Efficiently Updating Column Stores

Abhijeet Mohapatra, Michael R. Genesereth

Column Stores

Tables are vertically partitioned into columns typically w.r.t a *sort order*

Vertical Partitioning ➡ Better Compression ➡ More data fetched per I/O ➡ Faster Reads

crimeBosses ordered by **(State, Gender)**

NAME
B. Parker
A. Capone
J. Dillinger
M. Corleone
S. Corleone
B. Edgar

STATE
IL
IL
IL
NY
NY
WA

GENDER
Female
Male
Male
Male
Male
Male

crimeBosses (Name, State, Gender)

NAME	STATE	GENDER
B. Edgar	WA	Male
J. Dillinger	IL	Male
A. Capone	IL	Male
B. Parker	IL	Female
M. Corleone	NY	Male
S. Corleone	NY	Male

Column Stores: Compression

- Sorting and vertical partitioning exposes '*runs*' which can be compressed using **Run-length Encoding**

GENDER
Female
Male
Male
Male
Male
Male

run #1 : Female

run #2 : Male, Male, Male, Male, Male

GENDER	START	END
Female	1	2
Male	2	6

<value, offset>

GENDER	COUNT
Female	1
Male	5

<value, count>

Bottleneck: In-Place Updates

Compression schemes are **order-sensitive**

Out-of-order Updates ➡ Bad Compression% ➡ Inefficient Reads

- An in-place update takes **O(n)** time where n is the number of runs
- **Workaround**: Amortize the cost of updates in bulk

  Key Idea: Maintain updated tuples separately and merge with the 'read' or stable store through scans

  Upside: O(n) time on **k** updates

  Downsides:

  (a) Decompression and Recompression overheads
  (b) Scans are still linear in the number of runs

Trading off Updates and Lookups

- Runs can be represented using *offsets* or *counts*

Property \ Representation	**Offset based**	**Count based**
Monotonic	Yes	No
Look up Cost	O(log n) using Binary Search	O(n)
Update Cost	O(n)	O(1) if offset is known

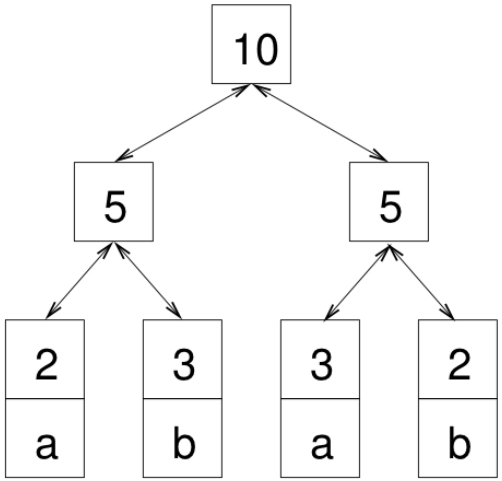
- If offsets are used, we have to update subsequent runs as well (this takes **O(n)** time)

- If offsets are not used, traditional indexes (variants of BTrees) cannot be used to support efficient lookups

Solution: Count Indexes

- Binary tree on the counts (or *run-lengths*)
  - Value at each node = Sum of its children's values

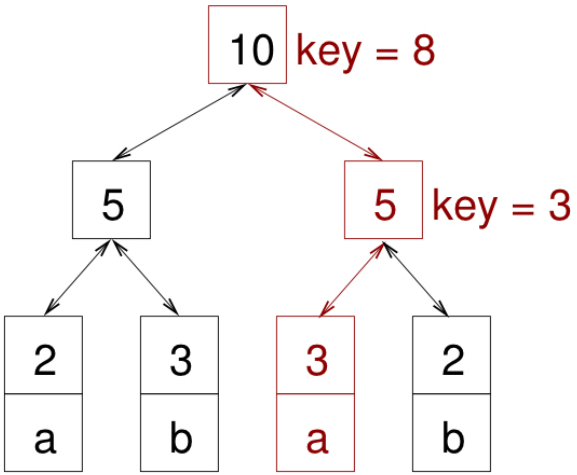
- Consider a sequence **S**: a, a, b, b, b, a, a, a, b, b



Count index over **S**

Lookups using Count Indexes

Suppose we want to look up the 8^{th} value in the sequence **S**



The cost of lookup is **O(h)** where **h** is the height of the count index

Properties of Count Indexes over a sequence of 'n' runs

PROPERTY	**DESCRIPTION**
Creation of a count index	**O(n) time**
Count indexed are balanced	Height **h = O(log n)**
Time Complexity of Updates and Lookups	**O(h) time**
Time Complexity of inserting a sequence of **k** runs	**O(k + log n) time**

Extensions

- Supporting efficient updates over Bitmap Encoded Columns
- Block-based storage systems: Generalizing count indexes to a fan-out > 2

Other applications:

Fast in-place updates over unsorted arrays.

Summary

- Column stores:
  - support fast reads by efficiently compressing data
  - in-place updates take linear time

- Existing techniques amortize the cost of bulk updates

  Downsides are:

  (a) the **decompression** and **recompression** costs

  (b) the **linear** cost of a merge **scan**

- We present an indexing technique *count indexes* which supports lookup and updates on run-length encoded columns in time that is **logarithmic** in the number of the runs