DuckDB CWI

Fastest Table Sort in the West - Redesigning DuckDB's Sort

Laurens Kuiper

Sorting

- One of the most well-studied problems in CS
  - Cache-efficiency
  - Worst-case analysis
  - Parallelism
  - …
  - … Arrays!

Sorting Relational Data

- Use cases:
  - ORDER BY
  - WINDOW
  - Sort-Merge Join
  - Inequality Join
  - ...

Sorting Relational Data

- Performance challenges
  - Multiple order clauses
  - Different types
  - Columns vs. Rows
  - …

Sorting Relational Data

- We distinguish two column types:

```
SELECT *                          → Payload columns
FROM customer
ORDER BY
    c_birth_country DESC,         → Key columns
    c_birth_year ASC;             → Key columns
```

- Both key and payload columns must be ordered!

The Cost of Sorting

- Dominated by
  - Comparing values
  - Moving data
- This presentation: Comparing key column values
  - Focus: Columnar representation

Comparing Values

- Sorting relational data: row-wise operation

- How to implement comparison for columns?
  - Need to sort indices:

```
bool Compare(columns, l_row_id, r_row_id):
    for col : columns:
        if col[l_row_id] != col[r_row_id]:
            return col[l_row_id] < col[r_row_id];
    return false;
```

Branch

Pointer chasing

Pointer Chasing

- Solution: pack key columns into a row

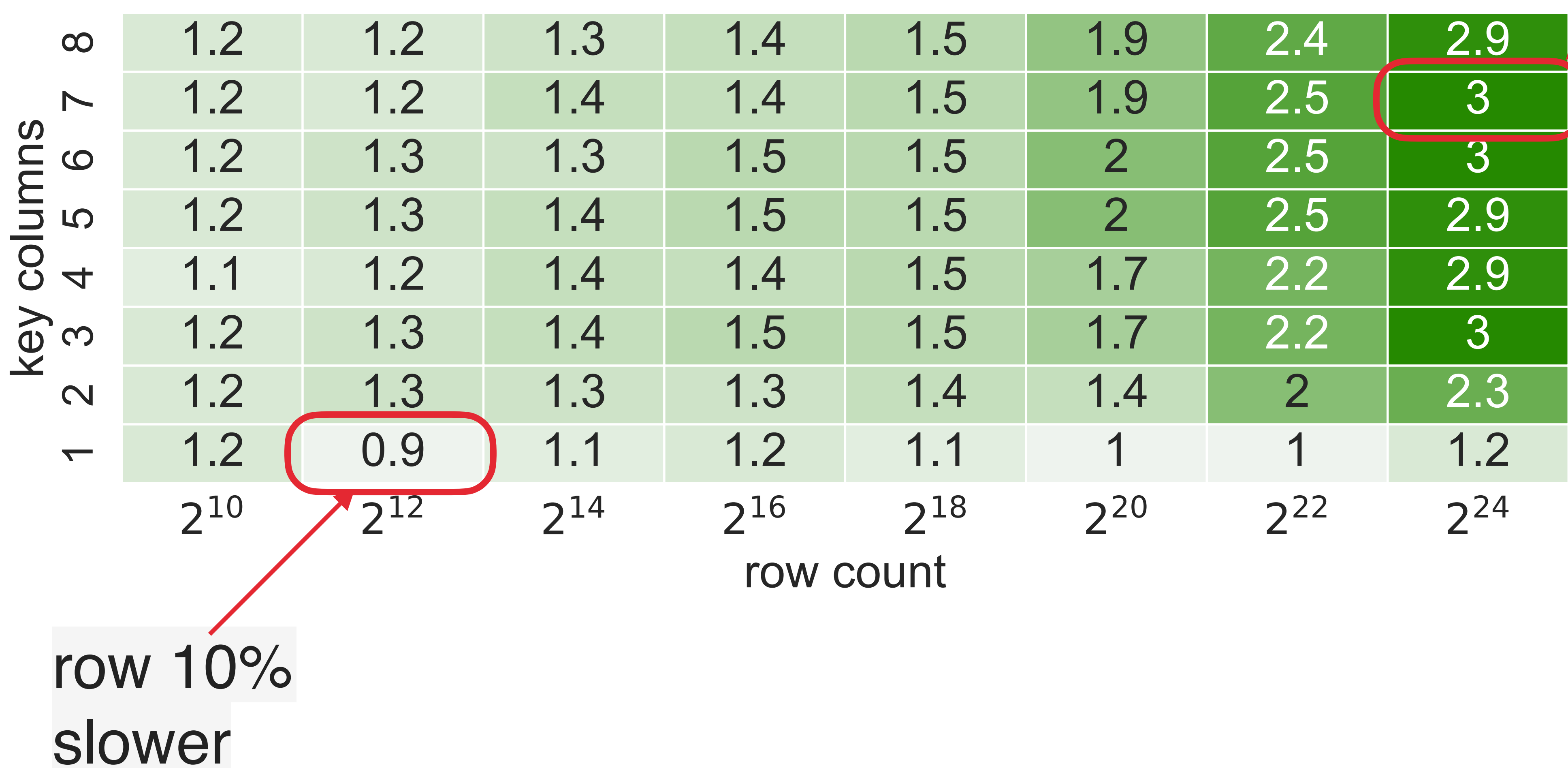
- No chasing pointers by index
  - Sort data directly
- Better locality
  - Values in row are co-located with each other

Pointer Chasing

- Simulated experiment:
    - 2^{10} to 2^{24} tuples
    - 1 to 8 key columns (uint32_t)
    - Data distribution: many ties
- Measure relative runtime difference
- Hardware: 2020 MacBook Pro (M1)

Pointer Chasing

- Sorting row data vs. column data



- Time includes column to row transformation!

Branch Prediction

```
SELECT *
FROM customer
ORDER BY
    c_birth_country DESC,
    c_birth_year ASC;
```

- Comparison has many branches:
  - If c_birth_country equal, compare c_birth_year
  - ASC/DESC
  - NULLS FIRST/LAST

Key Normalization

- Encode keys as binary string

```
SELECT *
FROM customer
ORDER BY
    c_birth_country DESC,
    c_birth_year ASC;
```

(a)

birth_country	birth_year
NETHERLANDS	1992
GERMANY	1924

(b)

birth_country	birth_year
78 69 84 72 69 82 76 65 78 68 83 0	200 7 0 0
71 69 82 77 65 78 89 0	132 7 0 0

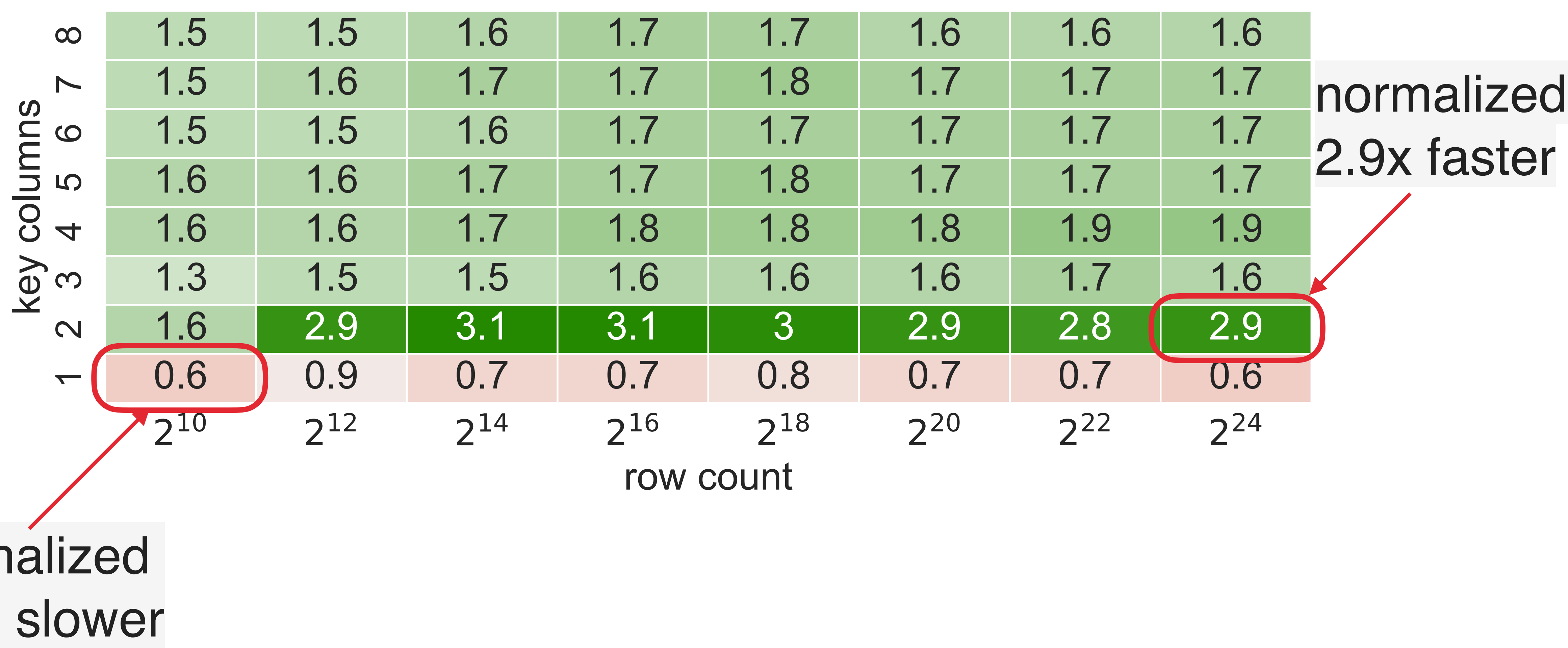
(c)

binary string
177 186 171 183 186 173 179 190 177 187 172 255 128 0 7 200
184 186 173 178 190 177 166 255 255 255 255 255 128 0 7 132

- Eliminates branch predictions from comparison

Key Normalization

- Sorting normalized row data vs. row data



- Time includes creating the normalized keys!

Comparing Values: Summary

- Comparing rows: Eliminates pointer chasing

- Key normalization: Eliminates branch predictions

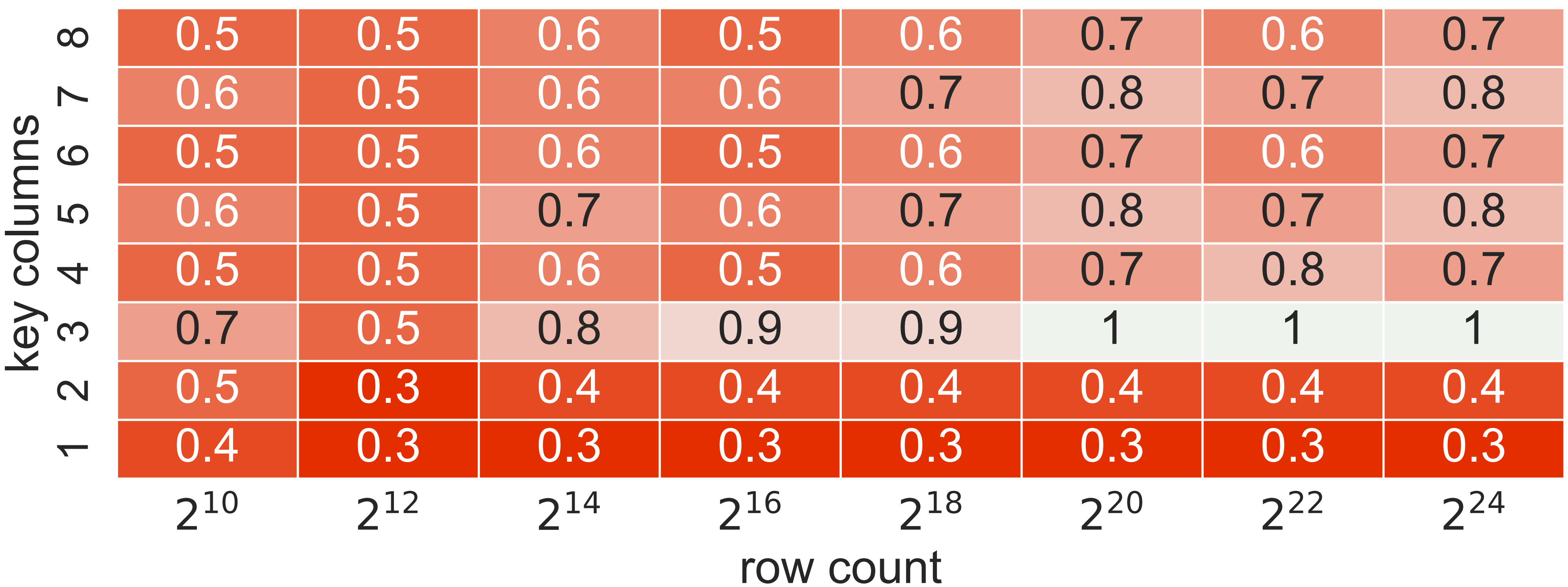
- Both optimizations are almost always worth it

Sorting Algorithm

- Key normalization enables byte-by-byte Radix Sort
- Key size $\leq$ 4 bytes: LSD
- Key size $>$ 4 bytes: MSD
  - Insertion sort

Radix Sort vs. Quicksort

- Looks pretty bad for Radix Sort …



- … or does it?

Radix Sort vs. Quicksort

- Simulation knows key size at compile time
  - In practice we don’t
- How much does pdqsort benefit from this?

Radix Sort vs. Quicksort

- Now with dynamic comparison:

key columns \ row count	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
8	0.6	0.5	0.7	0.6	0.7	0.8	0.8	0.9
7	0.7	0.5	0.7	0.6	0.7	0.9	0.8	1
6	0.6	0.5	0.6	0.5	0.7	0.8	0.7	0.9
5	0.7	0.5	0.7	0.6	0.8	1	0.9	1
4	0.6	0.5	0.7	0.5	0.7	0.9	0.9	0.8
3	1.7	1.3	2.2	2.5	2.4	2.8	2.9	2.8
2	1.7	1.5	2	2.2	2.2	2.2	2.2	2.1
1	1.5	1.5	1.4	1.5	1.4	1.3	1.4	1.4

- Still cumbersome: many struct definitions

Radix Sort vs. Quicksort

- To make quicksort efficient we could:
  - Create a lot of templated structs/functions
  - … and blow up our binary size :(
  - At this point it becomes an arms race

Sorting Algorithm: Summary

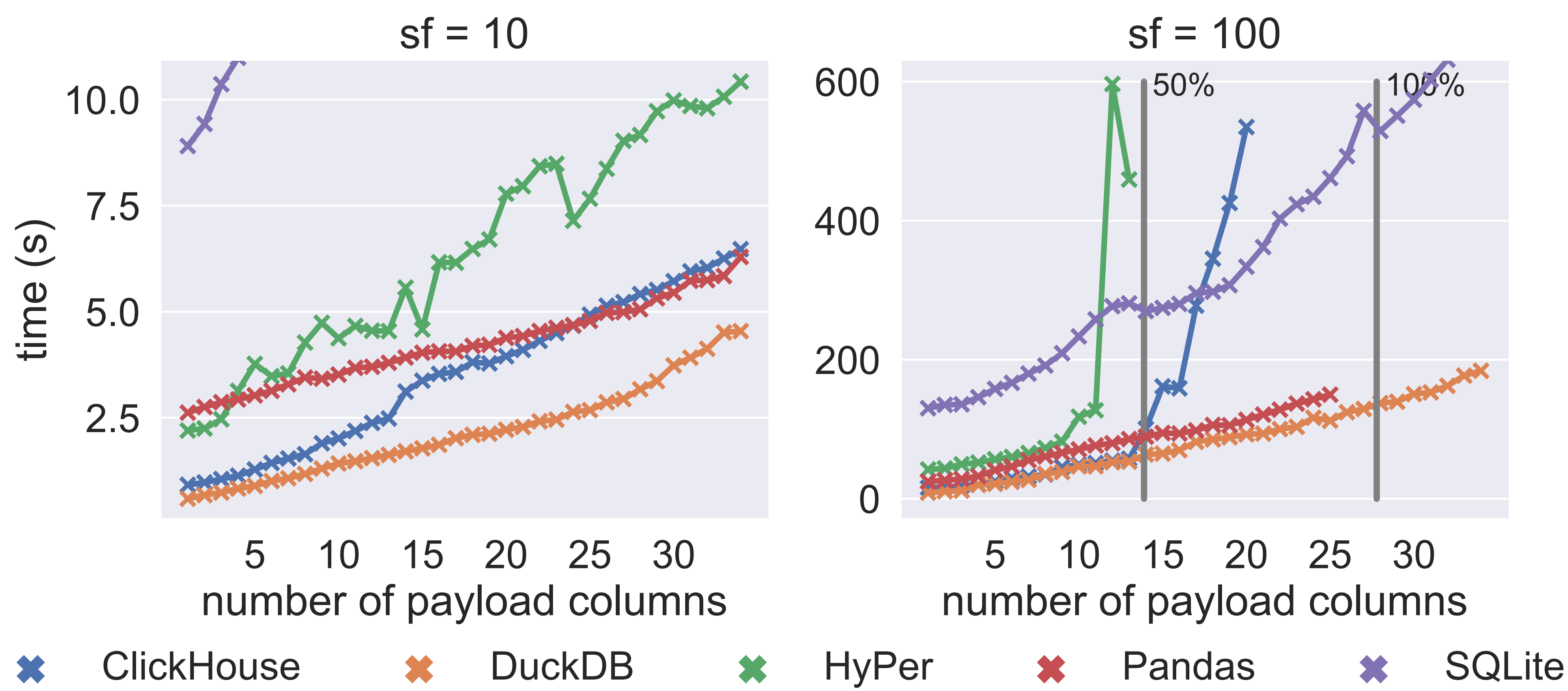
- Performance depends on key size / distribution
- Quicksort needs compile-time optimization
  - Radix Sort does not
- Efficient quicksort is possible for relational data
  - Comes at a cost
  - … or for free for JIT systems

End to end performance?

- Relational sorting benchmark using TPC-DS
- catalog_sales
  - 34 columns
  - SF10: 14.4M rows
  - SF100: 144M rows
- Ordered by cs_quantity, cs_item_sk

TPC-DS catalog_sales

- Increasing number of payload columns



- Note: 3GB/s SSD write speed

Wrapping Up

- Sorting relational data efficiently is challenging
- Performance is impacted by:
  - Random access
  - Branch predictions
- We can mitigate these problems with:
  - Row layout in memory
  - Key normalization
- Trade-off between Radix Sort and Quicksort

Fastest Table Sort in the West - Redesigning DuckDB's Sort

Laurens Kuiper

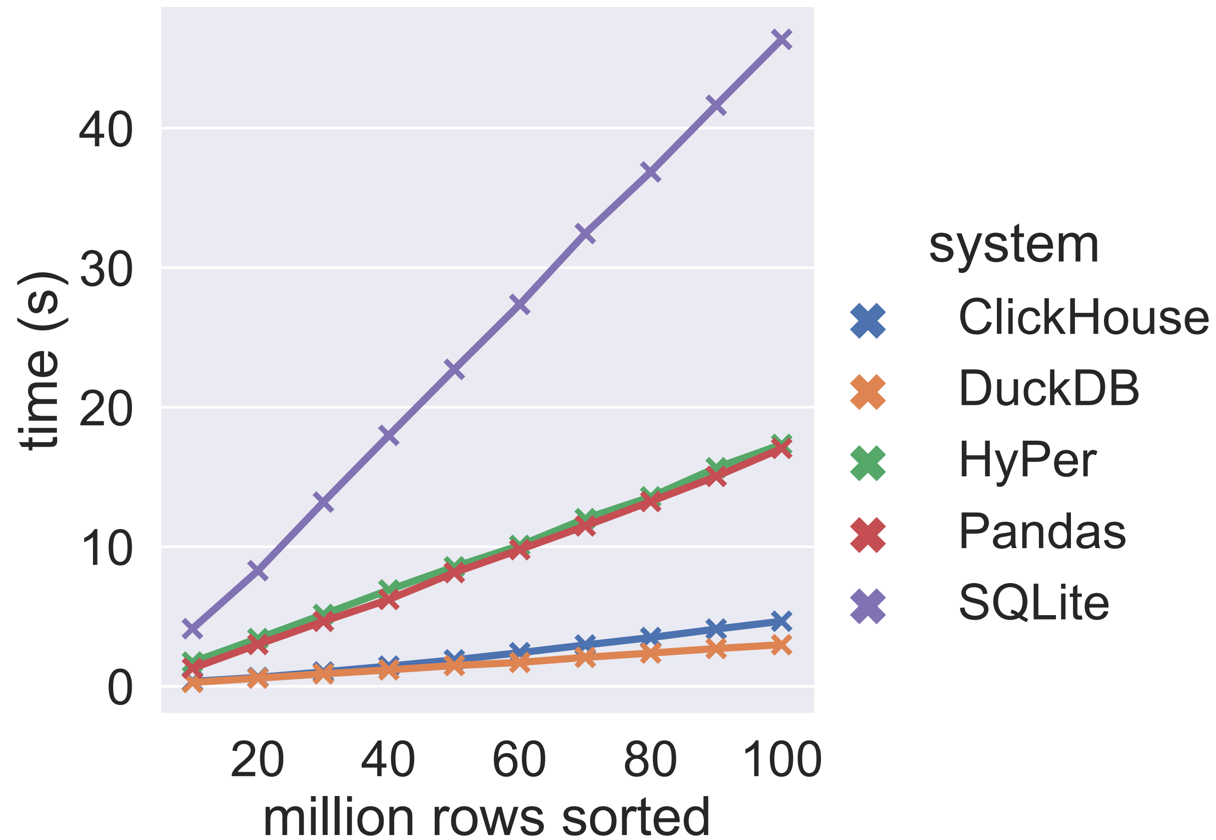
Key Normalization

- Strings:
  - Encode a prefix
  - Collation can be encoded

- NULL values:
  - Encode using additional byte

Random Integers

- 10-100M random integers



- … Actual relational data in the next experiment!

Parallelism

- DuckDB uses Morsel-Driven Parallelism
  - Threads collect data locally
  - Each thread sorts its own data
- Merge Sort needed for final result!

Merge Sort

- Parallelized using Merge Path

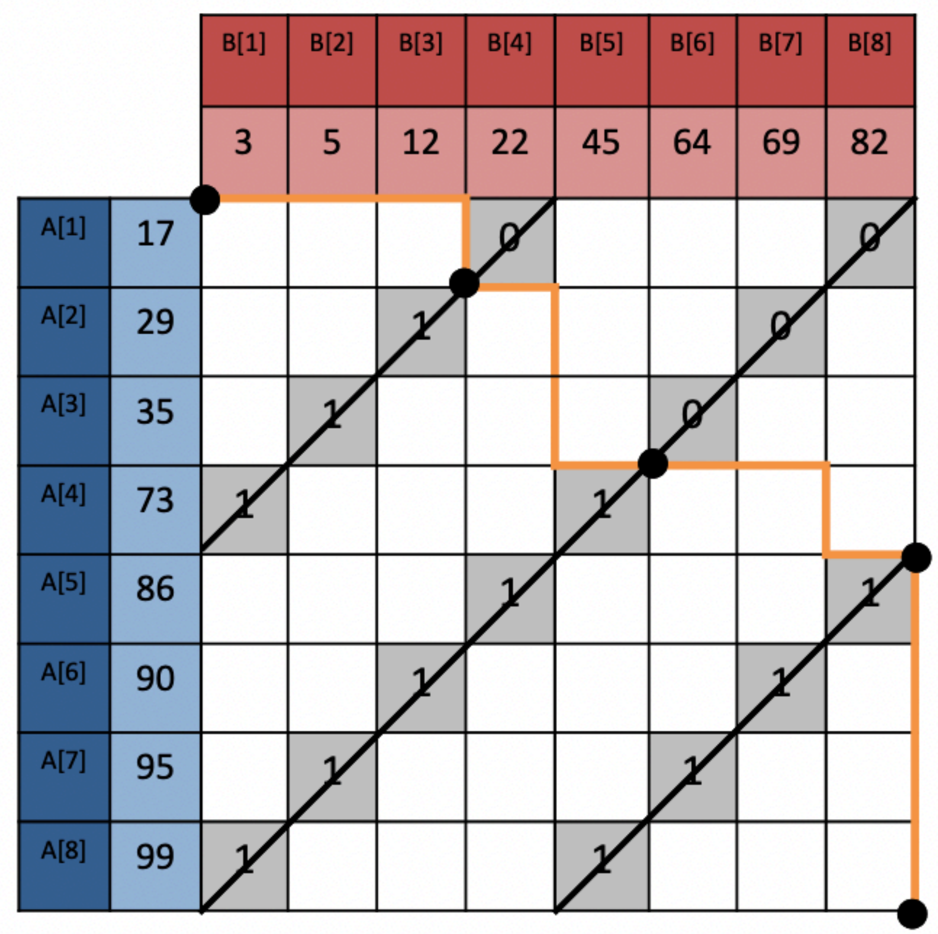
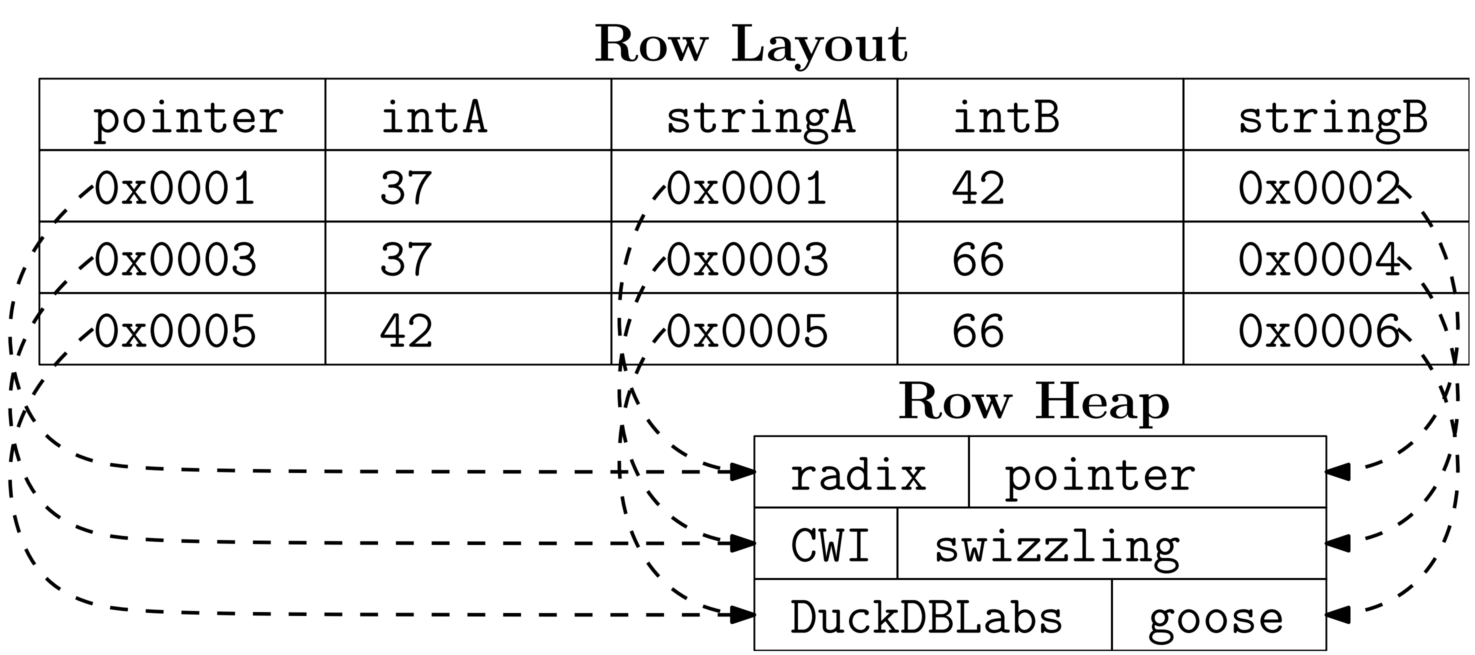


Image by Oded Green et al.

External Sorting

- Serialize payload in row format
- Buffer-managed blocks



External Sorting

- Heap also uses rows
- Swizzle pointer -> offset

Row Layout

offset	intA	stringA	intB	stringB
0	37	0	42	5
12	37	0	66	3
24	42	0	66	10

Row Heap

radix	pointer
CWI	swizzling
DuckDBLabs	goose

External Sorting

- External sorting is made possible because of
  - merge sort
  - buffer manager
  - pointer swizzling
- Modern hardware helps too!