(Not yet Adaptive) Compression of In-Memory Databases

Database Implementation Lab Course

Leon Windheuser

March 14, 2023

Project Introduction

Goal: We want to compress the transient part of DuckDB

Project Introduction

Goal: We want to compress the transient part of DuckDB

- Open Source SQL OLAP RDBMS in-process developed in Amsterdam research center 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

Goal: We want to compress the transient part of DuckDB

- Open Source SQL OLAP RDBMS in-process developed in Amsterdam research center 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

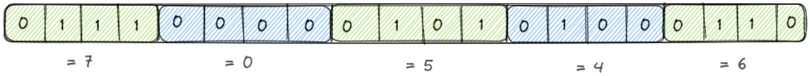
- Data structures that use space close to the theoretic lower bound but allow efficient query operations (in place without needing to decompress)
- Exists e.g. (bit) vectors, trees, planar graphs, ...

Background: Succinct Data Structures

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

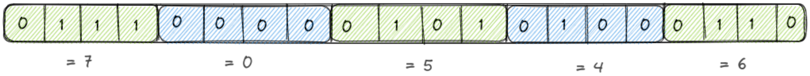
Succinct Integer Vector

Space requirement for integer x is $\ell = \lceil \log_2(x) \rceil$ bits



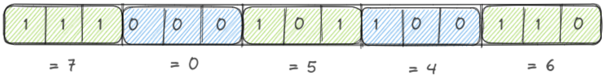
Succinct Integer Vector

Space requirement for integer x is $\ell = \lceil \log_2(x) \rceil$ bits



Encode integers with the minimal length of the max integer

$$3 = \lceil \log_2(7) \rceil$$



We already reduce memory by 25%

SDSL: Succinct Data Structure Library

- C++11 library for succinct data structures
- Open Source `https://github.com/simongog/sdsl-lite`
- Contains a variety of succinct data structures. We use **Integer Vector** to compress integer columns.

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
```

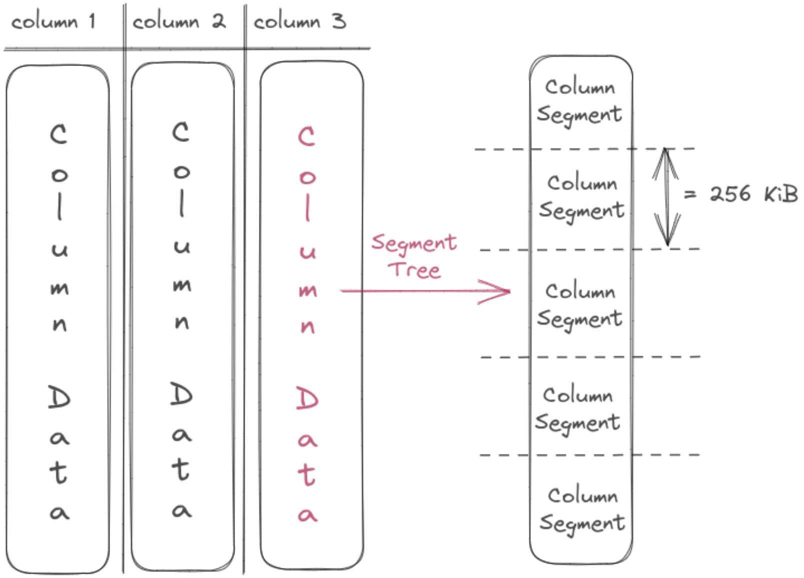
Delta Compression of sdsl::int_vector

```
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;
```

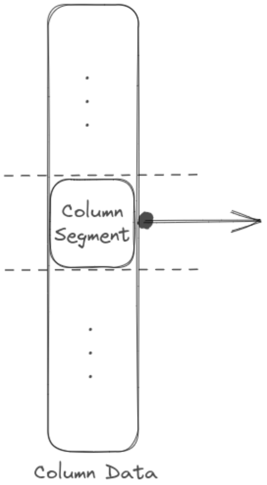
```
Width: 32, size: 40008
Width: 14, size: 17513
```

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

DuckDB Storage Architecture Bird's Eye View (100 meters)



DuckDB Storage Architecture Bird's Eye View (10 meters)



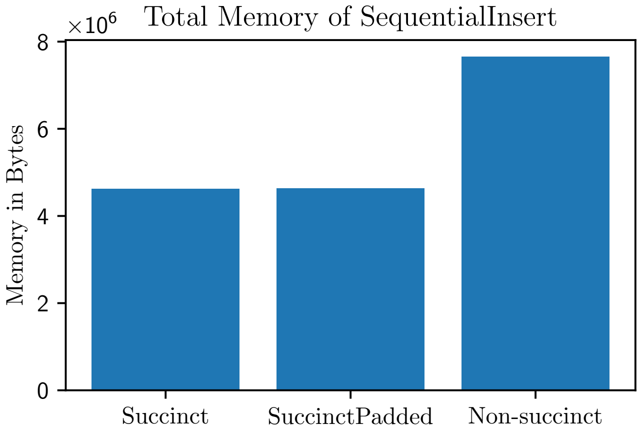
```
SegmentType: Transient / Persistent
block: Managed by BufferManager
CompressionType: Succinct / Uncompressed / ...
sdsl::int_vector<>
Compact()
Uncompact()

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

Evaluation: Sequential Insert and Total Scan

Scanning 1M rows.

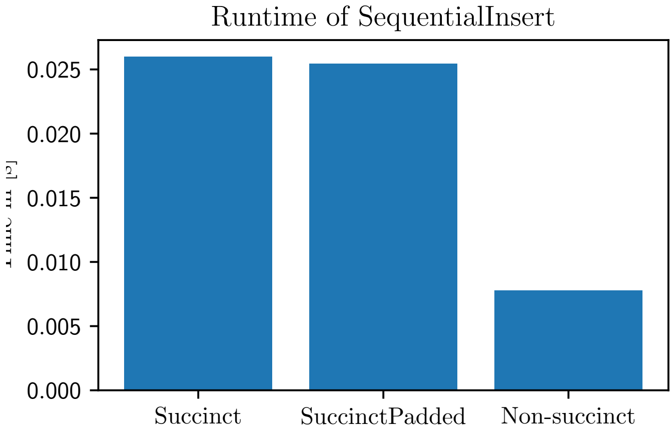
```
SELECT * FROM t1;
```



Evaluation: Sequential Insert and Total Scan

Scanning 1M rows.

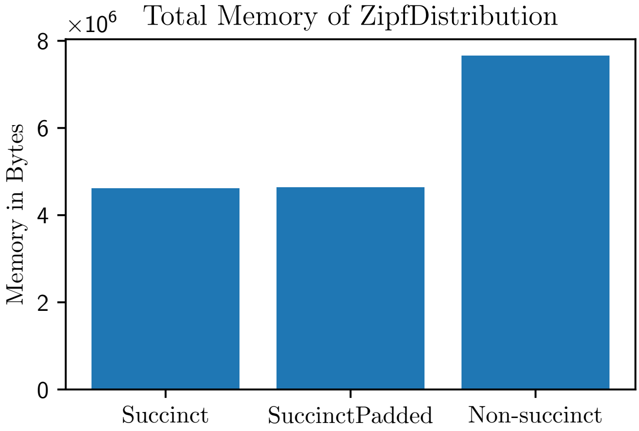
```
SELECT * FROM t1;
```



Evaluation: Zipf Selection

10000 selections with Zipf distribution of 1M total rows.

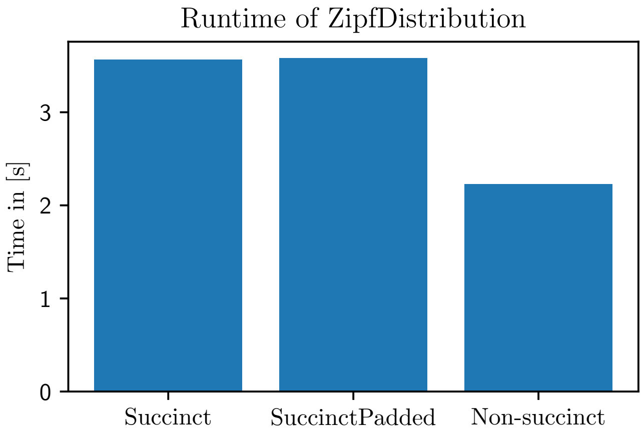
```
SELECT i FROM t1
WHERE i == {ZIPF_DISTRIBUTED_NUMBER};
```



Evaluation: Zipf Selection

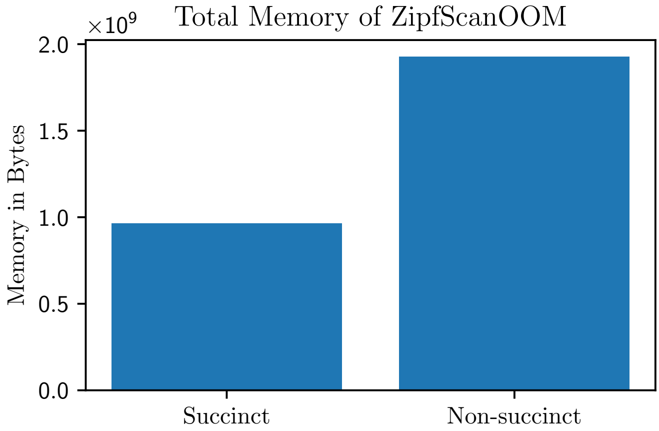
10000 selections with Zipf distribution of 1M total rows.

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



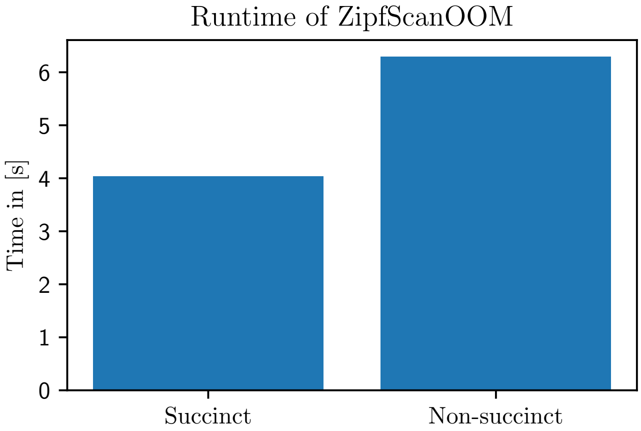
Evaluation: Zipf Out-Of-Memory (Limit 1GB)

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

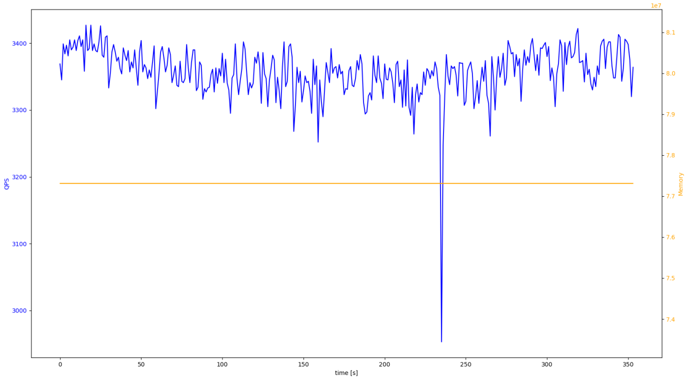


Evaluation: Zipf Out-Of-Memory (Limit 1GB)

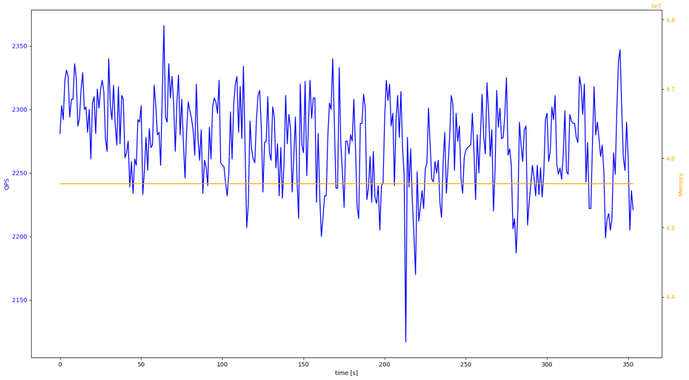
```
SELECT i FROM t1
WHERE i == {ZIPF_DISTRIBUTED_NUMBER};
```



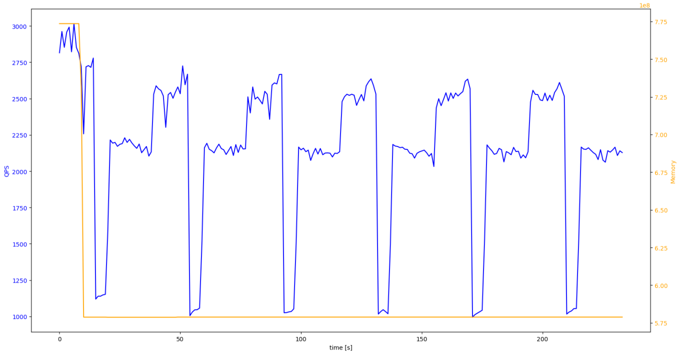
Zipf Over Time: Not Succinct



Zipf Over Time: Succinct



Zipf Over Time: Adaptive Succinct



Conclusion

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

Future Work and Discussion

1. Copying and shifting data is most time consuming ($\approx 40\%$) since execution engine expects a flat “normal” vector.
   - Non-succinct passes its data pointer that we need to decompress and copy the data.
   - Unnecessary since we still support random access and operations needed for the execution engine.
   - Non-succinct data pointer used everywhere in the execution engine (> 300 appearances). **Rewrite necessary?**

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