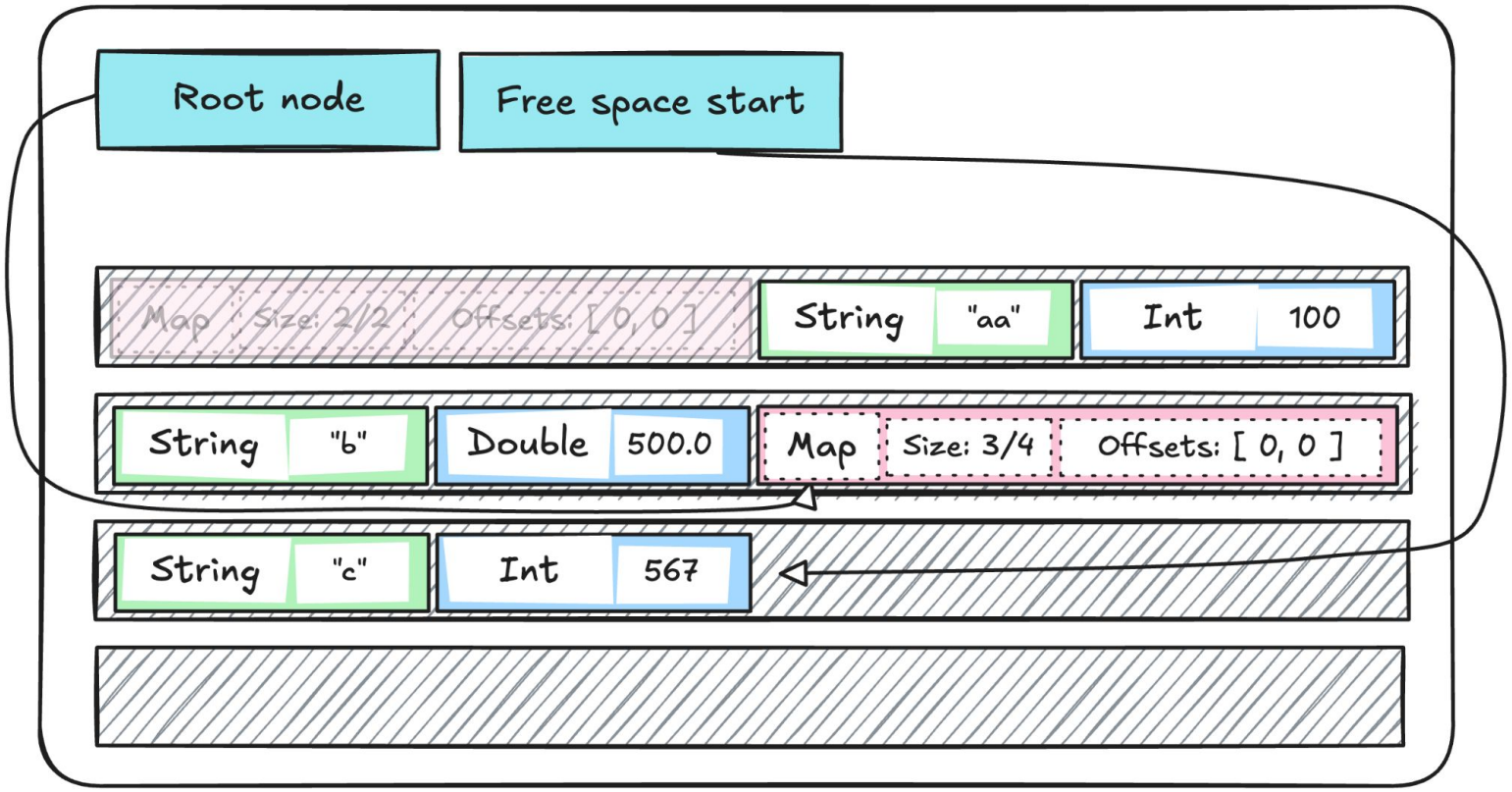


Bloomberg

Engineering

Bump Allocator

- Clearly, this strategy wastes a lot of space
- I will need to compact the whole buffer once I run out of space



Bloomberg

Engineering

How can I Improve on This?

- We could keep track of free space by maintaining a free-list through the buffer
- We would then try to satisfy allocations from the free-list before attempting to resize or compact

Faster Modifications

Idea 1: Move items “out-of-line” of their containing object

~~Idea 2: Implement a bump allocator~~

Idea 3: Implement a free-list allocator

Free List Allocator

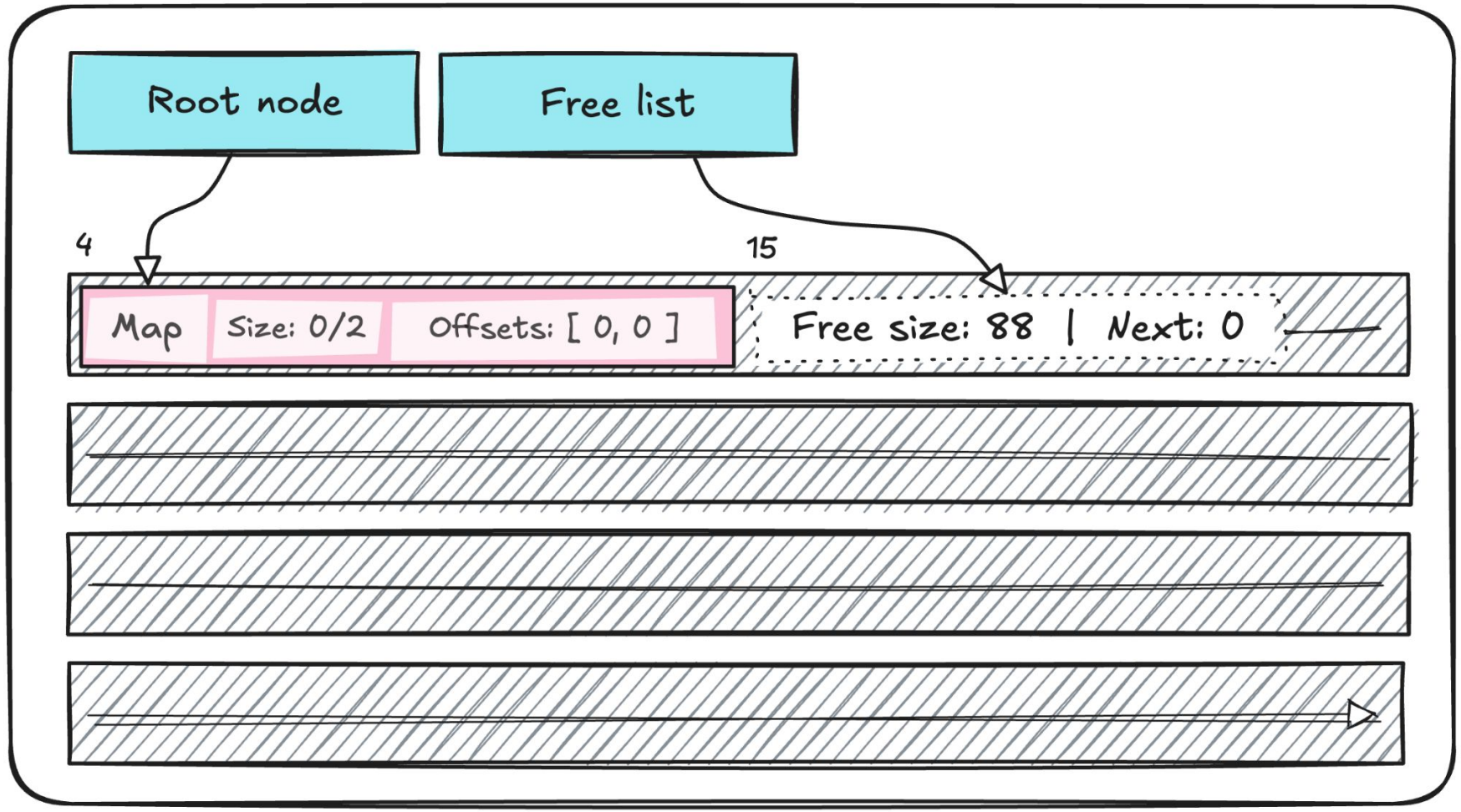
- This would look like this:

Bloomberg

Engineering

Free List Allocator

- This would look like this:

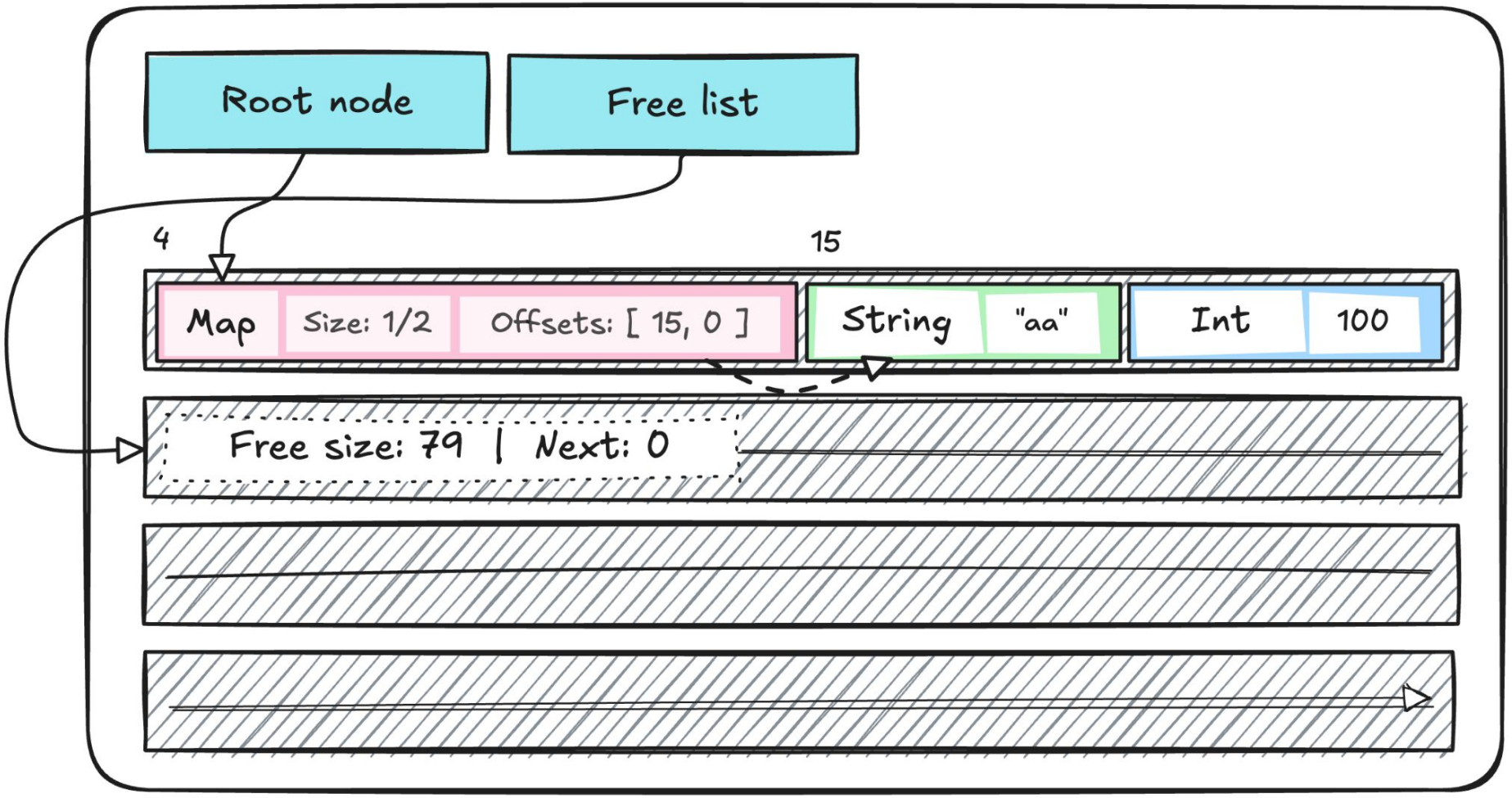


Bloomberg

Engineering

Free List Allocator

- This would look like this:

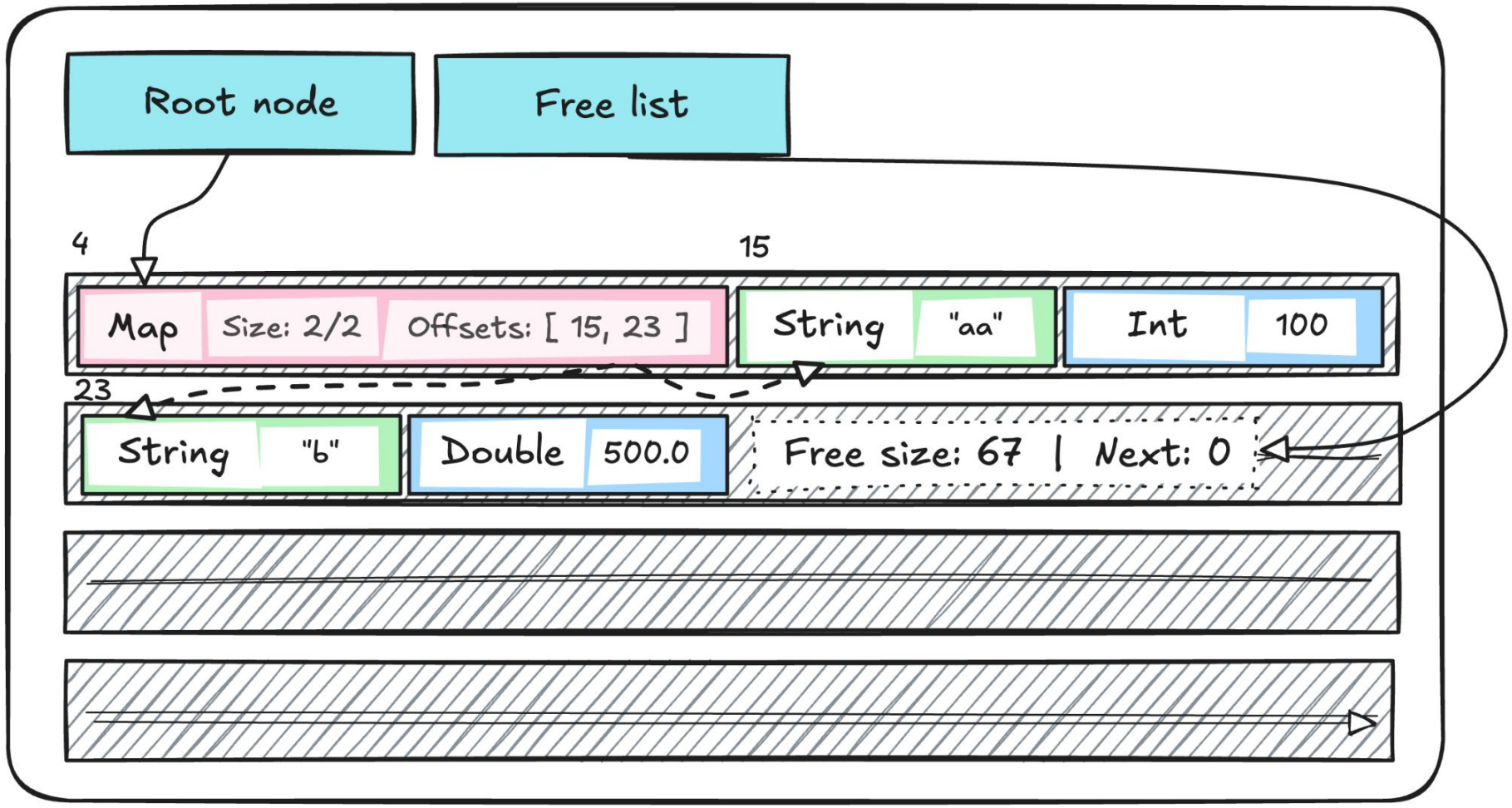


Bloomberg

Engineering

Free List Allocator

- This would look like this:

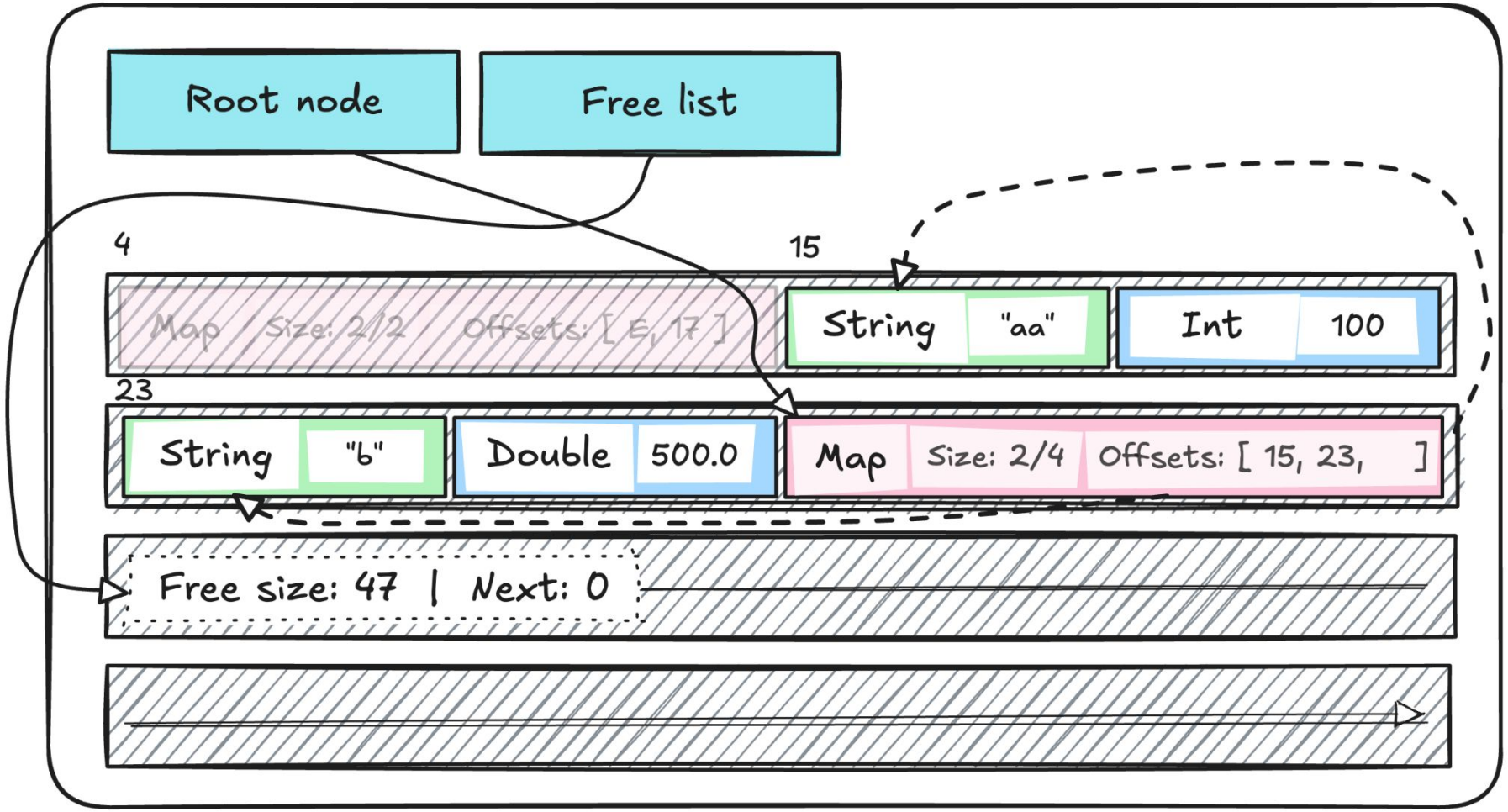


Bloomberg

Engineering

Free List Allocator

- This would look like this:

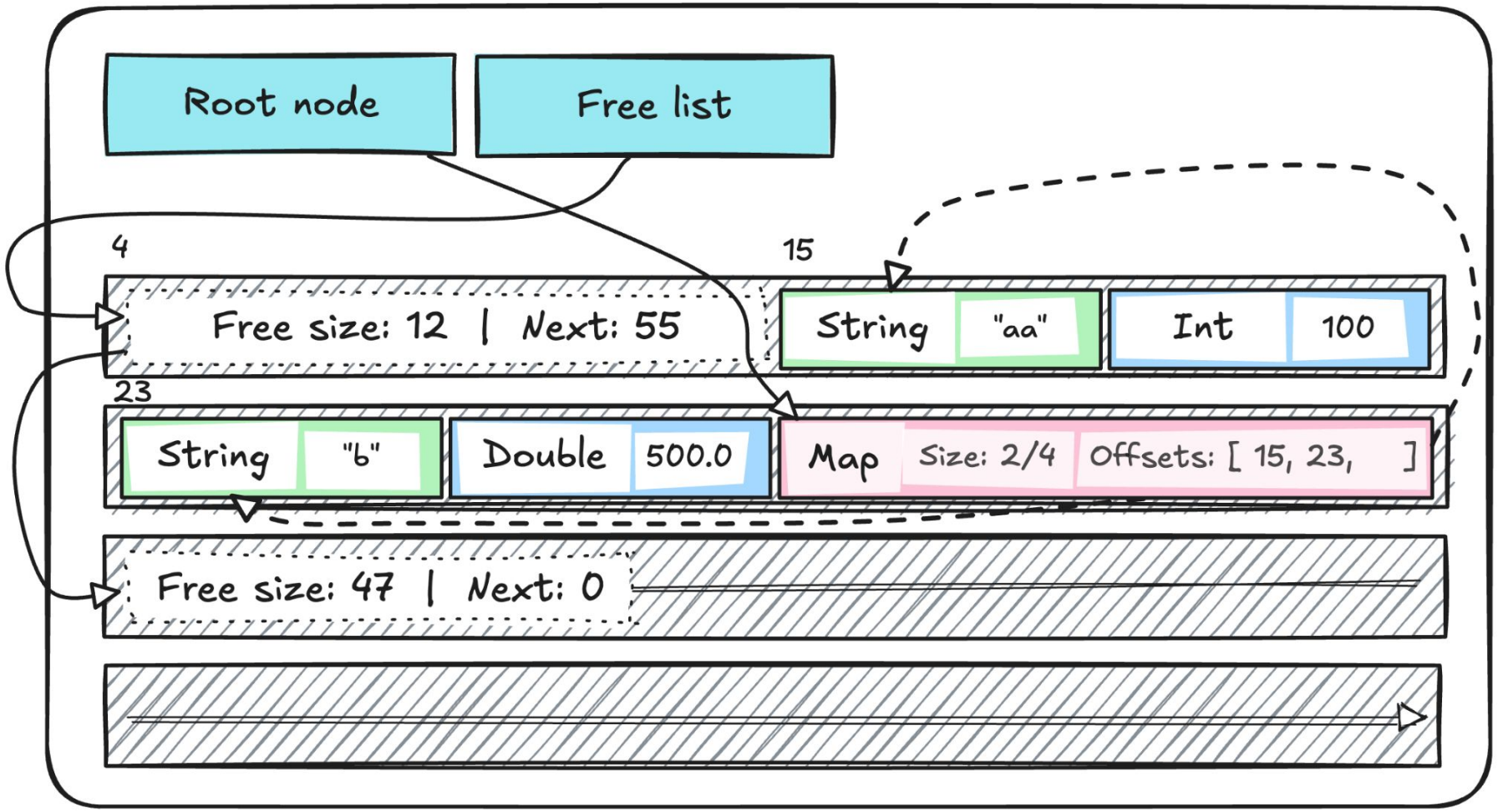


Bloomberg

Engineering

Free List Allocator

- This would look like this:

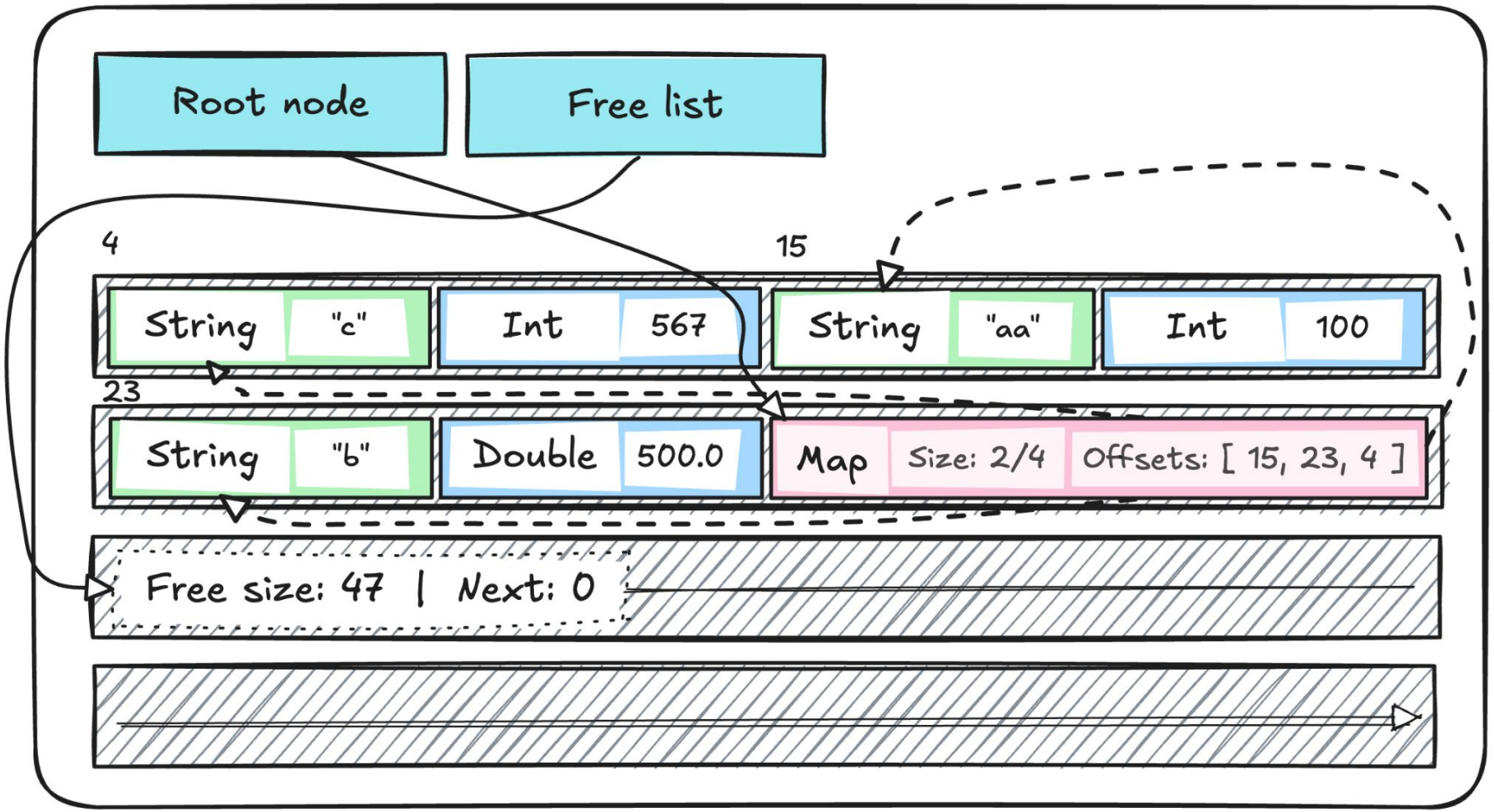


Bloomberg

Engineering

Free List Allocator

- This would look like this:



Bloomberg

Engineering

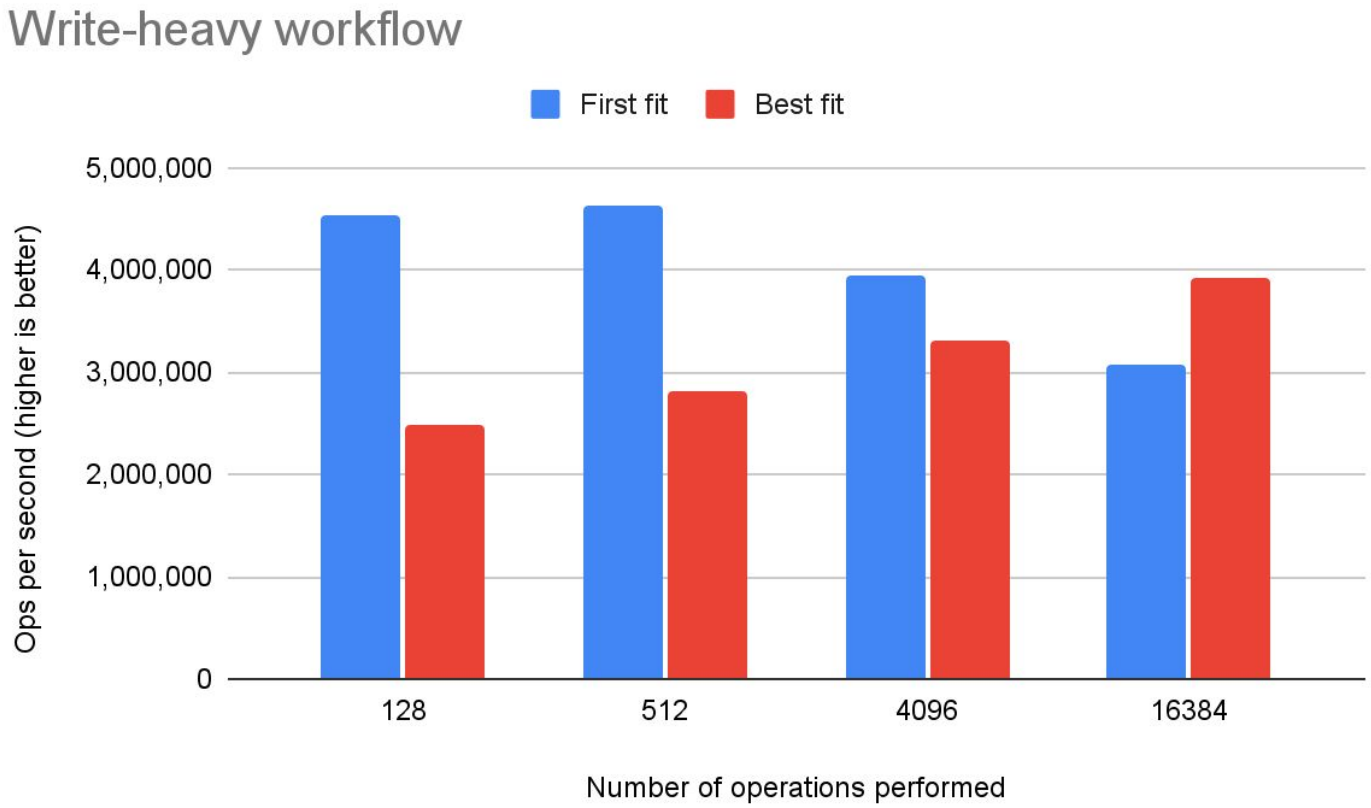
Free List Allocator

- We can satisfy new allocations using a few schemes:
  - First fit – return the first block I find that fits our allocation
  - Best fit – return the smallest block I can find that fits our allocation

- I benchmarked both schemes, in terms of both wasted space and performance

Free List Allocator

- First-fit performs better initially, but performance degrades as more operations are performed on the buffer
- After many operations, best-fit wins

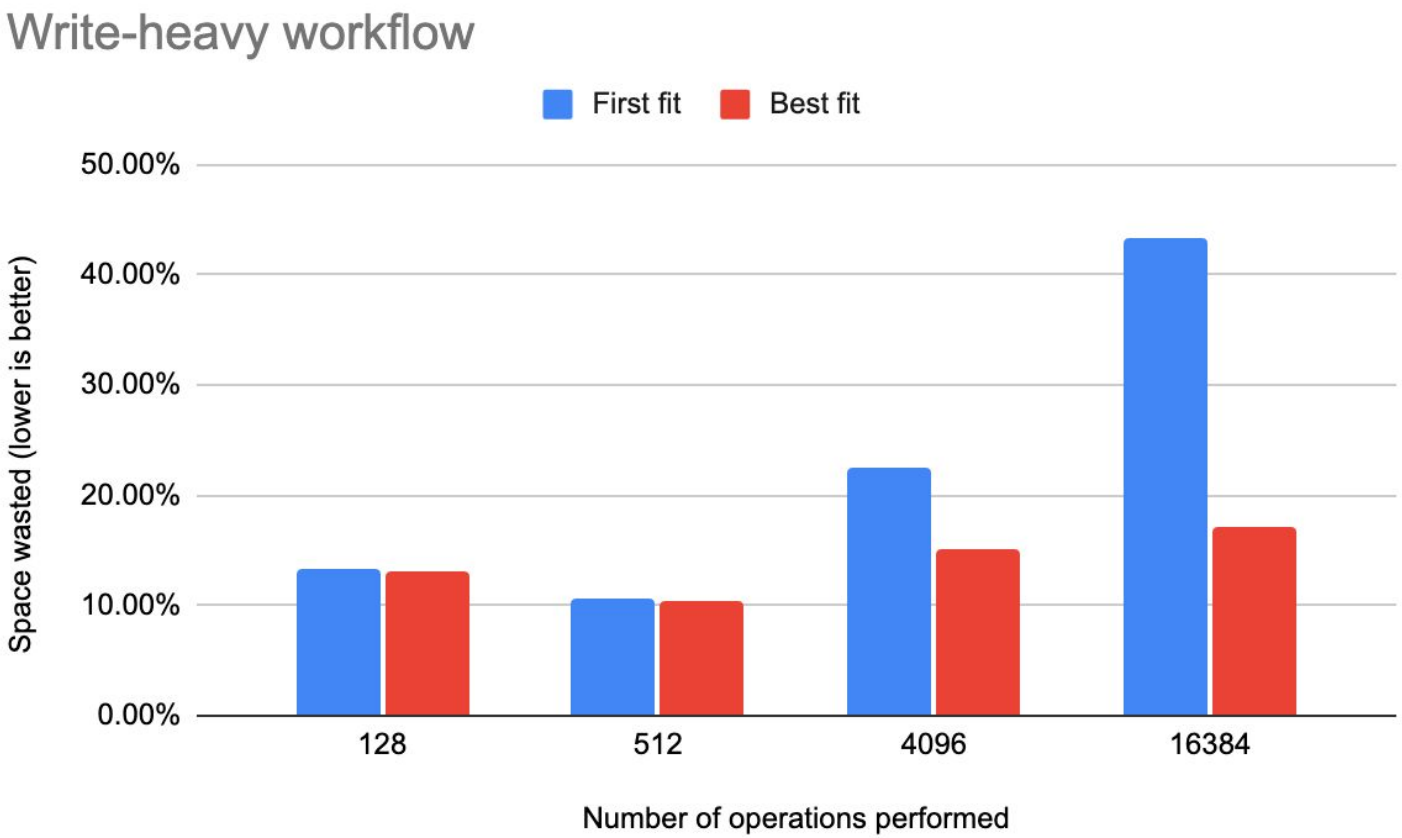


Bloomberg

Engineering

Free List Allocator

- With first-fit, space wastage increases considerably the more operations are performed:



Bloomberg

Engineering

Free List Allocator – Profiling

- Profiling allocation shows a lot of time is spent reading the size of each free-list chunk
- The free-list chunks are scattered throughout the buffer – so they might not be in cache

```
                        size > TaggedUtil::freeSpaceSize(&d_buffer[freeListOff])) {
                    freeListPtr = TaggedUtil::freeSpacePtr(&d_buffer[freeListOff]);
                    freeListOff = *freeListPtr;
  2.70        movzwl 0x3(%rcx),%ecx
                  while (freeListOff != 0 &&
  9.50        test   %rcx,%rcx
              → je     2438f8 <experimental::ser::adv::container::Container::allocate(int,
              _ZNSt6vectorIhSaIhEEixEm():
                      return *(this->_M_impl._M_start + __n);
  1.14        add    %r11,%rcx
              _ZN12experimental3ser6tagged10TaggedUtil13freeSpaceSizeEPKh():
                      return (data[2] << 8) | data[1];
 44.66        movzwl 0x1(%rcx),%esi
              _ZN12experimental3ser3adv9container9Container8allocateEiPPhS5_S5_():
  0.72        cmp    %r9,%rsi
 10.48        → jb     243828 <experimental::ser::adv::container::Container::allocate(int,
```

Faster Modifications

Idea 1: Move items “out-of-line” of their containing object
~~Idea 2: Implement a bump allocator~~
Idea 3: Implement a free-list allocator
Idea 4: Prefetch free-list entries while iterating

Free List Allocator – Prefetching

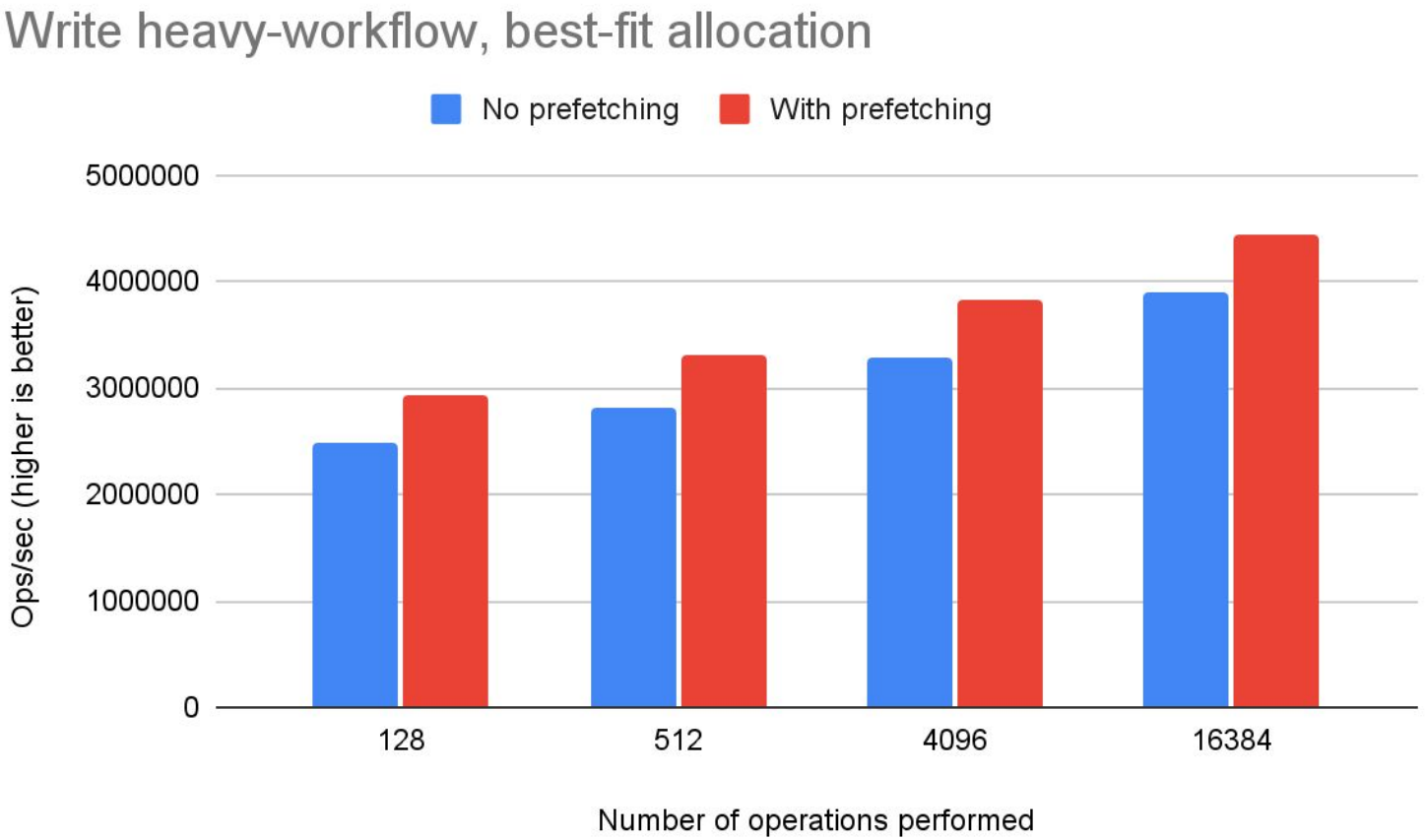
- We can issue a prefetch for the next item on the free list, before we know if `freeListOff != 0`

```
uint16_t *freeListPtr = &hdr->freeListHead;
uintptr_t freeListOff = *freeListPtr;

__builtin_prefetch(&d_buffer[freeListOff]);

while (freeListOff != 0 &&
       size > freeSpaceSize(&d_buffer[freeListOff])) {

    freeListPtr = freeSpacePtr(&d_buffer[freeListOff]);
    freeListOff = *freeListPtr;

    __builtin_prefetch(&d_buffer[freeListOff]);
}
```

If `freeListOff` is zero, we do a harmless prefetch of our buffer header, which is very likely already in cache anyway

Free List Allocator – Prefetching

- This gives us a ~15% performance boost with best-fit
- No improvement for first-fit



Bloomberg

Engineering

Free List Allocator – Prefetching

- A word of warning on prefetching:
  - After getting this result, I experimented with adding prefetching in other parts of the code
  - … But it uniformly made things worse!
  - So the hardware prefetcher is generally pretty good
  - Prefetching is only likely to be beneficial if you have genuinely unpredictable access patterns

Faster Modifications

Idea 1: Move items “out-of-line” of their containing object
~~Idea 2: Implement a bump allocator~~
Idea 3: Implement a free-list allocator
Idea 4: Prefetch free-list entries while iterating
I’m happy with modification performance now!

Final Benchmarks

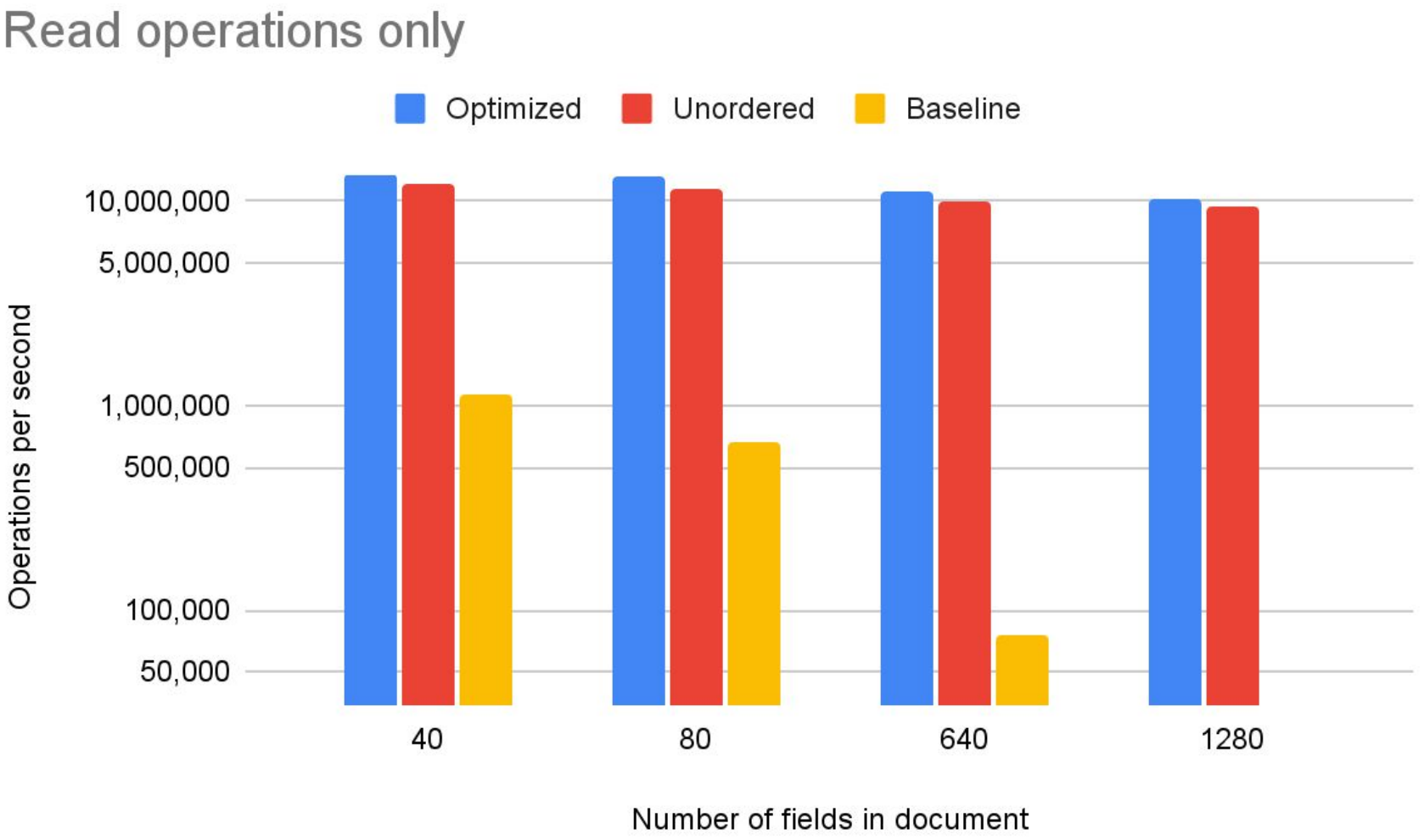
- For my final benchmarks I decided to compare the format against the baseline format, and against `unordered_map`

- In the `unordered_map` version:
  - Our `unordered_maps` have two levels of nesting
  - Our leaf values are variants of int, double, string, and vector of int
  - I used heterogeneous lookup to prevent excessive string allocations
  - I use the same fast hash function as my format

Bloomberg

Engineering

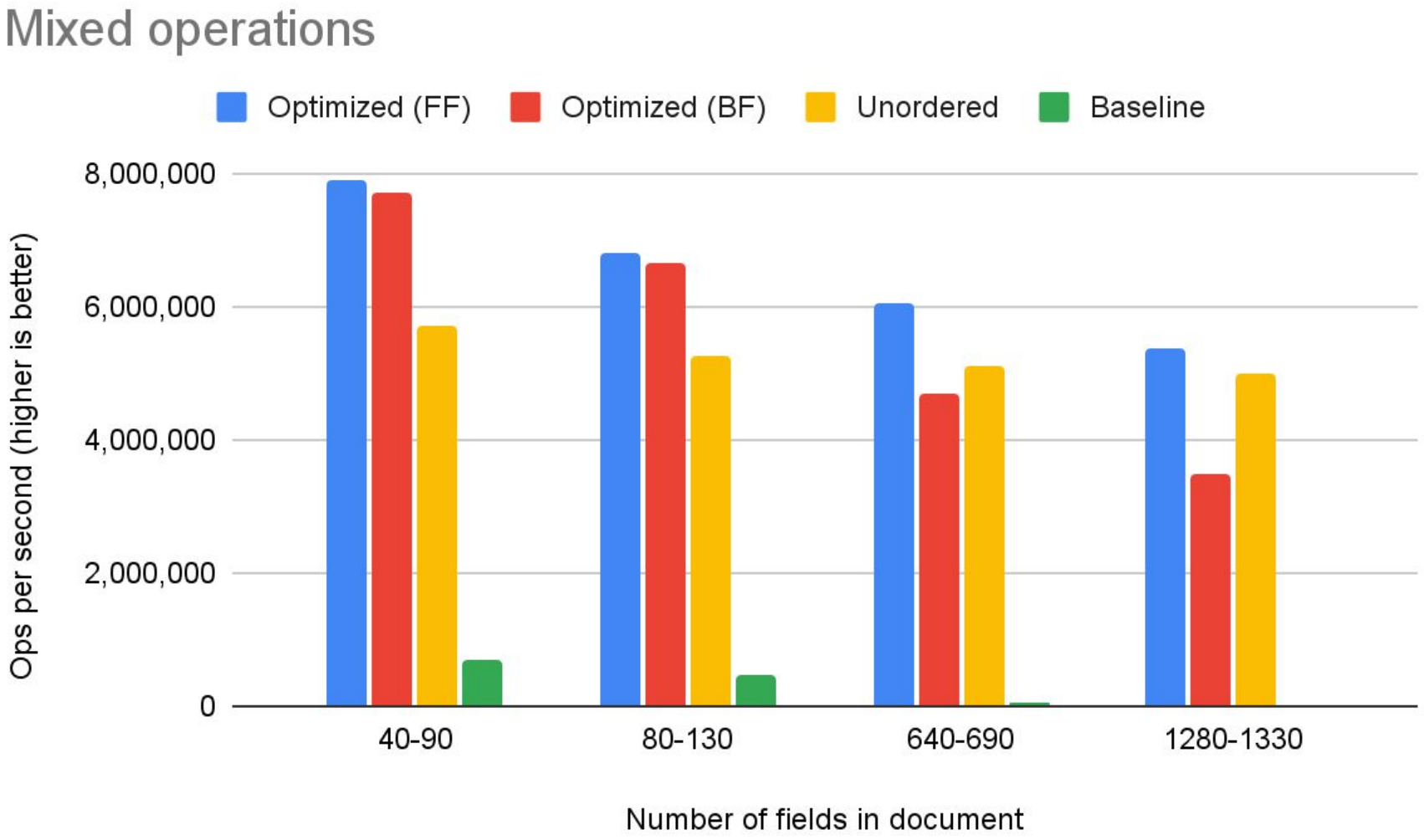
Final Benchmarks



Bloomberg

Engineering

Final Benchmarks



Bloomberg

Engineering

Limitations

- Fragmentation is a problem with first-fit
  - Can we achieve a middle-ground between first- and best-fit?

- Because of the added metadata necessary for performing searches, our format is not as compact as other schemaless formats
  - Our overhead is ~4 bytes per key/value pair in a map
  - Or ~2 bytes per value in an array

- Our format is full of offsets, so if we load a corrupted buffer, we could easily make out-of-bounds accesses
  - Our current implementation assumes it deals with a valid input buffer, so the results are based on that

Conclusions

- We’ve created a fast, schemaless serialization format which we can edit in place
- It compares favourably to using an `unordered_map` in memory
- By creating our own memory allocator and replacing pointers with offsets, we can put in-memory data structures into a small buffer

TechAtBloomberg.com

Bloomberg

Engineering

Conclusions

- When trying to improve performance:
  - Create highly parameterized benchmarks
  - Profile
  - Make algorithmic/data structure improvements
  - Explore vectorization, branchless code, prefetching

TechAtBloomberg.com

Bloomberg

Engineering

Future Work

- Explore how to compact the buffer efficiently
  - Compact in-place, or copy into a new buffer?
  - Could recent simd scatter/gather instructions help here?

- Explore building an offset-free serialization format
  - Could we build indexes of our data as needed, instead of storing them in the buffer?
  - Taking inspiration from things like `simdjson`

- Explore compressing repeated keys or fragments of keys to save space

Resources

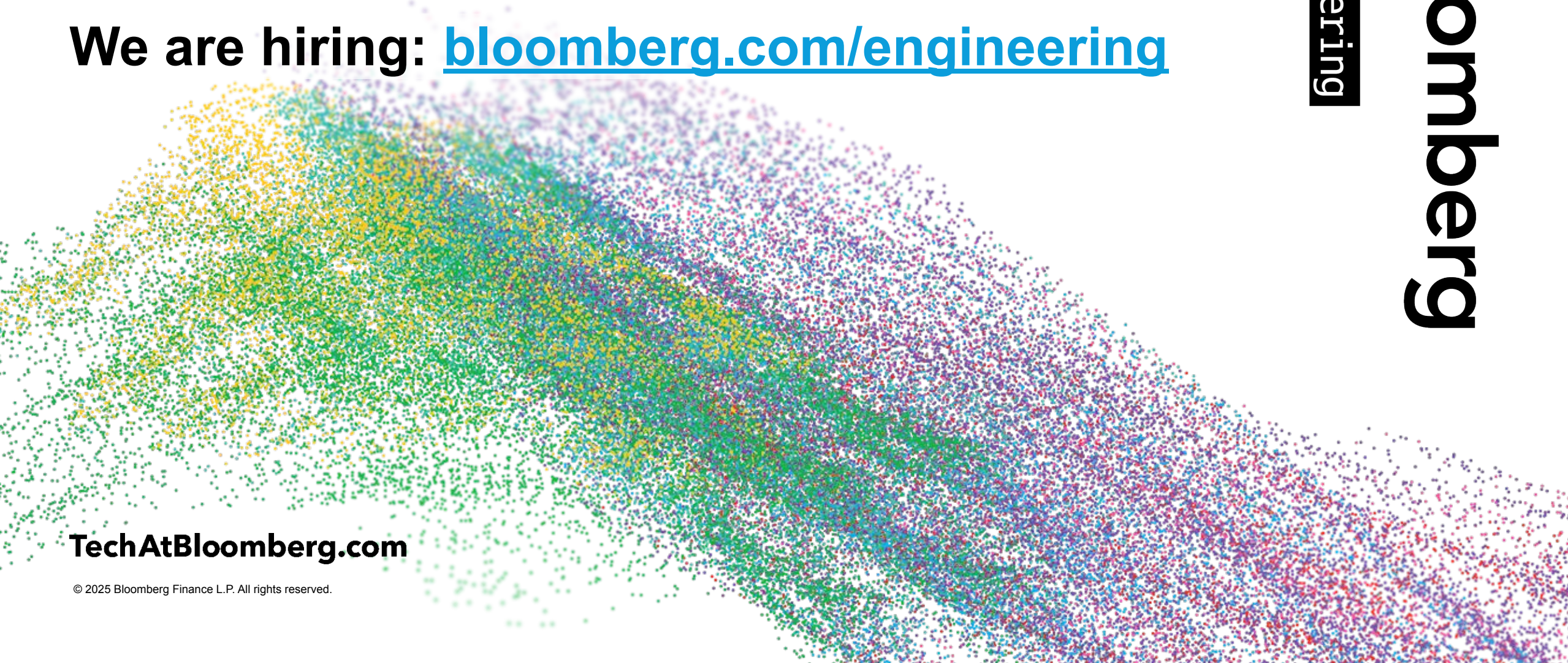
- “Algorithms for Modern Hardware” by Sergey Slotin, available online: https://en.algorithmica.org/hpc/
  - A comprehensive review of techniques for writing high performance code
- “The Garbage Collection Handbook” by Richard Jones, Antony Hosking and Eliot B. Moss
  - Covers a lot of memory allocation and buffer compaction techniques
- “Going Nowhere Faster” by Chandler Carruth at CppCon 2017: https://www.youtube.com/watch?v=2EWejmkKIxs
  - Good intro to benchmarking and profiling tools, including perf
- Fedor Pikus’s “Performance and Efficiency in C++” training
  - Covers all this stuff in a lot of depth!

Thank you!

We are hiring: bloomberg.com/engineering

Bloomberg

Engineering



TechAtBloomberg.com