Integrating Compression and Execution in Column-Oriented Database Systems

D. Abadi, S. Madden, M. Ferreira (SIGMOD, 2006)
Motivation
Motivation

• Increase DBMS performance

• Compression: I/O vs. CPU cycles

• Column stores better suited for compression
The Paper in a Nutshell

• Apply compression algorithms to column store

• Explore trade-offs depending on
  • Data set
  • Query types

• Introduce architecture to work on compressed data

• Show importance of ordered data
  • Including secondary and tertiary sort orders
C-Store Recap
C-Store

- Data stored in columns, not rows
- **Tables** represented by set of **projections**
  - Every column in at least one projection
  - Projections can have **secondary/tertiary** sort order

\[
\begin{align*}
&\text{(suppkey, quantity | suppkey)} \\
&(\text{shipdate, retflag, suppkey | retflag, suppkey)} \\
&(\text{shipdate, quantity, retflag, suppkey | quantity)} \\
&(\text{shipdate, quantity, retflag, suppkey | shipdate, quantity, retflag)}
\end{align*}
\]

- Storing columns in **multiple** projections with **different** sort order
C-Store

- Projections related by *join indices*

<table>
<thead>
<tr>
<th>Position</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>3</td>
<td>blue</td>
</tr>
<tr>
<td>4</td>
<td>green</td>
</tr>
<tr>
<td>5</td>
<td>orange</td>
</tr>
</tbody>
</table>

$\times$

<table>
<thead>
<tr>
<th>Position</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>orange</td>
</tr>
<tr>
<td>2</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>black</td>
</tr>
</tbody>
</table>

$=$

<table>
<thead>
<tr>
<th>Position</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Compression Schemes
Null Suppression

- Replace leading zeros/blanks with description
  - Example

  \[
  \begin{align*}
  \text{00000000 00100010 10010110} & \quad \rightarrow \quad \text{01 } \text{00100010 10010110} \\
  \text{00001011 11111111 10110100} & \quad \rightarrow \quad \text{10 } \text{00001011 11111111 10110100} \\
  \text{00000000 00000000 00000011} & \quad \rightarrow \quad \text{00 } \text{00000011} \\
  \text{00000000 00000000 00101001} & \quad \rightarrow \quad \text{00 } \text{00101001}
  \end{align*}
  \]

- Byte-align:

  \[
  \begin{align*}
  \text{01100000 00100010 10010110} & \quad \text{00001011 11111111 10110100} \quad \text{00000011} \quad \text{00101001}
  \end{align*}
  \]
Dictionary Encoding

• Encode frequent patterns
  • Example: 32 color values → 5-bit encoding
    
    red: 00000
    blue: 00001
    yellow: 00010

• Choose 3-value/2-byte encoding
  
  red yellow blue → X000000011000001

• Encoding dependent on CPU cache size
  • 2 bytes per entry, $32^3$ entries → 0.5 Mbit
Dictionary Encoding

- Encode frequent patterns
  - Example: 32 color values → 5-bit encoding

    red 00000
    blue 00001
    yellow 00010

- Single value parsing

  \[ \begin{align*}
  x000000001000001 & \quad 000000000111111 \quad >> \quad 0 &= 00001 = \text{blue} \\
  x000000001000001 & \quad 000011111000000 \quad >> \quad 5 &= 00010 = \text{yellow} \\
  x000000001000001 & \quad 111100000000000 \quad >> \quad 10 &= 00000 = \text{red}
  \end{align*} \]
Run-Length Encoding (RLE)

- Compress runs of the same value
  - Create tuple (value, start position, run-length)

<table>
<thead>
<tr>
<th>Position</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>54</td>
<td>Waterloo</td>
</tr>
<tr>
<td>55</td>
<td>Waterloo</td>
</tr>
<tr>
<td>56</td>
<td>Waterloo</td>
</tr>
<tr>
<td>57</td>
<td>Waterloo</td>
</tr>
<tr>
<td>58</td>
<td>Kitchener</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

..., (Waterloo, 54, 4), (Kitchener, 58, ...), ...

Bit-Vector Encoding

• Map occurrences per value

<table>
<thead>
<tr>
<th>Position</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>blue</td>
</tr>
<tr>
<td>3</td>
<td>yellow</td>
</tr>
<tr>
<td>4</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>red</td>
</tr>
<tr>
<td>6</td>
<td>yellow</td>
</tr>
<tr>
<td>7</td>
<td>blue</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

string for value
red: 0001100...
blue: 1100001...
yellow: 0010010...
“Heavyweight” Compression Schemes

- Lempel-Ziv algorithm (**LZ77**, used in gzip)

<table>
<thead>
<tr>
<th>Position</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Position</th>
<th>Match</th>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“Heavyweight” Compression Schemes

• Lempel-Ziv algorithm (**LZ77**, used in gzip)

<table>
<thead>
<tr>
<th>Position</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Position</th>
<th>Match</th>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>A</td>
<td>A(0,0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
“Heavyweight” Compression Schemes

- Lempel-Ziv algorithm (LZ77, used in gzip)

<table>
<thead>
<tr>
<th>Position</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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Position</th>
<th>Match</th>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>A</td>
<td>A(0,0)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>A</td>
<td>AB</td>
<td>B(1,1)</td>
</tr>
</tbody>
</table>
“Heavyweight” Compression Schemes

- Lempel-Ziv algorithm (**LZ77**, used in gzip)

<table>
<thead>
<tr>
<th>Position</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Position</th>
<th>Match</th>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>A</td>
<td>A(0,0)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>A</td>
<td>AB</td>
<td>B(1,1)</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-</td>
<td>C</td>
<td>C(0,0)</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>B</td>
<td>BB</td>
<td>B(2,1)</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>AB</td>
<td>ABC</td>
<td>C(5,2)</td>
</tr>
</tbody>
</table>
Extension of C-Store
Compression-Aware Optimization

**NLJoin(Predicate q, Column c1, Column c2)**

**Completely uncompressed data**

**Uncompressed data joined with RLE compression**

**Uncompressed data joined with bit-vector compression**

- If c1 is not compressed and c2 is not compressed
  - For each value valc1 with position i in c1 do
  - For each value valc2 with position j in c2 do
  - If q(valc1, valc2) then:
  - Output-left: (i), output-right: (j)

- If c1 is not compressed and c2 is RLE compressed
  - For each value valc1 with position i in c1 do
  - For each triple t with val v, startpos j and runlen k in c2
  - If q(valc1, v) then:
    - Output-left: t,
    - Output-right: (j ... j+k-1)

- If c1 is not compressed and c2 is bit-vector compressed
  - For each value valc1 with position i in c1 do
  - For each value valc2 with bitstring b in c2 do
    // Assume that there are num '1's in b
  - If q(valc1, valc2) then:
    - Output-left: new RLE triple (NULL, i, num),
    - Output-right: b

Etc. Etc. For every possible combination of encoding types
Compression-Aware Optimization

- Naively, for $n$ compression algorithms need
  - $n$ operators with single input
  - $n^2$ operators with two inputs

- Solution: Abstract away optimization relevant properties
Compression-Aware Optimization

- **Solution:** Abstract away optimization relevant properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Optimization</th>
</tr>
</thead>
</table>
| One value, Contiguous Positions | **Aggregation:** If both the group-by and aggregate input blocks are of this type, then the aggregate input block can be aggregated with one operation (e.g. if size was 8 and aggregation was sum, result is 8*value)  
**Join:** Perform optimization shown in the second if statement in Figure 1 (works in general, not just for RLE). |
| One value, Pos. Non-contiguous | **Join:** Perform optimization shown in the third if statement in Figure 1 (works in general, not just for bit-vector compression). |
| One value | **Aggregation Group-By clause:** The position list of the value can be used to probe the data source for the aggregate column so that only values relevant to the group by clause are read in. |
| Sorted | **Max or Min Aggregation:** Finding the maximum or minimum value in a sorted block is a single operation  
**Join** Finding a value within a block can be done via binary search. |

<table>
<thead>
<tr>
<th>Encoding Type</th>
<th>Sorted?</th>
<th>1 value?</th>
<th>Pos. contig.?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLE</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Bit-string</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Null Supp.</td>
<td>no/yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Lempel-Ziv</td>
<td>no/yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dictionary</td>
<td>no/yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Uncompressed</td>
<td>no/yes</td>
<td>no</td>
<td>no/yes</td>
</tr>
</tbody>
</table>
Compression Block

- Buffer of compressed data
- API:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Iterator Access</th>
<th>Block Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>isOneValue()</td>
<td>getNext()</td>
<td>getSize()</td>
</tr>
<tr>
<td>isValueSorted()</td>
<td>asArray()</td>
<td>getStartValue()</td>
</tr>
<tr>
<td>isPosContig()</td>
<td></td>
<td>getEndPosition()</td>
</tr>
</tbody>
</table>
Data Source

- **Interface** between query plan and storage manager

- Compression specific knowledge
  - Serves up **compression blocks**

- **Scan** operator
  - Executes (equality) predicates directly against compressed data
Example

\begin{itemize}
\item \texttt{COUNT(Column c1)}
\item \hspace{1em} \texttt{b = get next compressed block from c1}
\item \hspace{1em} \textbf{while} \texttt{b is not null}
\item \hspace{2em} \textbf{if} \texttt{b.isOneValue()}
\item \hspace{3em} \texttt{x = fetch current count for b.getStartVal()}
\item \hspace{3em} \texttt{x = x + b.getSize()}
\item \hspace{2em} \textbf{else}
\item \hspace{3em} \texttt{a = b.isArray()}
\item \hspace{4em} \textbf{for each} \texttt{i in a}
\item \hspace{5em} \texttt{x = fetch current count for i}
\item \hspace{5em} \texttt{x = x + 1}
\item \hspace{2em} \texttt{b = get next compressed block from c1}
\end{itemize}
Experimental Results
Compressed Data

(a) Run size 50  
No Compression  
LZ Compression  
Null-suppression  
RLE Compression  
Dictionary compression  
Bit-vector compression

(b) Run size 1000
Eager Decompression

(a) run size 50

(b) run size 1000
Operating on Compressed Data

run size 50

run size 1000

CPU contention
Higher column cardinalities

29

run size 50

run size 1000
Generated Data vs. TPC-H Data

(time, retflag) [15]
(suppkey, linenum) [86]
(suppkey, retflag) [200]
(shipdate, quantity) [475]
(shipdate, retflag, quantity) [314]
Compression-Dependent Query Performance

(a) Position filter on RLE column

(b) Predicate on RLE column

SELECT COL1, COUNT(*) FROM CSTORE_PROJ1 WHERE PREDICATE(COL2) GROUP BY COL1
Conclusion
Conclusion

• Beneficial to operate on compressed data

• Need to take into account both I/O and CPU

• Database design should be aware of compression schemes
  • Take advantage of data locality
  • Use projections with multiple sort orders
  • Use low cardinality columns as primary sort order
Providing a Decision Helper

Are there moderate length runs? (Avg. size > ~ 4)

Yes
- Compress column with RLE

No
- Is the number of distinct values > ~50,000?

Yes
- Is the column likely to be used in a position-contiguous manner?

Yes
- Use Dictionary Encoding

No
- Use Bit-Vector Encoding

Is the column likely to be used in a position-contiguous manner?

No
- Is the number of distinct values > ~50?

Yes
- Use Dictionary Encoding

No
- Use Bit-Vector Encoding

Does the data exhibit good locality?

Yes
- Use LZ

No
- Don't Compress
Future

• Automate decision of optimal compression algorithm

• Monitor query load and change compression dynamically
Critique
Critique

- Easy to read and understand
- Great basis for presentation
- Detailed experimentation
- Property abstraction great idea
- Very little research into compression impacts

- Calculation error for dictionary size (bits mixed up with bytes)
- Cherry-picked TPC-H data
Questions & Discussion