Column Databases

This is a very preliminary and incomplete version
Column stores in 1985

- **Row store (N-ary Storage Model, NSM)**

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Age</th>
<th>House</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id1</td>
<td>John</td>
<td>32</td>
<td>HK245</td>
</tr>
<tr>
<td>Id3</td>
<td>Mary</td>
<td>33</td>
<td>HK324</td>
</tr>
<tr>
<td>Id4</td>
<td>John</td>
<td>45</td>
<td>HK245</td>
</tr>
</tbody>
</table>

- **Column store (Decomposition Storage Model, DSM)**

<table>
<thead>
<tr>
<th>Header</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Id1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Id3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Id4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Mary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>John</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>HK245</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>HK324</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>HK245</td>
</tr>
</tbody>
</table>
The reasons behind current trend

• Applicative:
  – Diffusion of analytical tasks

• Technological:
  – Widening the RAM – I/O time gap: must access disk less
  – Widening the seek-transfer gap: disk access must be sequential
  – RAM is bigger: I/O is not the only concern any more
  – Widening the cache – RAM time gap: must reduce the cache miss
  – Instruction pipelining: must reduce function call
  – SIMD instruction: make better use of SIMD parallelism
Column stores: pro and cons

• The advantages of column stores
  – Read the relevant columns only
  – Columns are easier to compress than rows: less I/O
  – Moving to tuple-at-a-time to block-at-a-time:
    • Better use of cache, pipelining, SIMD

• The problems with column stores
  – Tuple reconstruction requires a join
  – Tuple insertion requires many I/O operations
Column store performance

• From [Abadi et al.]
How to sort a column

• Columns stored in RID order:
  – Tuple reconstruction by sort-merge-like algorithm, parallel scanning
  – No need to store the RID

• Column stored in value order:
  – Excellent compression by Run-Length-Encoding
  – Range search with optimal efficiency
  – Needs to store the RID in some way
  – Tuple reconstruction by index-nested-loop like random access
  – The column looks like an index
## Storing a column: the RID

<table>
<thead>
<tr>
<th>Sorted by RID</th>
<th>RID-value</th>
<th>One RID per page</th>
<th>Implicit RID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1  20  2  50</td>
<td>1  20  50  45</td>
<td>20  50  45  20</td>
</tr>
<tr>
<td></td>
<td>3  45  4  20</td>
<td>20  45  20  34</td>
<td>45  20  34  50</td>
</tr>
<tr>
<td></td>
<td>5  45  6  20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7  34  8  50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sorted by value</th>
<th>Value-RIDS</th>
<th>Column + Join index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20  1  4  6</td>
<td>20  *3  34  *1</td>
</tr>
<tr>
<td></td>
<td>34  7  45  3</td>
<td>45  *2  50  *2</td>
</tr>
<tr>
<td></td>
<td>5  50  2  8</td>
<td></td>
</tr>
</tbody>
</table>

- **RID-value:** Sorted by RID.
- **Value-RIDS:** Sorted by value.
- **One RID per page:** Represents the RIDs stored in a way that each RID is on a separate page.
- **Implicit RID:** Does not explicitly store RIDs, relying on external mechanisms to identify records.
C-Store: projections

- A projection is a set of columns stored together and sorted according to one of them.
- Optimal for $\sigma_{k<1 \text{date}<k2} \pi_{\text{saleid}, \text{date}, \text{region}}(\text{Sales})$
Projections

• Storing ( saleid, date, region | date )

• Advantages of sorting \textit{date} by \textit{date}:
  – Better compression
  – Optimal I/O for range queries on \textit{date}
  – No cost for \textit{group-by} on \textit{date}

• Advantages of sorting \textit{saleid, region} by \textit{date}:
  – No cost for join with similarly sorted columns
  – Optimal I/O for range queries on \textit{date}
  – No cost for \textit{group-by} on \textit{date}
What is a projection

- It is not storing half-rows:
  - Every column is stored as a column
- A sorted column is like a compressed index
- A projection `(saleid, date, region | date)` is like a compressed index on `date` with the additional attributed `saleid` and `region`
- If a column is not RID-sorted, then RIDs should be stored
- Or, should they?
Join indexes
Vectorized execution

• Traditional dicotomy:
  – Tuple-at-a-type (tuple pipeline) execution: too many function calls, does not exploit array parallelism
  – Full execution of each operator: much faster on modern CPUs but intermediate results may not fit main memory
• Solution: each ‘next()’ call returns a block of values – typically, 1000 – that fits L1 cache size
• This reduces 1000 times the number of next() call
• A tight loop on a 1000 elements array can be SIMD parallelized and subject to parallel cache load
Compression

• Trading I/O – or even memory access - vs CPU
• Bit saving means more values in the registers
• Importance of fixed-width arrays
Compression algorithms

- RLE (Run Length Encoding):
  - a,a,a,c,c,d,c,c,c,c: (a,1,3), (c,4,2), (d,6,1), (c,7,4)

- Bit-Vector encoding:
  - a,b,a,a,b,c,b: a:1011000, b:0100101, c:0000010

- Dictionary encoding:
  - john,mary,john,john: (0:john)(1:mary) – 0,1,0,0
Compression algorithms

• Frame Of Reference
  – 1003,1017,1005: 1010 + (-7, +7, -5)

• Difference encoding
  – 1003,1017,1005: 1003, +14, -12

• Frequency partitioning:
  – Put similar values in the same page
  – Separate dictionary per page

• The patching technique
Operating on compressed data

• RLE encoded data can be operated upon without decompression
• Monotone dictionary encoding allows range queries
• Bit-Vectory encoding allows intersection???
Tuple reconstruction

- Early materialization: ScanTable+Project(A,B,C) -> ScanColumn(A)+ScanColumn(B)+Join+ScanColumn(C)+Join
- Late materialization: Column Algebras
- Multi-column blocks
Storage formats

- Pure columns
- Multi-column blocks
- Column groups
\[
\sum \left( \pi_{R.A} \left( \sigma_{5<A<10 \text{ and } 30<B<40} R \right) \text{ join} \pi_{R.C=S.B} \left( \sigma_{55<A<65} S \right) \right)
\]

Initial Status

<table>
<thead>
<tr>
<th>Relation R</th>
<th>Relation S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Rb</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>56</td>
<td>75</td>
</tr>
<tr>
<td>9</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>49</td>
</tr>
<tr>
<td>27</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>97</td>
</tr>
<tr>
<td>41</td>
<td>75</td>
</tr>
<tr>
<td>19</td>
<td>42</td>
</tr>
<tr>
<td>35</td>
<td>55</td>
</tr>
</tbody>
</table>

Query and Query Plan (MAL Algebra)

1. \text{inter1} = \text{select}(Ra, 5, 20)
2. \text{inter2} = \text{reconstruct}(Rb, \text{inter1})
3. \text{inter3} = \text{select}(\text{inter2}, 30, 40)
4. \text{join_input}_R = \text{reconstruct}(\text{Inter2}, \text{inter3})
5. \text{inter4} = \text{select}(Sa, 55, 65)
6. \text{inter5} = \text{reconstruct}(\text{Sb}, \text{inter4})
7. \text{join_input}_S = \text{reverse}(\text{inter5})
8. \text{join_res}_R_S = \text{join}(\text{join_input}_R, \text{join_input}_S)
9. \text{inter6} = \text{voidTail}(\text{join_res}_R_S)
10. \text{inter7} = \text{reconstruct}(Ra, \text{inter6})
11. \text{result} = \text{sum}(\text{inter7})

\[
\text{Inter1} = \sigma_{5<A<10} Ra
\]
\[
\text{Inter2} = \text{Rb \ semijoin} \sigma_{5<A<10} Ra
\]
\[
\text{Inter3} = \sigma_{30<B<40} \text{Inter2}
\]
\[
\text{join_input}_R = \text{Rc \ semijoin} \text{Inter3}
\]
\[
\text{sum}(\pi_{R.A} (\sigma_{5<A<10 \text{ and } 30<B<40} R \join_{R.C=S.B} \sigma_{55<A<65} S))
\]

**Initial Status**

<table>
<thead>
<tr>
<th>Relation R</th>
<th>Relation S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>Rb</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>56</td>
<td>75</td>
</tr>
<tr>
<td>9</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>49</td>
</tr>
<tr>
<td>27</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>97</td>
</tr>
<tr>
<td>41</td>
<td>75</td>
</tr>
<tr>
<td>19</td>
<td>42</td>
</tr>
<tr>
<td>35</td>
<td>55</td>
</tr>
</tbody>
</table>

**Query and Query Plan (MAL Algebra)**

1. \(\text{inter1} = \text{select}(Ra,5,20)\)
2. \(\text{inter2} = \text{reconstruct}(Rb,\text{inter1})\)
3. \(\text{inter3} = \text{select}(<20,30,40,\text{inter2})\)
4. \(\text{join_input}_R = \text{reconstruct}(Rc,\text{inter3})\)
5. \(\text{inter4} = \text{select}(Sa,55,65)\)
6. \(\text{inter5} = \text{reconstruct}(Sb,\text{inter4})\)
7. \(\text{join_input}_S = \text{reverse}(\text{inter5})\)
8. \(\text{join_res}_{R.S} = \text{join}(\text{join_input}_R,\text{join_input}_S)\)
9. \(\text{inter6} = \text{voidTail}(\text{join_res}_{R.S})\)
10. \(\text{inter7} = \text{reconstruct}(Ra,\text{inter6})\)
11. \(\text{result} = \text{sum}(\text{inter7})\)

**Intermediate Results**

- \(\text{inter4} = \sigma_{55<A<65} Sa\)
- \(\text{inter5} = Rb \text{ semijoin } \text{inter4}\)
- \(\text{join_input}_S = \pi_{S.B} \sigma_{55<A<65 \text{ and } 30<B<40} S\)
8. \( \text{join}_{\text{res}} = (\pi_{R.C} \sigma_{5<A<10 \text{ and } 30<B<40} R \text{ join } \pi_{S.B} \sigma_{55<A<65} S) \)
9. Inter6 = \( \pi_{R.*} (\sigma_{5<A<10 \text{ and } 30<B<40} R \text{ join } R.C=S.B \sigma_{55<A<65} S) \)
10. Inter7 = \( \pi_{R.A} (\sigma_{5<A<10 \text{ and } 30<B<40} R \text{ join } R.C=S.B \sigma_{55<A<65} S) \)
11. Result = sum(\( \pi_{R.A} (\sigma_{5<A<10 \text{ and } 30<B<40} R \text{ join } R.C=S.B \sigma_{55<A<65} S) \))
Column join

- Column stores avoid indexes, hence:
  - Hash Join
  - Sort-merge
- Column Join typically fits main memo
- Column Join: value columns -> position pairs
The Jive join

- R(Rid,A,B,...) and S(Sid,C,D,...)
- Want to compute $\pi_{ABCD} (R \bowtie_{A=C} S)$
- We have the JoinIndex $JI = \pi_{Rlid,Sd} (R \bowtie_{A=C} S)$
- We have $NPag < B^2$, but $NPag$ only includes pages in the two projected semijoins:
  - $NPag(\pi_{AB} (R \bowtie_{A=C} S)) + NPag (\pi_{CD} (R \bowtie_{A=C} S)) + Npag(JI)$
The Jive join

- We partition projected $\pi_{\text{Sid,C,D}} S$ in $K$ files $S_i$ ($K = B/2$)
- Partition is logical!
- We scan $R$ and $JI$ in parallel. We use $2*K$ buffers to create $2*K$ files that contain:
  - $JR_i = \pi_{\text{AB}} (R \land_{A=C} S_i)$ (the $K$ output files)
  - $JS_{\text{Sid}}_i = \pi_{\text{Sid}} (R \land_{A=C} S_i)$ (the $K$ temporary files)
- We are almost done: we must just create, for each $JR_i$, a corresponding $JS_i$ that contains $\pi_{\text{CD}} (R \land_{A=C} S_i)$ in the same order
The Jive join

• For each i, we load the whole JSIdᵢ in memory, and sort a copy in SId order
• We read the corresponding i-partition of $\pi_{SId,C,D} S$, skipping the useless blocks
• We scan the unsorted copy of JSId. For each Sid in JSId, we put the corresponding record of $Sᵢ$ (which is in memory) in JSᵢ
• Now we have:
  - $JRᵢ = \pi_{AB} (R \bowtie_{A=C} Sᵢ)$
  - $JSᵢ = \pi_{CD} (R \bowtie_{A=C} Sᵢ)$
Memory needs

• Phase 1: $2^K$ buffers, hence $K = B/2$ ($y = m/2$)
• Phase 2:
  - Size of JSIdi: $|JI|/K$
  - Plus size of $\pi_{S_{ld,C,D}} S$ – Sld may be implicit: $|\pi S|/K$
• Hence
  - $(|JI| + |\pi S|)/K < B$
  - $2*|JSIdi| + 2*|\pi S| < B*B$
Group by and aggregation

• Group by is typically hash-based
• Aggregation (sum, count...) takes advantage of columnar layout in order to perform a tight loop over a cache-sized array