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

Laurens Kuiper
Introduction

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 * FROM customer
  ORDER BY
    c_birth_country DESC,
    c_birth_year ASC;

- 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:

```cpp
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

  \[
  \begin{array}{cccc}
  id & c_0 & c_1 & \cdots & c_k \\
  \end{array}
  \quad \rightarrow \quad
  \begin{array}{cccc}
  c_0 & c_1 & \cdots & c_k & id \\
  \end{array}
  \begin{array}{cccc}
  c_0 & c_1 & \cdots & c_k & id \\
  \vdots \\
  c_0 & c_1 & \cdots & c_k & id \\
  \end{array}
  \]

- 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**

<table>
<thead>
<tr>
<th>key columns</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>1.2</td>
<td>1.3</td>
<td>1.4</td>
<td>1.5</td>
<td>1.9</td>
<td>2.4</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>1.2</td>
<td>1.4</td>
<td>1.4</td>
<td>1.5</td>
<td>1.9</td>
<td>2.5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.3</td>
<td>1.5</td>
<td>1.5</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.4</td>
<td>1.5</td>
<td>1.5</td>
<td>2</td>
<td>2.5</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>1.2</td>
<td>1.4</td>
<td>1.4</td>
<td>1.5</td>
<td>1.7</td>
<td>2.2</td>
<td>2.9</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.4</td>
<td>1.5</td>
<td>1.5</td>
<td>1.7</td>
<td>2.2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
<td>1.4</td>
<td>1.4</td>
<td>2</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>0.9</td>
<td>1.1</td>
<td>1.2</td>
<td>1.1</td>
<td>1</td>
<td>1</td>
<td>1.2</td>
<td></td>
</tr>
</tbody>
</table>

- **Time includes column to row transformation!**

- Row 10% slower

- Row 3x faster
Branch Prediction

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

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

- Encode keys as binary string

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

<table>
<thead>
<tr>
<th>birth_country</th>
<th>birth_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETHERLANDS</td>
<td>1992</td>
</tr>
<tr>
<td>GERMANY</td>
<td>1924</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>birth_country</th>
<th>birth_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>78 69 84 72 69 82 76 65 78 68 83 0</td>
<td>200 7 0 0</td>
</tr>
<tr>
<td>71 69 82 77 65 78 89 0</td>
<td>132 7 0 0</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>binary string</th>
</tr>
</thead>
<tbody>
<tr>
<td>177 186 171 183 186 173 179 190 177 187 172 255 128 0 7 200</td>
</tr>
<tr>
<td>184 186 173 178 190 177 166 255 255 255 255 255 128 0 7 132</td>
</tr>
</tbody>
</table>

(c)

- Eliminates branch predictions from comparison
Key Normalization

- Sorting normalized row data vs. row data

<table>
<thead>
<tr>
<th>key columns</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 ...

<table>
<thead>
<tr>
<th>key columns</th>
<th>row count</th>
<th>2^10</th>
<th>2^12</th>
<th>2^14</th>
<th>2^16</th>
<th>2^18</th>
<th>2^20</th>
<th>2^22</th>
<th>2^24</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>0.5 0.5 0.6 0.5 0.6 0.7 0.6 0.7</td>
<td>0.6 0.5 0.6 0.6 0.7 0.8 0.7 0.8</td>
<td>0.5 0.5 0.6 0.5 0.6 0.7 0.8 0.7</td>
<td>0.7 0.5 0.8 0.9 0.9 1.0 1.0 1.0</td>
<td>0.5 0.3 0.4 0.4 0.4 0.4 0.4 0.4</td>
<td>0.4 0.3 0.3 0.3 0.3 0.3 0.3 0.3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ... 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:

<table>
<thead>
<tr>
<th>key columns</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>0.7</td>
<td>0.6</td>
<td>0.7</td>
<td>0.8</td>
<td>0.8</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>0.5</td>
<td>0.7</td>
<td>0.6</td>
<td>0.7</td>
<td>0.9</td>
<td>0.8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>0.6</td>
<td>0.5</td>
<td>0.7</td>
<td>0.8</td>
<td>0.7</td>
<td>0.9</td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>0.5</td>
<td>0.7</td>
<td>0.6</td>
<td>0.8</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>0.5</td>
<td>0.7</td>
<td>0.5</td>
<td>0.7</td>
<td>0.9</td>
<td>0.9</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>1.3</td>
<td>2.2</td>
<td>2.5</td>
<td>2.4</td>
<td>2.8</td>
<td>2.9</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>1.5</td>
<td>2</td>
<td>2.2</td>
<td>2.2</td>
<td>2.2</td>
<td>2.2</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>1.5</td>
<td>1.4</td>
<td>1.5</td>
<td>1.4</td>
<td>1.3</td>
<td>1.4</td>
<td>1.4</td>
<td></td>
</tr>
</tbody>
</table>

- 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!
Parallelism

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

<table>
<thead>
<tr>
<th>offset</th>
<th>intA</th>
<th>stringA</th>
<th>intB</th>
<th>stringB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>37</td>
<td>0</td>
<td>42</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>37</td>
<td>0</td>
<td>66</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>42</td>
<td>0</td>
<td>66</td>
<td>10</td>
</tr>
</tbody>
</table>

| Row Heap
<table>
<thead>
<tr>
<th>radix</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWI</td>
<td>swizzling</td>
</tr>
<tr>
<td>DuckDBLabs</td>
<td>goose</td>
</tr>
</tbody>
</table>
External Sorting

- External sorting is made possible because of
  - merge sort
  - buffer manager
  - pointer swizzling

- Modern hardware helps too!