(Not yet Adaptive) Compression of In-Memory Databases

Database Implementation Lab Course

Leon Windheuser

March 14, 2023

Project Introduction

We want to compress the transient part of DuckDB

Project Introduction

We want to compress the transient part of DuckDB

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

Project Introduction

We want to compress the transient part of DuckDB

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

How do we compress the transient data while having efficient lookups without decompressing everything?

Background: Succinct Data Structures

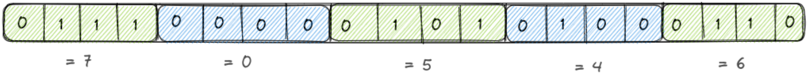
- Data structures which uses space close to the theoretic lower bound but allows efficient query operations (in-place without needing to decompress)
- Exists for e.g. (bit) vectors, trees, planar graphs, ...

Background: Succinct Data Structures

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

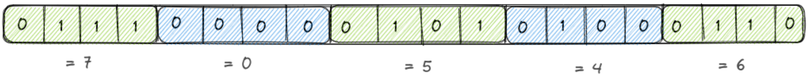
Succinct Integer Vector

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

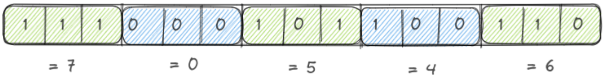


Succinct Integer Vector

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)

Delta compression of 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
```

Delta compression of 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
```

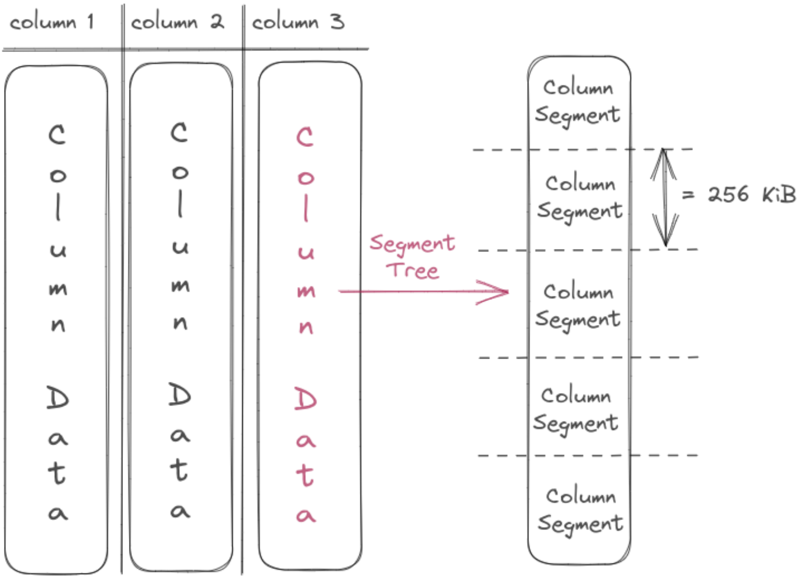
Delta compression of sdsl::int_vector

```
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;
```

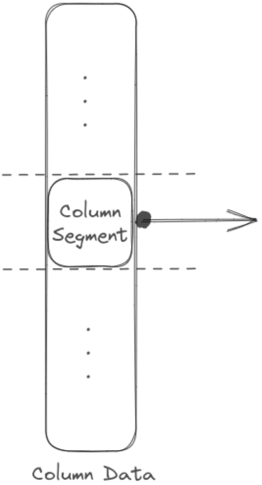
```
Width: 32, size: 40008
Width: 14, size: 17513
```

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

DuckDB Storage Architecture 100 meter view



DuckDB Storage Architecture 10 meter view



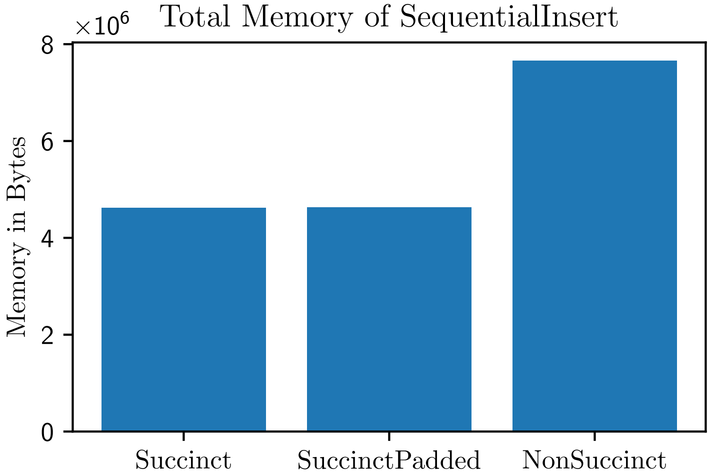

```
SegmentType: Transient / Persistent
block: Manager by BufferManager
CompressionType: Succinct / Uncompressed / ...
sdsl::int_vector<>

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

Benchmarks: Sequential Insert and total Scan

Scanning 10^6 rows.

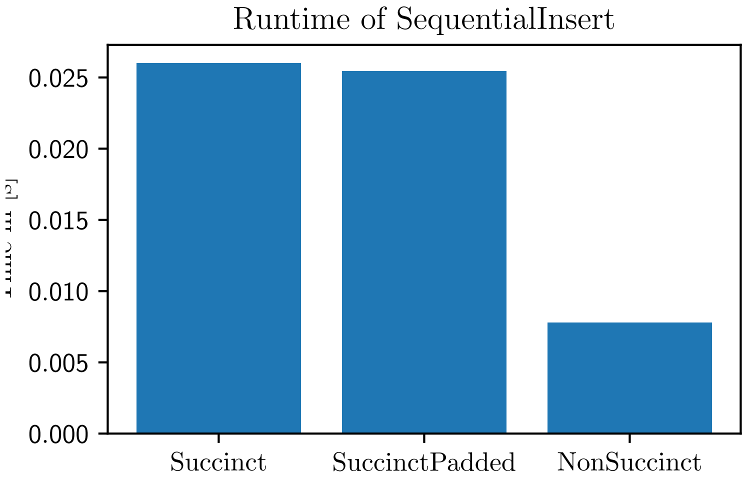
```
SELECT * FROM t1;
```



Benchmarks: Sequential Insert and total Scan

Scanning 10^6 rows.

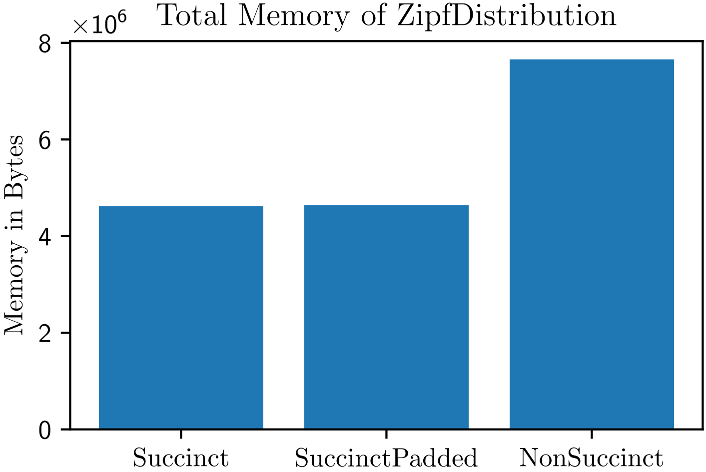
```
SELECT * FROM t1;
```



Benchmarks: Zipf Selection

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

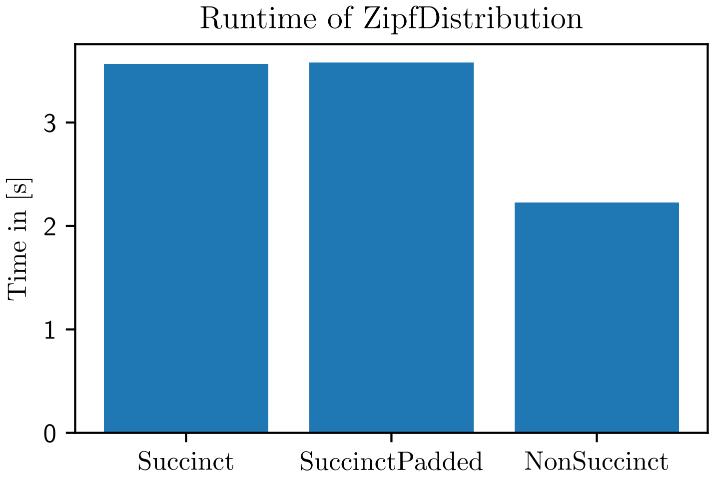
```
SELECT i FROM t1
WHERE i == {ZIPF_DISTRIBUTED_NUMBER};
```



Benchmarks: Zipf Selection

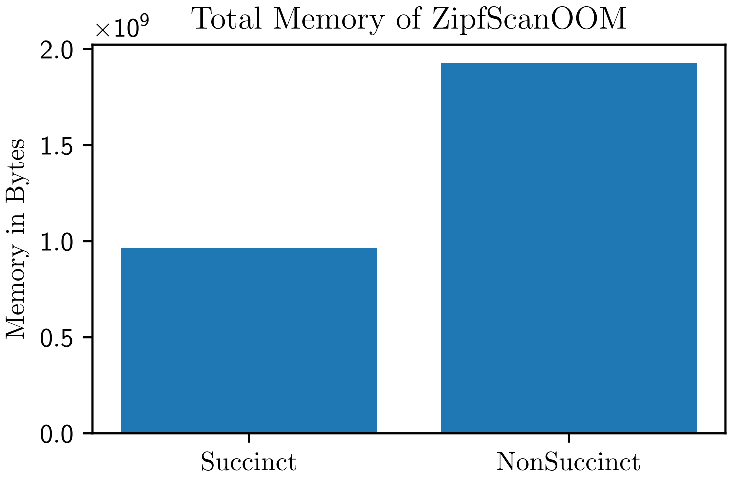
10.000 selections with Zipf Distribution of 10^6 total rows.

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



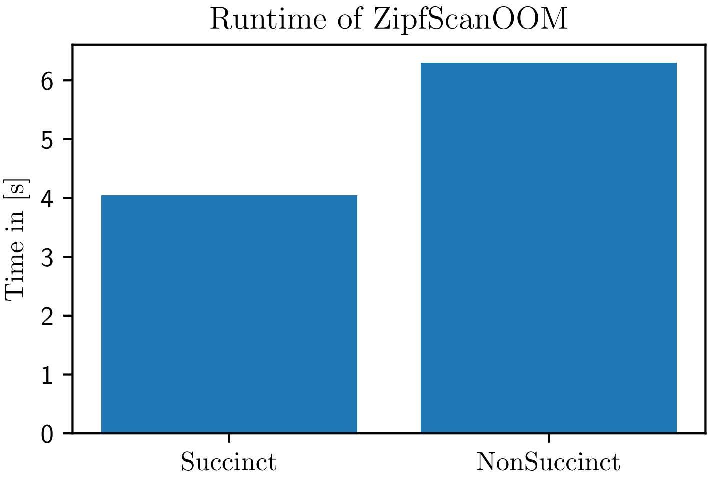
Benchmarks: Zipf Out Of Memory (Limit 1GB)

```
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

- 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.

Future Work and Discussion

1. Copying and shifting data is most time consuming (40%) since (DuckDB) execution engine expects a flat "normal" vector for OLAP queries (e.g. `SELECT *`).
   - Non succinct just passes its data pointer, we need to decompress everything and copy the data.
   - Unecessary, since we still support random access and all operations needed for the execution engine.
   - Non succinct data pointer used everywhere in the execution engine ($>$ 300 appereances). **Is a major rewrite necessary?**

2. Adaptive compression for rarely accessed column segments. Zipf Distribution accesses 4 of 50 segements over $>$ 70%.
   - How to track access statistics over time for column segments?
   - What if the access statistics change after greater period of time?