(Not yet Adaptive) Compression of In-Memory Databases

Database Implementation Lab Course

Leon Windheuser

February 28, 2023

Project Introduction

Compression of the In-Memory part of DuckDB

Project Introduction

Compression of the In-Memory part of DuckDB

- Open Source SQL OLAP RDBMS developed in Amsterdam research centre CWI (`https://github.com/duckdb/duckdb`)
- Columnar Storage format
- Vectorized execution engine
- Has lots of different compression possibilities for persistent data on disk

Project Introduction

Compression of the In-Memory part of DuckDB

- Open Source SQL OLAP RDBMS developed in Amsterdam research centre CWI (`https://github.com/duckdb/duckdb`)
- Columnar Storage format
- Vectorized execution engine
- Has lots of different compression possibilities for persistent data on disk

How do we compress data in-memory while having efficient lookups without decompressing everything?

Succinct Data Structures

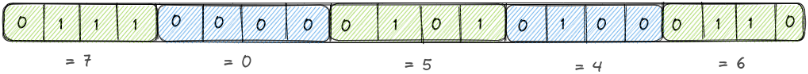
- Data structures which uses close to the *information-theoretic* lower bound of space but allows efficient query operations (in-place without needing to decompress)
- Exists for e.g. (bit) vectors, trees, planar graphs, ...

Succinct Data Structures

- Data structures which uses close to the *information-theoretic* lower bound of space but allows efficient query operations (in-place without needing to decompress)
- Exists for e.g. **(bit) vectors**, trees, planar graphs, ...

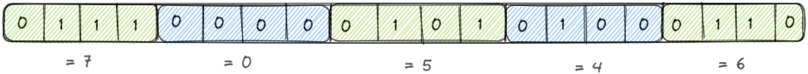
Succinct Integer Vector

Space requirement for integer x is $\ell = \lfloor \log_2(x) \rfloor + 1$ bits

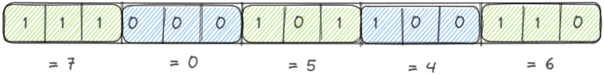


Succinct Integer Vector

Space requirement for integer x is $\ell = \lfloor \log_2(x) \rfloor + 1$ bits



Encode integers with the minimal length of the max integer
$3 = \lfloor \log_2(7) \rfloor + 1$



We already reduce memory by 25%

SDSL: Succinct Data Structure Library

- C++11 library and abstraction for succinct data structures
- Open Source `https://github.com/simongog/sdsl-lite`
- Contains varierty of different data structures. For now we only used the **Integer Vectors**.

SDSL: Integer Vectors

```
sdsl::int_vector<32> v(10000);
for (size_t i = 0; i < 10000; i++) v[i] = i;
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
sdsl::util::bit_compress(v);
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
```

SDSL: Integer Vectors

```
sdsl::int_vector<32> v(10000);
for (size_t i = 0; i < 10000; i++) v[i] = i;
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
sdsl::util::bit_compress(v);
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
```

```
Width: 32, size: 40008
Width: 14, size: 17513
```

SDSL: Integer Vectors

```
sdsl::int_vector<32> v(10000);
for (size_t i = 0; i < 10000; i++) v[i] = i;
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
sdsl::util::bit_compress(v);
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
```

```
Width: 32, size: 40008
Width: 14, size: 17513
```

Reduces memory by 56.2% ($\approx$ 22.5 KB)

Extract min element from sdsl::int_vector

```
sdsl::int_vector<32> v(10000);
for (size_t i = 0; i < 10000; i++)
    v[i] = i + 10.000.000;
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
sdsl::util::bit_compress(v);
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
```

```
Width: 32, size: 40008
Width: 24, size: 30008
```

Extract min element from sdsl::int_vector

```
sdsl::int_vector<32> v(10000);
for (size_t i = 0; i < 10000; i++)
    v[i] = i + 10.000.000;
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
extractMinFromVector(v);
sdsl::util::bit_compress(v);
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
```

```
Width: 32, size: 40008
Width: 14, size: 17513
```

Extract min element from sdsl::int_vector

```
sdsl::int_vector<32> v(10000);
for (size_t i = 0; i < 10000; i++)
    v[i] = i + 10.000.000;
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
extractMinFromVector(v);
sdsl::util::bit_compress(v);
cout << "Width: " << v.width() << ", size: "
     << sdsl::size_in_bytes(v) << endl;
```

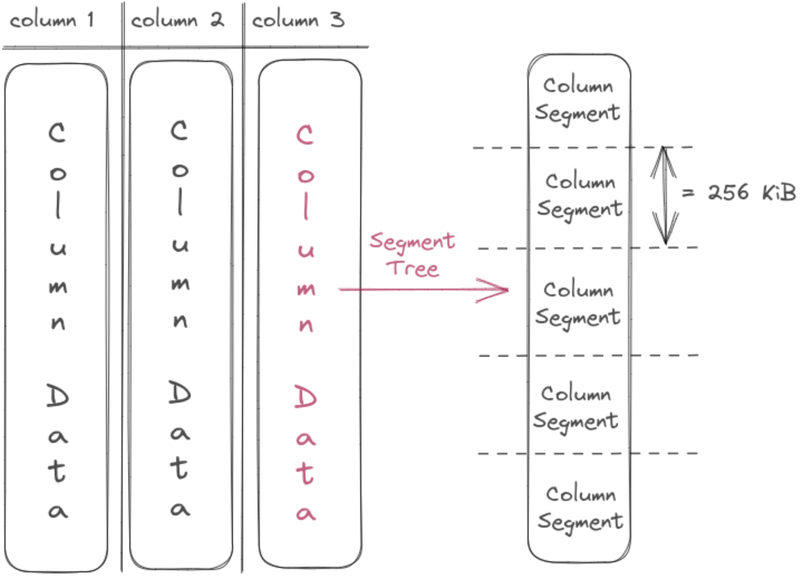
```
Width: 32, size: 40008
Width: 14, size: 17513
```

In DuckDB we know the minimum of the vector directly without searching (column statistics)

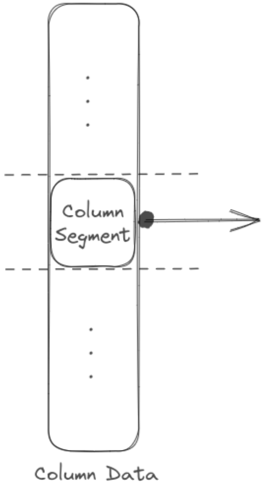
Random Access of `sdsl::int_vector[i]` vs `std::vector[i]`

Byte-Align `sdsl::int_vector`

DuckDB Storage Architecture 100 meter view



DuckDB Storage Architecture 10 meter view



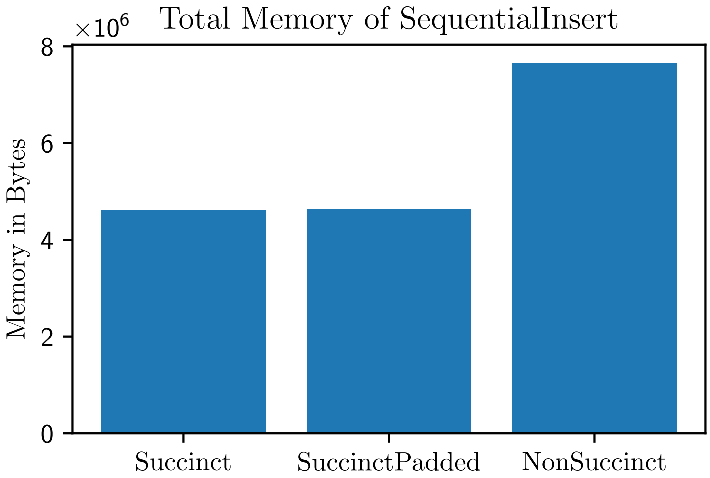

```
SegmentType: Transient / Persistent
block: Manager by BufferManager
CompressionType: Succinct / Uncompressed / ...
sdsl::int_vector<>

CompressionFunction::SuccinctFunctions{
    Append, Scan, PartialScan, ...
}
```

Benchmarks: Sequential Insert and total Scan

Scanning 10^6 rows.

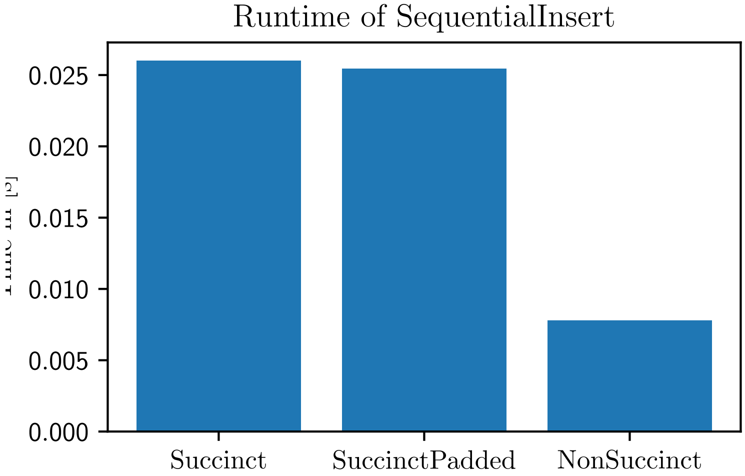
```
SELECT * FROM t1;
```



Benchmarks: Sequential Insert and total Scan

10.000 selections with Zipf Distribution of 10^6 total rows.

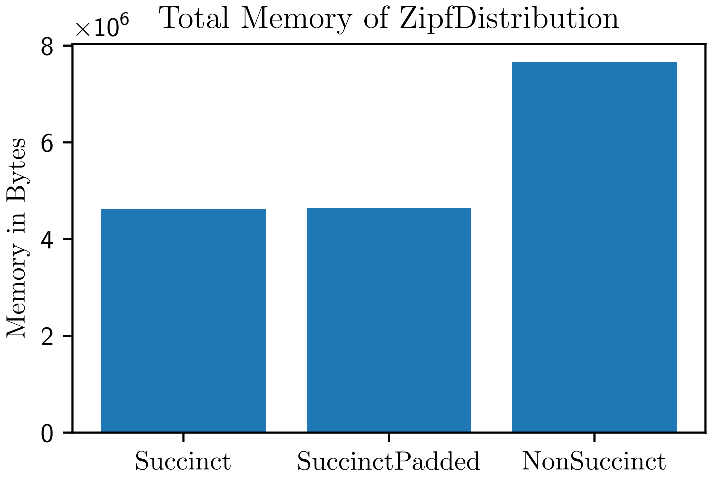
```
SELECT * FROM t1;
```



Benchmarks: Zipf Selection

10.000 selections with Zipf Distribution of 10^6 total rows.

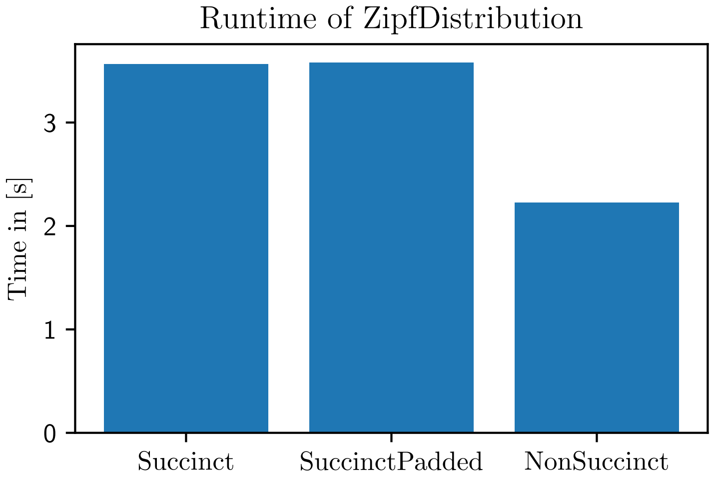
```
SELECT i FROM t1
WHERE i == {ZIPF_DISTRIBUTED_NUMBER};
```



Benchmarks: Zipf Selection

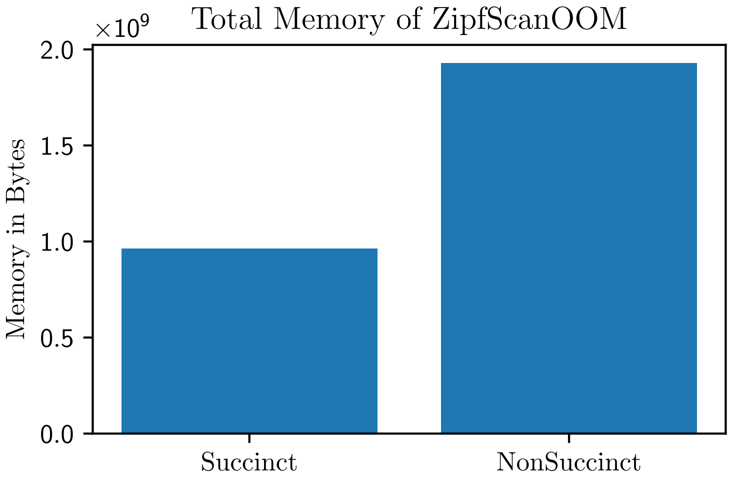
10.000 selections with Zipf Distribution of 10^6 total rows.

```
SELECT i FROM t1
WHERE i == {ZIPF_DISTRIBUTED_NUMBER};
```



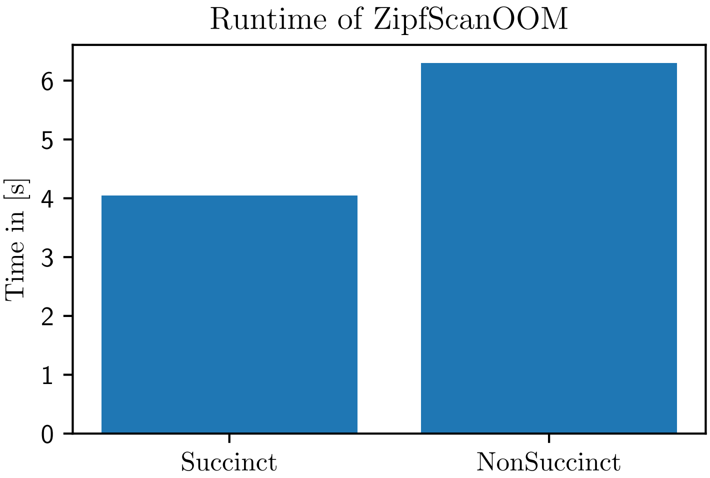
Benchmarks: Zipf Out Of Memory (Limit 1GB)

```
SELECT i FROM t1
WHERE i == {ZIPF_DISTRIBUTED_NUMBER};
```



Benchmarks: Zipf Out Of Memory (Limit 1GB)

```
SELECT i FROM t1
WHERE i == {ZIPF_DISTRIBUTED_NUMBER};
```



Conclusion and Future Work

- For OLAPish queries it is not (yet?) worth it, large overhead.
- For OLTP transactions it might be worth it. Reduces memory by $\approx 40\%$ but increases runtime by $\approx 35\%$.
- Huge benefit if succinct representation fits in memory vs spilling to disk.

Conclusion and Future Work

- For OLAPish queries it is not (yet?) worth it, large overhead.
- For OLTP transactions it might be worth it. Reduces memory by $\approx 40\%$ but increases runtime by $\approx 35\%$.
- Huge benefit if succinct representation fits in memory vs spilling to disk.

TODO:

- Need to investigate in access statistics to only compress rarely used column segments.
- Currently copying a lot of data as execution engine expects an uncompressed flat vector. Adapt execution engine to allow succinct vectors as well since they allow random access.