Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Abhijeet Mohapatra, Michael Genesereth
# Motivation

<table>
<thead>
<tr>
<th>CrimeBosses</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vincent</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td>John</td>
<td>Dillinger</td>
<td>IL</td>
</tr>
<tr>
<td>Michael</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td>Bonnie</td>
<td>Parker</td>
<td>IL</td>
</tr>
<tr>
<td>John</td>
<td>Gotti</td>
<td>NY</td>
</tr>
<tr>
<td>Clyde</td>
<td>Barrow</td>
<td>IL</td>
</tr>
<tr>
<td>Vito</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td>Al</td>
<td>Capone</td>
<td>IL</td>
</tr>
<tr>
<td>Sonny</td>
<td>Corleone</td>
<td>NY</td>
</tr>
</tbody>
</table>
# Incrementally Maintaining Run-length Encoded Attributes in Column Stores

## Motivation

CrimeBosses sorted by State, First Name

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al</td>
<td>Capone</td>
<td>IL</td>
</tr>
<tr>
<td>Bonnie</td>
<td>Parker</td>
<td>IL</td>
</tr>
<tr>
<td>Clyde</td>
<td>Barrow</td>
<td>IL</td>
</tr>
<tr>
<td>John</td>
<td>Dillinger</td>
<td>IL</td>
</tr>
<tr>
<td>John</td>
<td>Gotti</td>
<td>NY</td>
</tr>
<tr>
<td>Michael</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td>Sonny</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td>Vincent</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td>Vito</td>
<td>Corleone</td>
<td>NY</td>
</tr>
</tbody>
</table>

### Results

### Extensions
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

<table>
<thead>
<tr>
<th>Motivation</th>
<th>Count Indexes</th>
<th>Results</th>
<th>Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrimeBosses sorted by State, First Name and vertically partitioned</td>
<td>First Name</td>
<td>Last Name</td>
<td>State</td>
</tr>
<tr>
<td>= Runs exposed</td>
<td>Al</td>
<td>Capone</td>
<td>IL</td>
</tr>
<tr>
<td></td>
<td>Bonnie</td>
<td>Parker</td>
<td>IL</td>
</tr>
<tr>
<td></td>
<td>Clyde</td>
<td>Barrow</td>
<td>IL</td>
</tr>
<tr>
<td></td>
<td>John</td>
<td>Dillinger</td>
<td>IL</td>
</tr>
<tr>
<td></td>
<td>John</td>
<td>Gotti</td>
<td>NY</td>
</tr>
<tr>
<td></td>
<td>Michael</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td></td>
<td>Sonny</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td></td>
<td>Vincent</td>
<td>Corleone</td>
<td>NY</td>
</tr>
<tr>
<td></td>
<td>Vito</td>
<td>Corleone</td>
<td>NY</td>
</tr>
</tbody>
</table>
Motivation

Count Indexes

Results

Extensions

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Vertical Partitioning

Better Compression

Faster Reads

Set of tuples

Sequence of tuples

Decompression and Tuple Alignment

Sorting

RLE

Bitmap Encoding
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

<table>
<thead>
<tr>
<th>Value</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL</td>
<td>4</td>
</tr>
<tr>
<td>NY</td>
<td>9</td>
</tr>
</tbody>
</table>

Count Indexes

<table>
<thead>
<tr>
<th>Value</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL</td>
<td>4</td>
</tr>
<tr>
<td>NY</td>
<td>9</td>
</tr>
</tbody>
</table>

Results

<table>
<thead>
<tr>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL</td>
<td>4</td>
</tr>
<tr>
<td>NY</td>
<td>5</td>
</tr>
</tbody>
</table>

Extensions

Updates: slow
Look Ups: fast

Updates: fast
Look Ups: slow
Choice of representation trades-off Look Up and Update costs.

<table>
<thead>
<tr>
<th>Representation Scheme</th>
<th>Cost of Operation</th>
<th>Cost of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Look Up</td>
<td>Update</td>
</tr>
<tr>
<td>Count-based</td>
<td>(O(n))</td>
<td>(O(n))*</td>
</tr>
<tr>
<td>Offset-based</td>
<td>(O(\log n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

\(*\) If offset is known at priori then \(O(1)\) time is required. Otherwise \(O(n)\) time is required.
Incementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

**Workarounds:** Amortize the cost of batch updates.

**Key Idea:** Differential store buffers the updates which are later merged using a table scan.

**Pros:** Spend $O(k + n)$ time on $k$ updates than one update.

**Cons:** Decompression and Re-compression overheads.

Update cost $\propto$ total number of tuples.
Choice of representation trades-off Look Up and Update costs.

<table>
<thead>
<tr>
<th>Representation Scheme</th>
<th>Cost of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Look Up</td>
</tr>
<tr>
<td>Count-based</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Offset-based</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Workarounds</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

$n =$ number of runs in the sequence
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

**Count Indexes**

Motivation

How to optimally trade-off the update and the lookup costs?

Input Sequence \[\xrightarrow{\text{incremental maintenance}}\] Output Sequence

Index Structure \[\xrightarrow{\text{incremental maintenance}}\] Index Structure

Our solution
A *count index* is a tree over run lengths.

- Each node stores the sum of the values of its children.
- Leaf nodes store the run lengths.
- Root node stores the size of the sequence.
Count Indexes optimally trade-off the Look Up and Update costs.

<table>
<thead>
<tr>
<th>Representation Scheme</th>
<th>Cost of Operation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Look Up</td>
<td>Update</td>
</tr>
<tr>
<td>Count-based</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Offset-based</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Workarounds</td>
<td>O(log n)</td>
<td>k updates: O(k + n)</td>
</tr>
<tr>
<td>Count Indexes</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td></td>
<td>k updates: O(k + log n)</td>
<td></td>
</tr>
</tbody>
</table>
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Our solution

Input Sequence (n runs) → Count Index

\[
\text{update one run: } O(\log n) \text{ time}
\]

\[
\text{update a sequence of } k \text{ runs: } O(k + \log n) \text{ time}
\]

\[
\text{update } k \text{ runs: } O(k + n) \text{ time}
\]

Output Sequence → Count Index

Motivation  Count Indexes  Results  Extensions
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Results

Extensions

Count Index over the sequence: a, a, b, b, b, a, a, a, b, b
Suppose we want to look up the value at the 8th position in the sequence: \texttt{a, a, b, b, b, a, a, a, b, b}
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Look Ups

The left child of root contains nodes up to the 5th position. Hence, search the right child for the offset (8 - 5) = 3.
Look Ups

The left child now contains nodes up to the 3rd position. Hence our answer lies in the left child.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Results

Extensions

Look Ups

Diagram:

```
   10
   /\  \\
  5 /  \ 5
  /    /\  /
 2  a  3 b 5
  / \   / \  /
 a  b a  b a
```

Number of lookups: 3
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Results

Extensions

Look Ups

Time Complexity = O(h)
Suppose we delete the first value (‘a’) and the last value ‘b’ in the sequence: a, a, b, b, b, a, a, a, b, b
Motivation

Count Indexes

Results

Extensions

Deletions

Case 1: At each step we only update the count at each node.
Case 1: At each step we only update the count at each node.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Deletions

Results

Extensions

**Case 1:** At each step we only update the count at each node.
Suppose we delete the last value ‘b’ from the sequence: a, b, b, b, a, a, a, b
Case 2: If we delete a paired-up node that has no unpaired neighbors, we update the ancestors’ counts.
Case 2: If we delete a paired-up node that has no unpaired neighbors, we update the ancestors’ counts.

Long branches may result from deletions
Case 2: If we delete a paired-up node that has no unpaired neighbors, we update the ancestors’ counts.
Case 2: If we delete a paired-up node that has no unpaired neighbors, we update the ancestors’ counts.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Results

Extensions

Suppose we delete the first value ‘a’ from the sequence: a, b, b, b, a, a, a
Case 2: If the deleted node or its sibling has an unpaired neighbor, we pair up the sibling and the neighbor.
Case 2: If the deleted node or its sibling has an unpaired neighbor, we pair up the sibling and the neighbor.
Case 2: If the deleted node or its sibling has an unpaired neighbor, we pair up the sibling and the neighbor.
Deletions

At each level of the count index, we either delete a node, update its value or pair its sibling with a neighbor.

Therefore, time complexity of deletions = $O(h)$ where $h$ is the height of count index.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation Count Indexes Results Extensions

Bulk Inserts

Input sequence: a, a, b, b, b, a, a, a, b, b
Suppose we want to insert: a, a, a, a, c, c at 6th position
Step 1: Build a count index over the $k$-run sequence to be inserted. This takes $O(k)$ time.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Bulk Inserts

Step 2: Merge the count index from Step 1 with the count index.

Results

Extensions

h

h1

4

a

6

2

2

a

c

3

3

b

a

5

5

2

b

10
Motivation

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Count Indexes

Bulk Inserts

Step 2: Merge the count index from Step 1 with the count index.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Step 2: Merge the count index from Step 1 with the count index.

Bulk Inserts
Motivation

Count Indexes

Results

Extensions

Bulk Inserts

Step 2: Merge the count index from Step 1 with the count index.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation  Count Indexes  Results  Extensions

Bulk Inserts

Time Complexity of Inserting $k$ runs $= O(k + h - h_1)$
$= O(k + h)$
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

1. Count Indexes are balanced i.e. height $h = O(\log n)$ where $n$ is number of leaves.

2. Incrementally maintaining $n$ runs takes $\Omega(\log n)$ time. Hence count indexes are optimal.
About \(\frac{2}{3}\)rd of the nodes are paired-up at each level.

\[\therefore \text{ Height } h \text{ of the count index } = O(\log n).\]

<table>
<thead>
<tr>
<th>Operation over (n) runs</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look ups</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Inserting / Deleting a run</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Inserting a sequence of (k) runs</td>
<td>(O(k + \log n))</td>
</tr>
</tbody>
</table>
1. Count Indexes are balanced i.e. height $h = O(\log n)$ where $n$ is number of leaves.

2. Incrementally maintaining $n$ runs takes $\Omega(\log n)$ time. Hence count indexes are optimal.
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Results

Extensions

- To Bit-Map Encoded sequences

Bitmap ‘IL’ : 110000110

Bitmap ‘NY’ : 001111001

few distinct values
multiple runs

State

IL
IL
NY
NY
NY
NY
IL
IL
NY
To Block-Oriented Stores (fan-out > 2)

Deletions: Merge a half-filled node with its neighbor.

Insertions: Split a full node into two halves before inserting.
Motivation

Count Indexes

Results

Extensions

Summary

• Out-of-order updates degrade compression.

• In-place updates on a sequence with \( n \) runs require \( O(n) \) time.

• Workarounds amortize the cost of batch updates but need to decompress and re-compress the sequence.

• Count indexes can incrementally maintain and look up a sequence of \( n \) runs in \( O(\log n) \) time (which is optimal).
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Results

Extensions

Thank You!
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation  Count Indexes  Results  Extensions

End of Presentation