Schema-based Column Reordering for Dremel-encoded Data

Patrick Hansert and Sebastian Michel

BiDEDE@SIGMOD 23
Seattle, WA, USA

DBiS LAB
University of Kaiserslautern-Landau (RPTU)
Buffer and determining the sort order can be memory intensive!

→ Can we determine the sort order before any data is buffered, i.e., just with the schema?
Dremel Encoding

- Column-oriented storage of nested data
- Requires schema
- One column for each root-to-leaf path
- Encode NULL values as definition level (DL):
  Number of present optional steps

\{
  "B": {"C": 3, "D": 7, "E": {"F": 5, "G": 2}}
\}

\{
  "B": {"C": 3, "D": 7}
\}

\{
  "A": 4
\}

\[
\begin{array}{ccccc}
A & B.C & B.D & B.E.F & B.E.G \\
0 & 1 & 1 & 2 & 2 \\
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\textbf{1} Melnik et al. “Dremel: Interactive Analysis of Web-Scale Datasets”, VLDB 2010
Column Reordering

- Run-length encoding employed for definition levels & Boolean columns
- Sensitive to row order
  → Optimal order is NP-hard
- Heuristic: Sort lexicographically
  → Optimal column order is still NP-hard
- Increasing-cardinality heuristic:\n  Sort rows lexicographically, considering columns in the order of their increasing cardinality

<table>
<thead>
<tr>
<th>A</th>
<th>B.C</th>
<th>B.D</th>
<th>B.E.F</th>
<th>B.E.G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

→ 22 Runs

Schema: Block-based Sorting
Key Observation: Interdependence

For definition levels, the schema details:

- Min & Max value → cardinality
- Dependencies

Each DL determines the DL of all paths with a prefix of its optional nodes

But: the increasing-cardinality heuristic assumes data independence!
Blocks of Dependent Data

Consider data block each optional node affects:

- Columns it appears on
- Rows where parent is present
- Nested wrt. schema
- Repeated wrt. sort order

Upper bound: Each block at most doubles runs
Sort Blocks by Decreasing Size

- All block orderings adhere to tree order
- Nodes with many columns first → upper bound is minimized
- Deriving and ordering blocks in $\mathcal{O}(n \log n)$ using heapsort

\[
\begin{array}{cccc}
A & B.C & B.D & B.E.F & B.E.G \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 2 & 2 \\
1 & 1 & 1 & 2 & 2 \\
\end{array}
\]

→ 16 Runs
Evaluation
Experimental Setup

- Spark 3.2.0 & Parquet
- Extract schema and then sort in multiple ways:
  - Unsorted
  - Increasing-cardinality (schema)
  - Block-based (schema)
  - Exact

<table>
<thead>
<tr>
<th></th>
<th>Yelp</th>
<th>Steam</th>
<th>Tweets</th>
<th>GitHub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Nodes</td>
<td>61</td>
<td>54</td>
<td>1,101</td>
<td>705</td>
</tr>
<tr>
<td>Optional Nodes</td>
<td>49</td>
<td>18</td>
<td>748</td>
<td>134</td>
</tr>
<tr>
<td>Boolean Leaves</td>
<td>0</td>
<td>0</td>
<td>58</td>
<td>43</td>
</tr>
<tr>
<td>Document Count</td>
<td>150,346</td>
<td>74,821</td>
<td>79,219</td>
<td>218,939</td>
</tr>
</tbody>
</table>
Decreased Runcount

• Number of runs in definition level and Boolean columns
• Relevant metric for bitmap Indexes
• Block-based between factor 1.19 and 2.06 better
• Relative gains smaller when considering all columns
Compression Rate

- File size relative to increasing-cardinality with exact cardinalities
- Yelp and Tweets very close to Exact
- Steam and GitHub still within 10% → lower optional node count
- All results better than unsorted
- Block-based on average 0.53% better than increasing-cardinality
Conclusion

- Schemas provide **ample information** for column reordering
- Block-based improves runcount between **factor 1.19 and 2.04** over increasing-cardinality
  - Good for **bitmap indexes** over the structure
- **Comparable to compression rates** (within 10% or less) despite:
  - Use less information
  - Less computationally intensive
  - Fewer columns considered