Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Abhijeet Mohapatra, Michael Genesereth

STANFORD UNIVERSITY

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes

Results

Extensions

CrimeBosses

First Name	Last Name	State
Vincent	Corleone	NY
John	Dillinger	IL
Michael	Corleone	NY
Bonnie	Parker	IL
John	Gotti	NY
Clyde	Barrow	IL
Vito	Corleone	NY
Al	Capone	IL
Sonny	Corleone	NY

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

CrimeBosses

sorted by

State, First Name

First Name	Last Name	State
Al	Capone	IL
Bonnie	Parker	IL
Clyde	Barrow	IL
John	Dillinger	IL
John	Gotti	NY
Michael	Corleone	NY
Sonny	Corleone	NY
Vincent	Corleone	NY
Vito	Corleone	NY

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation

Count Indexes | Results | Extensions

CrimeBosses

sorted by

State, First Name

and

vertically partitioned

=

Runs exposed

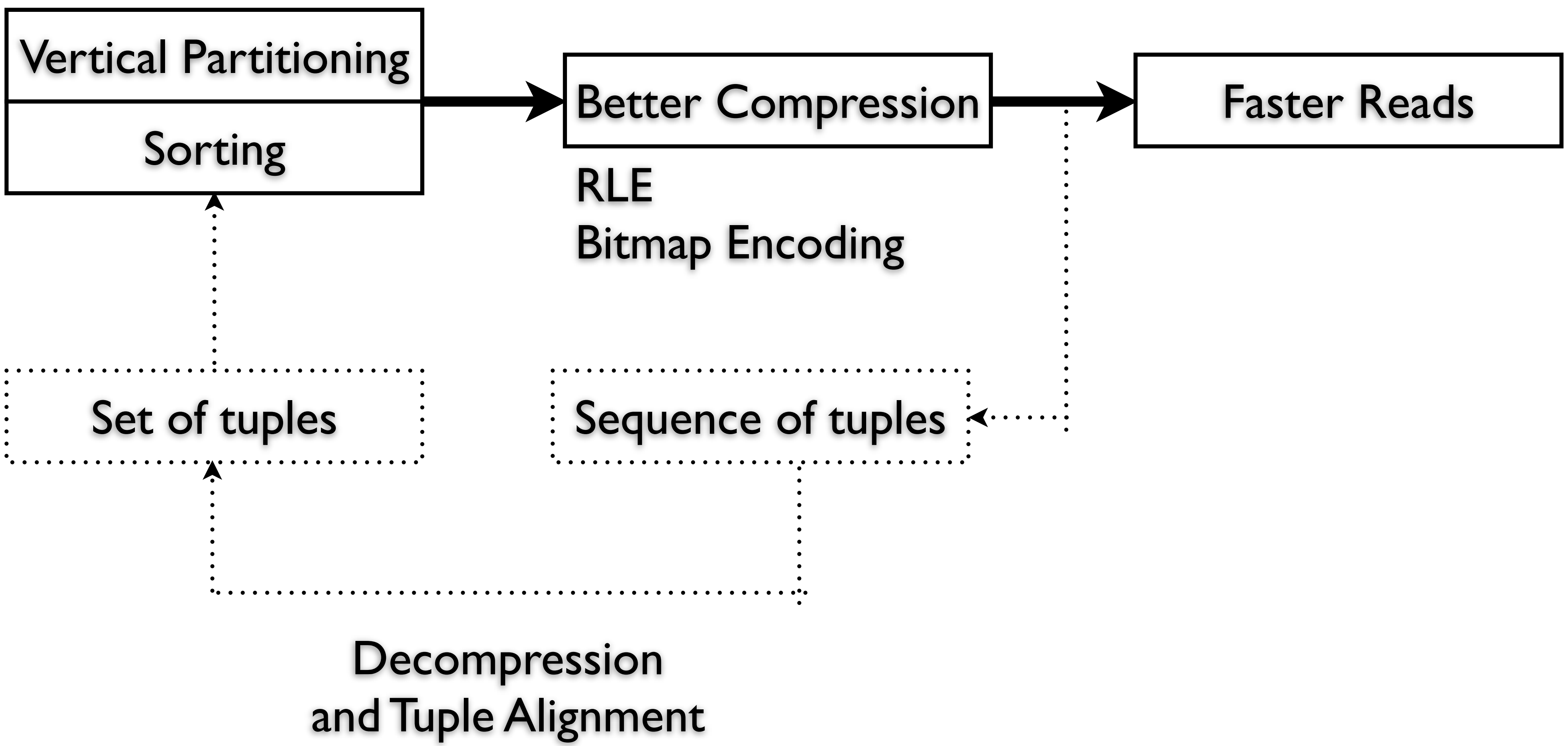
First Name
Al
Bonnie
Clyde
John
John
Michael
Sonny
Vincent
Vito

Last Name
Capone
Parker
Barrow
Dillinger
Gotti
Corleone
Corleone
Corleone
Corleone

State
IL
IL
IL
IL
NY
NY
NY
NY
NY

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores



STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

⟨value, offset⟩

Value	Offset
IL	4
NY	9

Updates: slow
Look Ups: fast

State
IL
IL
IL
IL
NY
NY
NY
NY
NY

⟨value, count⟩

Value	Count
IL	4
NY	5

Updates: fast
Look Ups: slow

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Choice of representation trades-off Look Up and Update costs.

n = number of runs in the sequence		
Representation Scheme	Cost of Operation	
	Look Up	Update
Count-based	$O(n)$	$O(n)^*$
Offset-based	$O(\log n)$	$O(n)$

* If offset is known at priori then $O(1)$ time is required. Otherwise $O(n)$ time is required.

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.

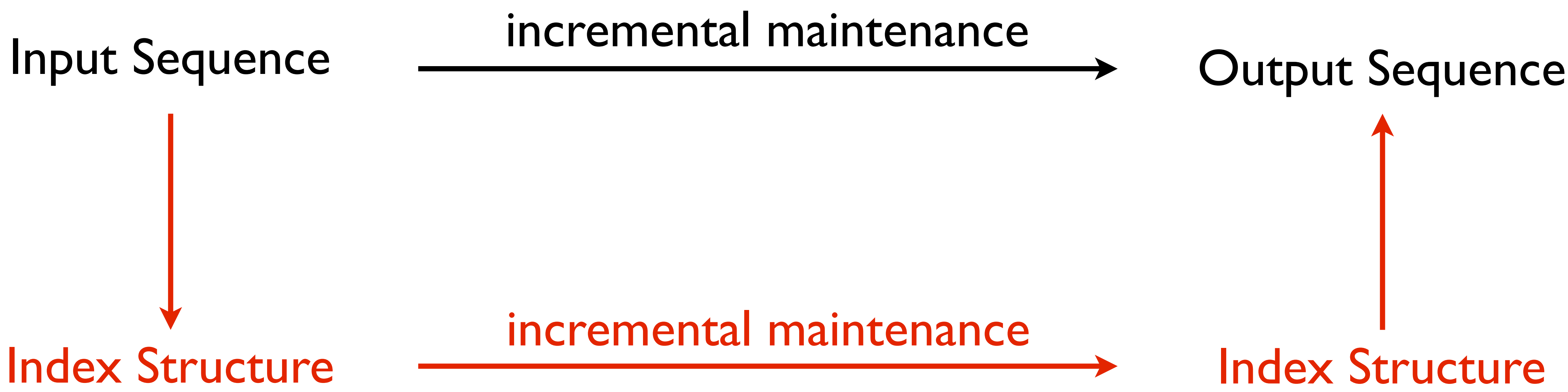
Update cost ∝ total number of tuples.

Choice of representation trades-off Look Up and Update costs.

n = number of runs in the sequence		
Representation Scheme	Cost of Operation	
	Look Up	Update
Count-based	$O(n)$	$O(n)$
Offset-based	$O(\log n)$	$O(n)$
Workarounds	$O(\log n)$	k updates: $O(k + n)$

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

How to optimally trade-off the update and the look up costs?



Our solution

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

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.

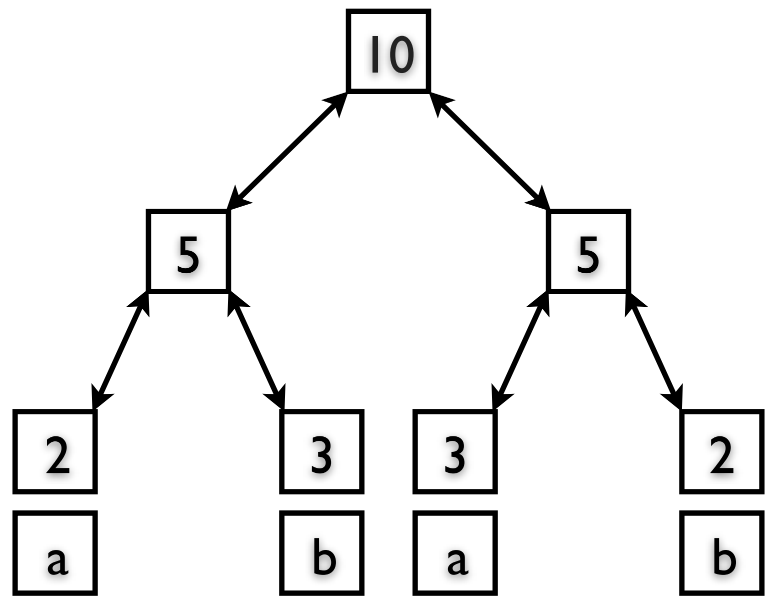
n = number of runs in the sequence		
Representation Scheme	Cost of Operation	
	Look Up	Update
Count-based	$O(n)$	$O(n)$
Offset-based	$O(\log n)$	$O(n)$
Workarounds	$O(\log n)$	k updates: $O(k + n)$
Count Indexes	$O(\log n)$	$O(\log n)$
		k updates: $O(k + \log n)$

Our solution

Input Sequence (n runs) → update k runs: $O(k + n)$ time → Output Sequence

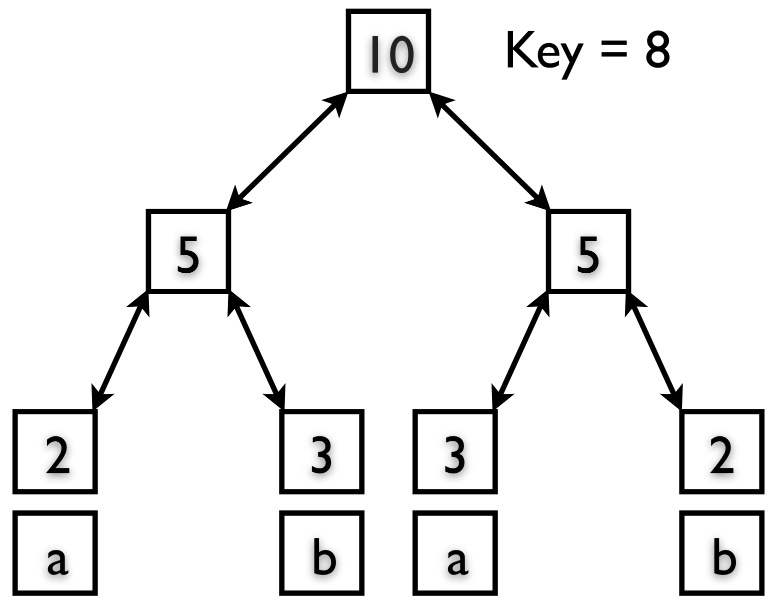
↓ Count Index → update one run: $O(\log n)$ time; update a sequence of k runs: $O(k + \log n)$ time → Count Index ↑

Incrementally Maintaining Run-length Encoded Attributes in Column Stores



Count Index over the sequence: a, a, b, b, b, a, a, a, b, b

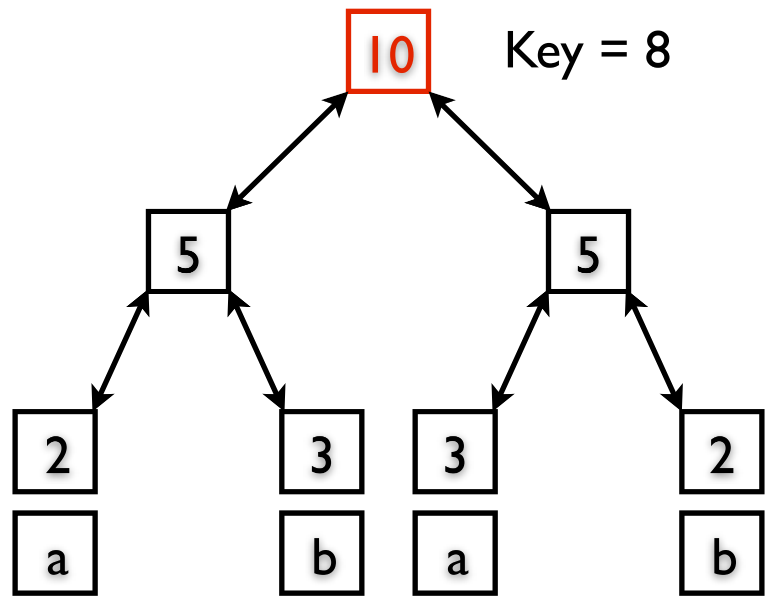
Look Ups



Suppose we want to look up the value at the 8th position in the sequence: a, a, b, b, b, a, a, a, b, b

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Look Ups



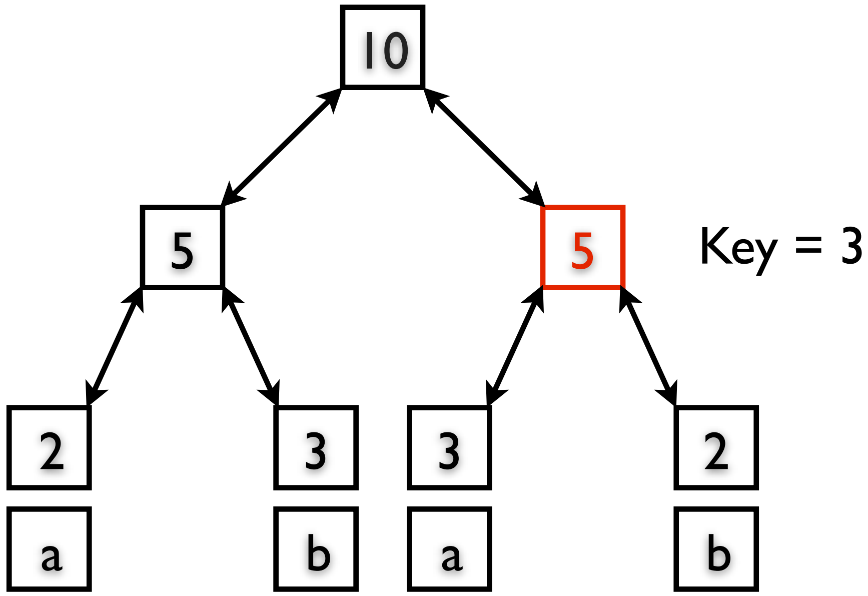
The left child of root contains nodes up to the 5th position. Hence, search the right child for the offset (8 - 5) = 3.

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation **Count Indexes** Results Extensions

Look Ups



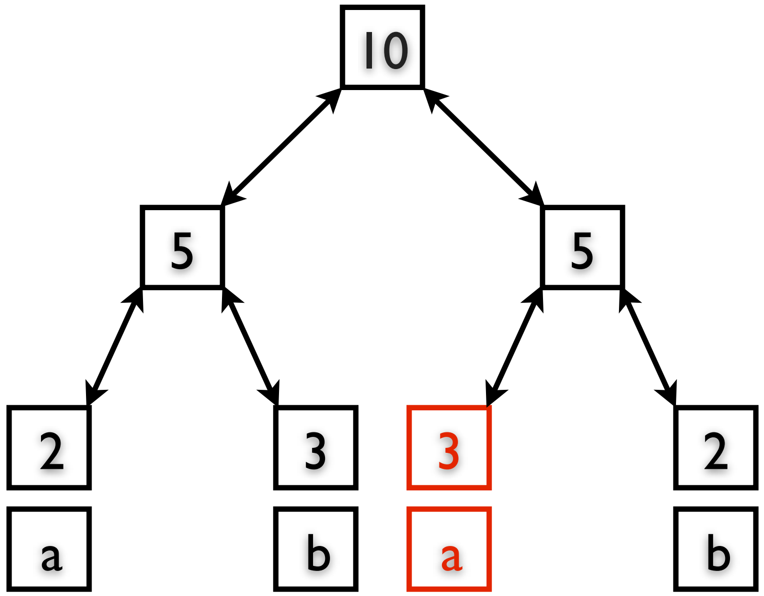
The left child now contains nodes up to the 3rd position. Hence our answer lies in the left child.

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

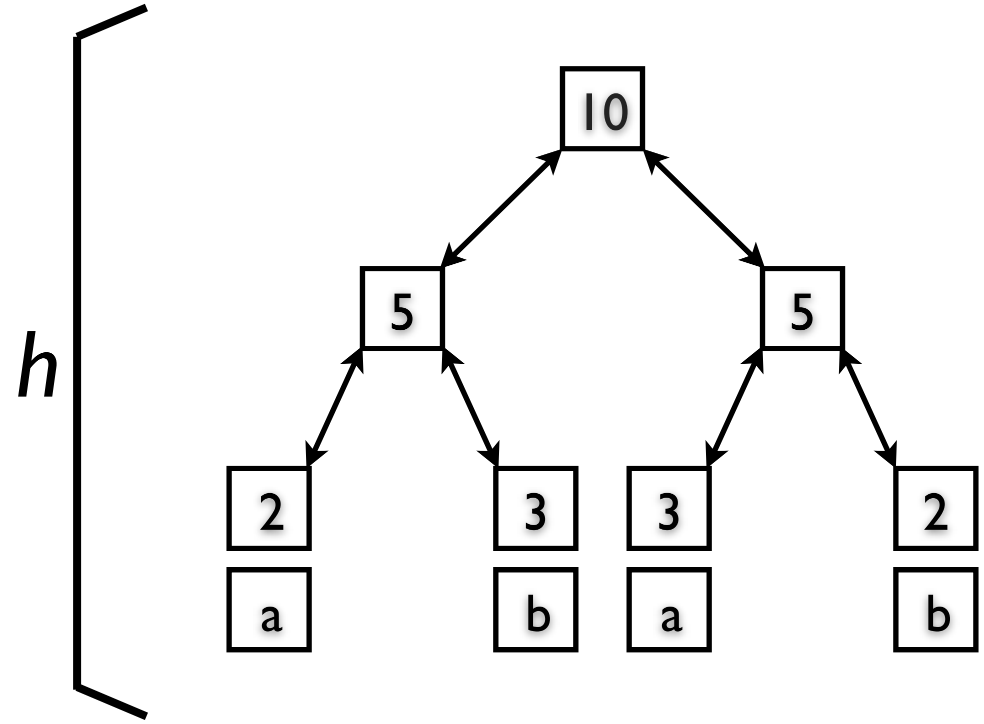
Motivation | **Count Indexes** | Results | Extensions

Look Ups



Incrementally Maintaining Run-length Encoded Attributes in Column Stores

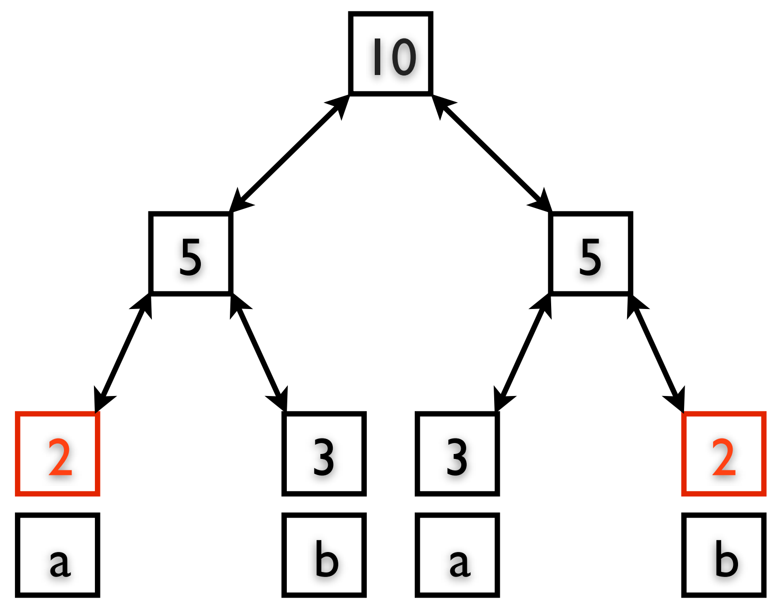
Look Ups



Time Complexity = O(h)

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

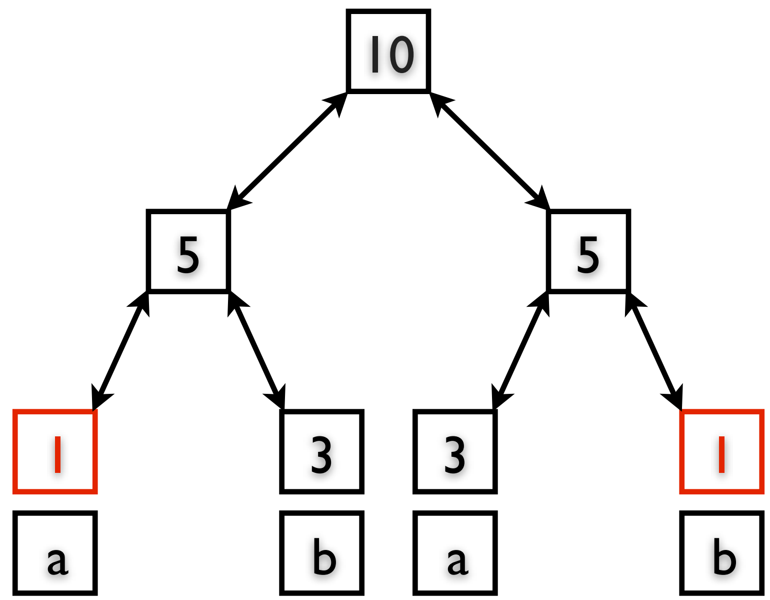
Deletions



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

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Deletions



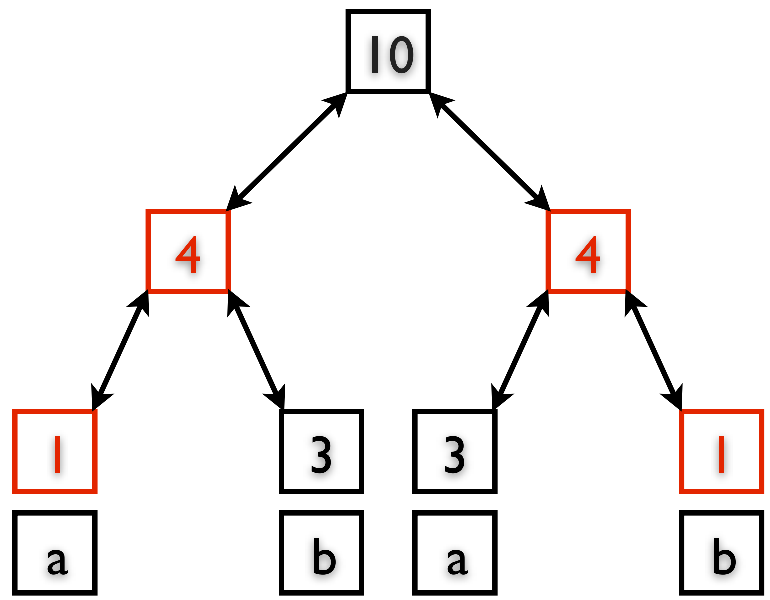
Case 1: At each step we only update the count at each node.

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation | **Count Indexes** | Results | Extensions

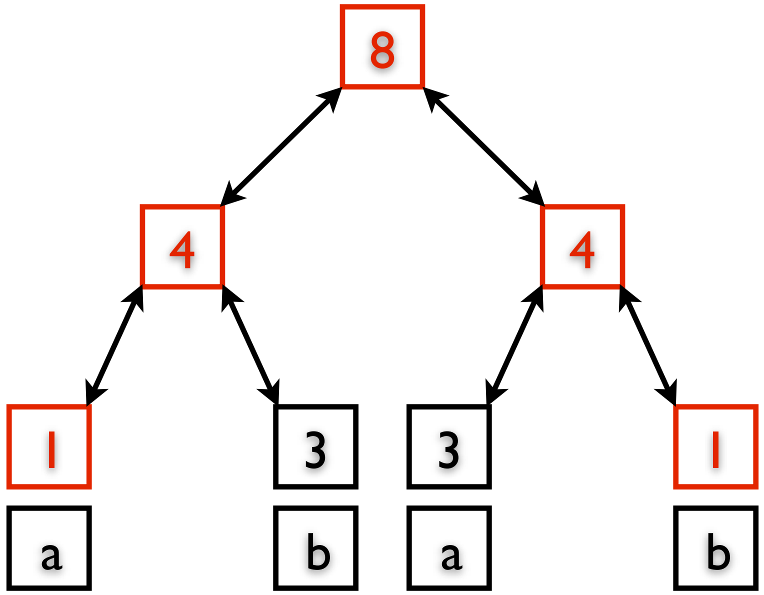
Deletions



Case 1: At each step we only update the count at each node.

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Deletions



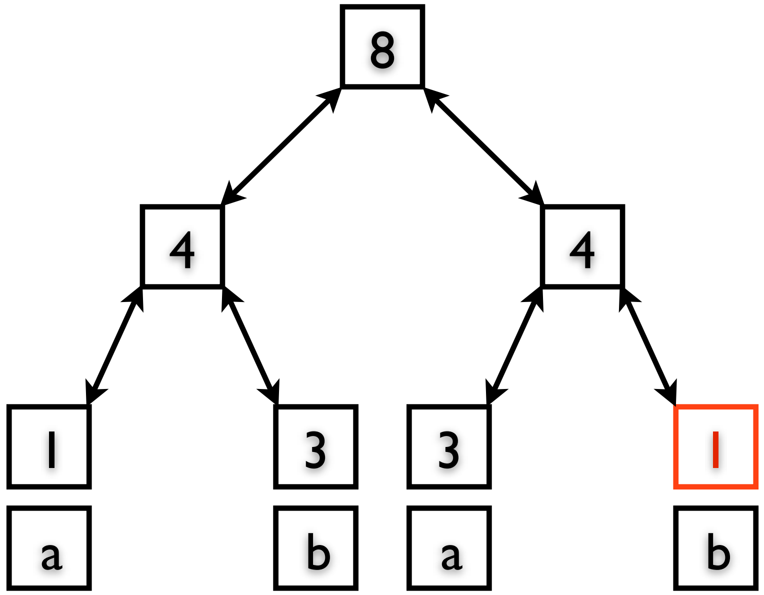
Case 1: At each step we only update the count at each node.

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation | **Count Indexes** | Results | Extensions

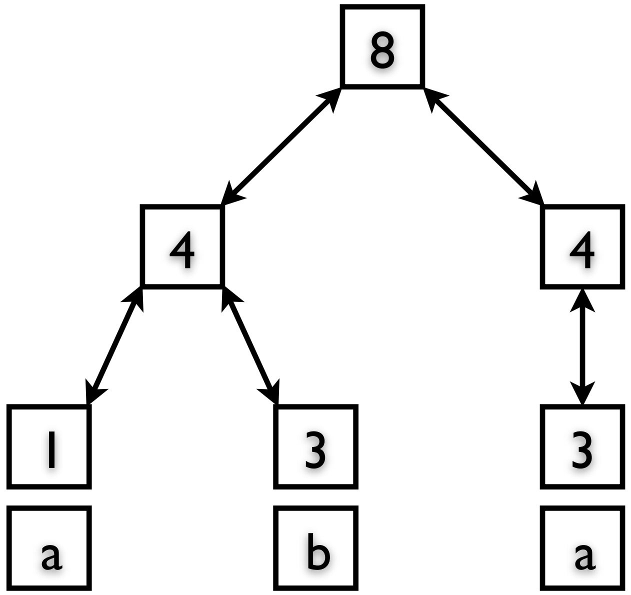
Deletions



Suppose we delete the last value 'b' from the sequence:
a, b, b, b, a, a, a, b

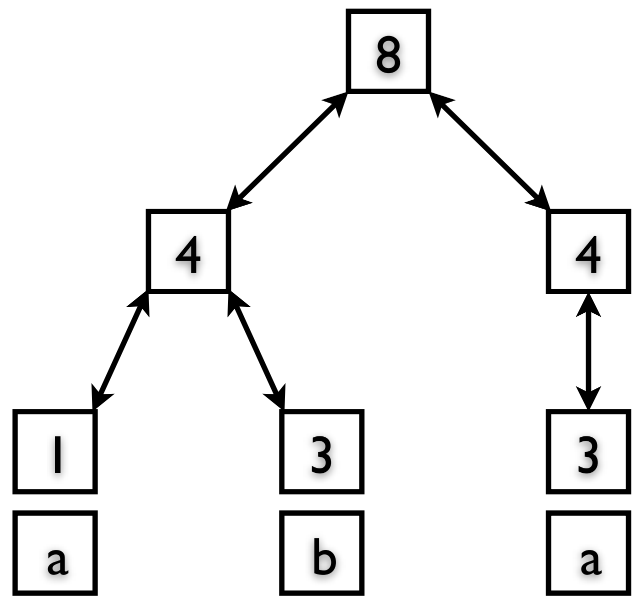
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Deletions



Case 2: If we delete a paired-up node that has no unpaired neighbors, we update the ancestors' counts.

Deletions



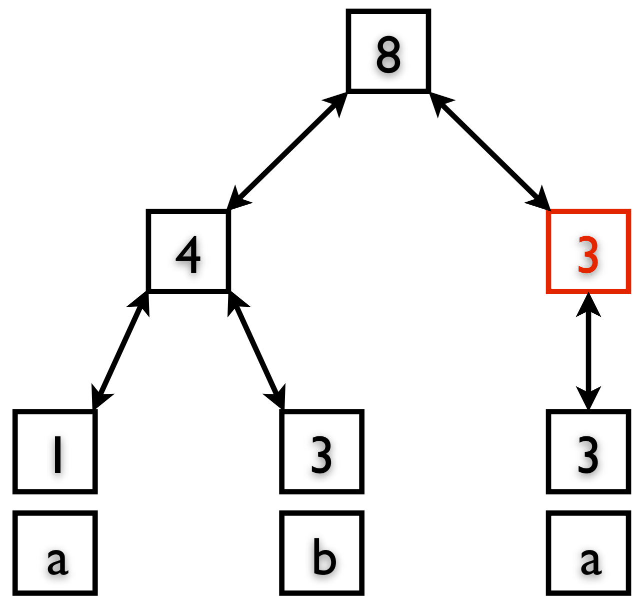
Long branches may result from deletions

Case 2: If we delete a paired-up node that has no unpaired neighbors, we update the ancestors' counts.

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

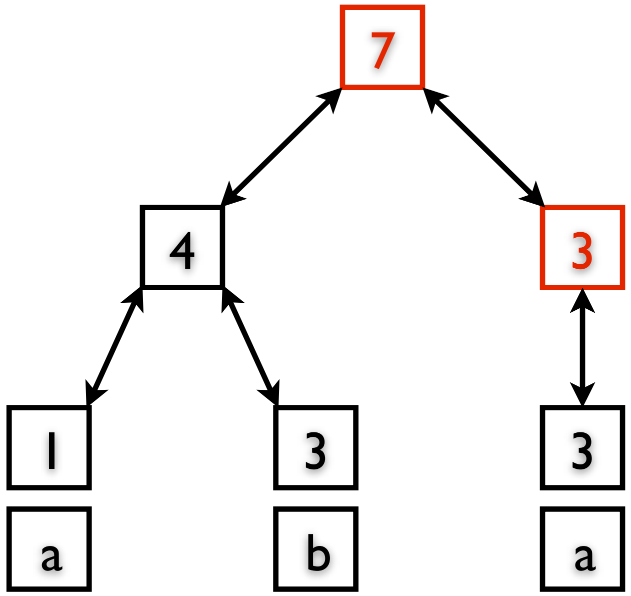
Deletions



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

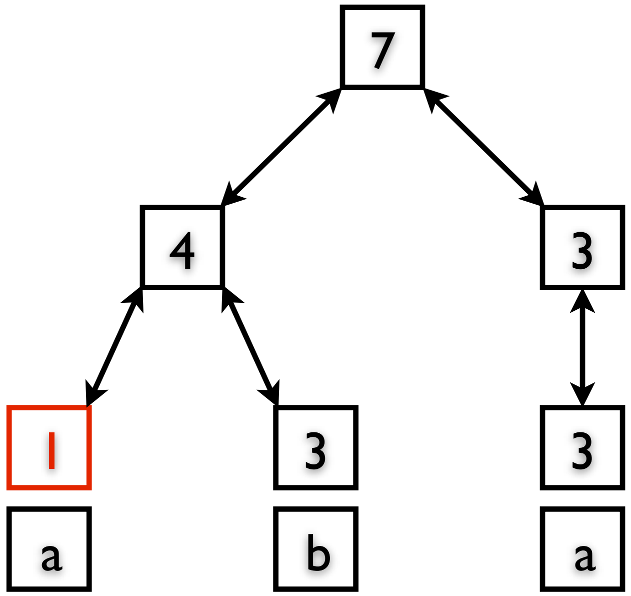
Deletions



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

Deletions



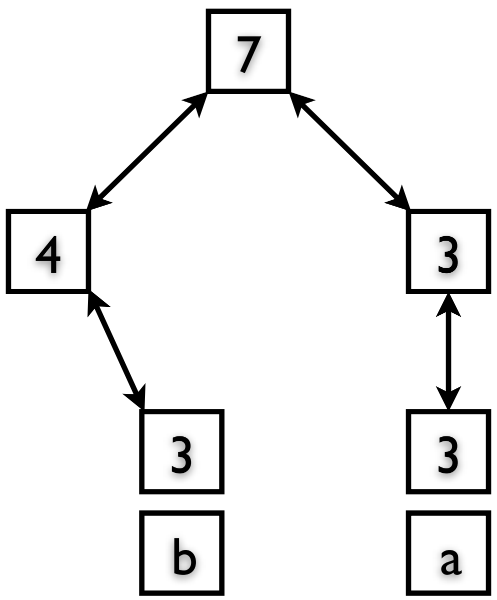
Suppose we delete the first value 'a' from the sequence: a, b, b, b, a, a, a

STANFORD UNIVERSITY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Motivation | **Count Indexes** | Results | Extensions

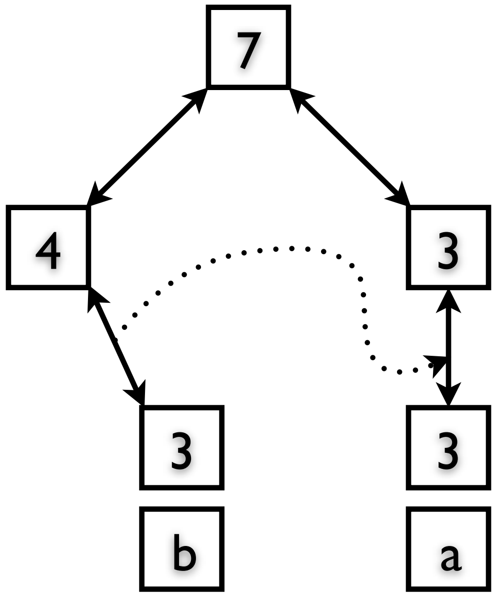
Deletions



Case 2: If the deleted node or its sibling has an unpaired neighbor, we pair up the sibling and the neighbor.

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

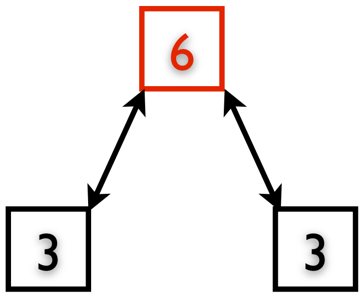
Deletions



Case 2: If the deleted node or its sibling has an unpaired neighbor, we pair up the sibling and the neighbor.

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Deletions



Case 2: If the deleted node or its sibling has an unpaired neighbor, we pair up the sibling and the neighbor.

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

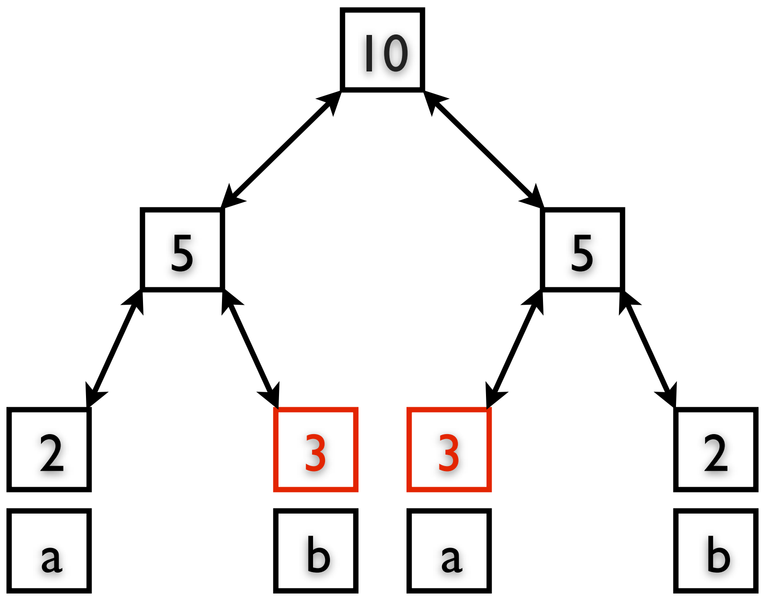
Deletions

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

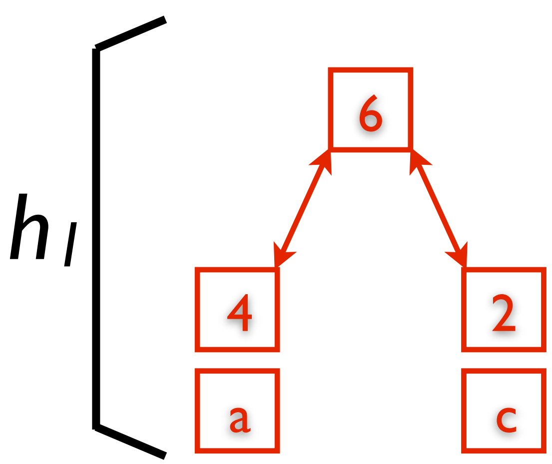
Bulk Inserts



Input sequence: a, a, b, b, b, a, a, a, b, b

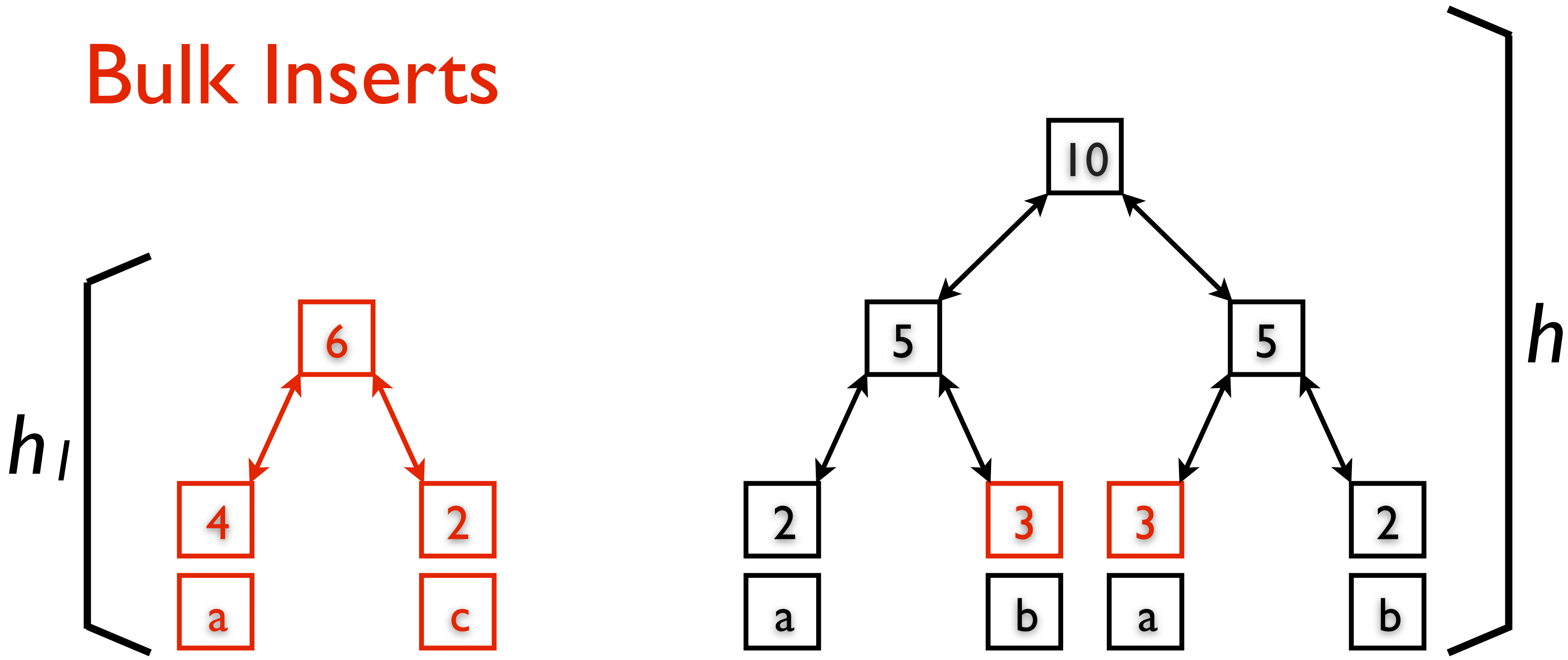
Suppose we want to insert: a, a, a, a, c, c at 6th position

Bulk Inserts



Step 1: Build a count index over the *k*-run sequence to be inserted. This takes O(*k*) time.

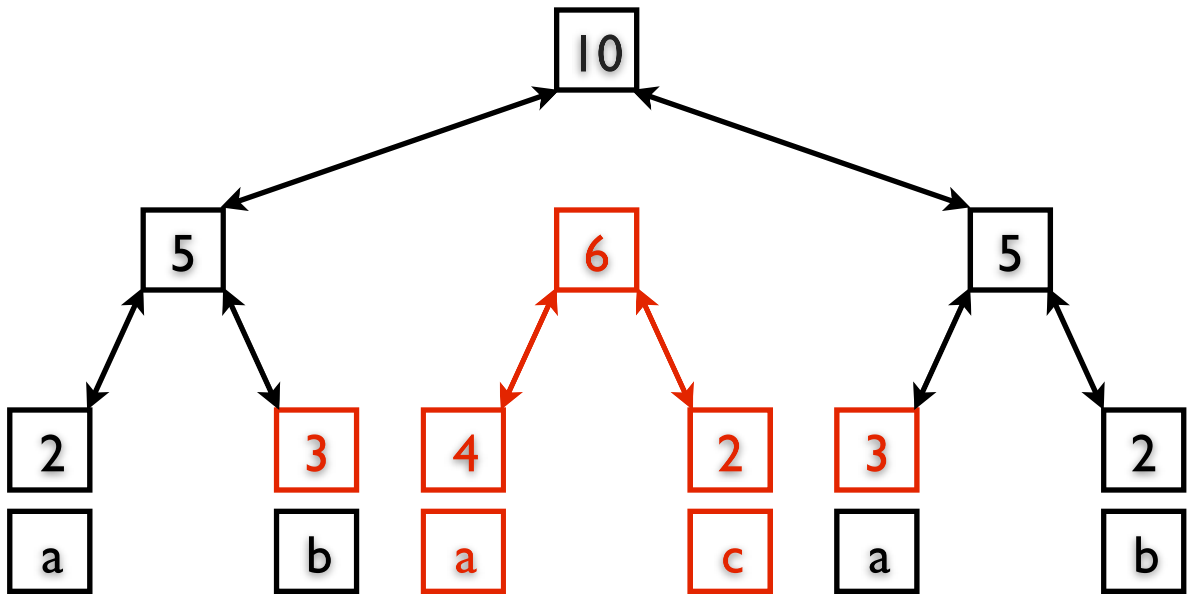
Bulk Inserts



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

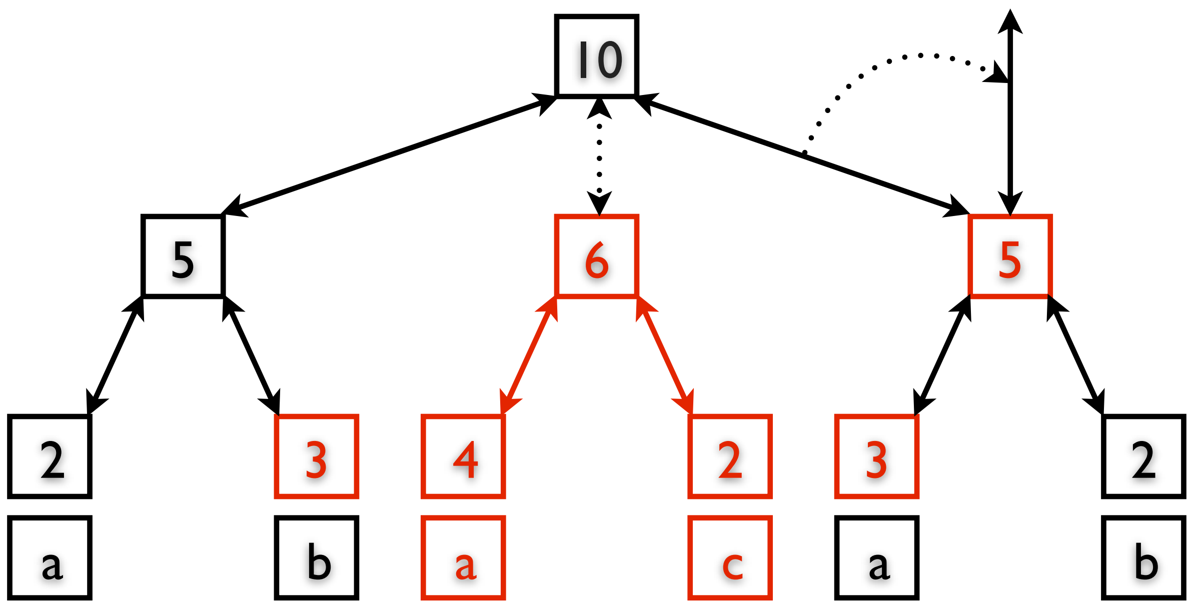
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Bulk Inserts



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

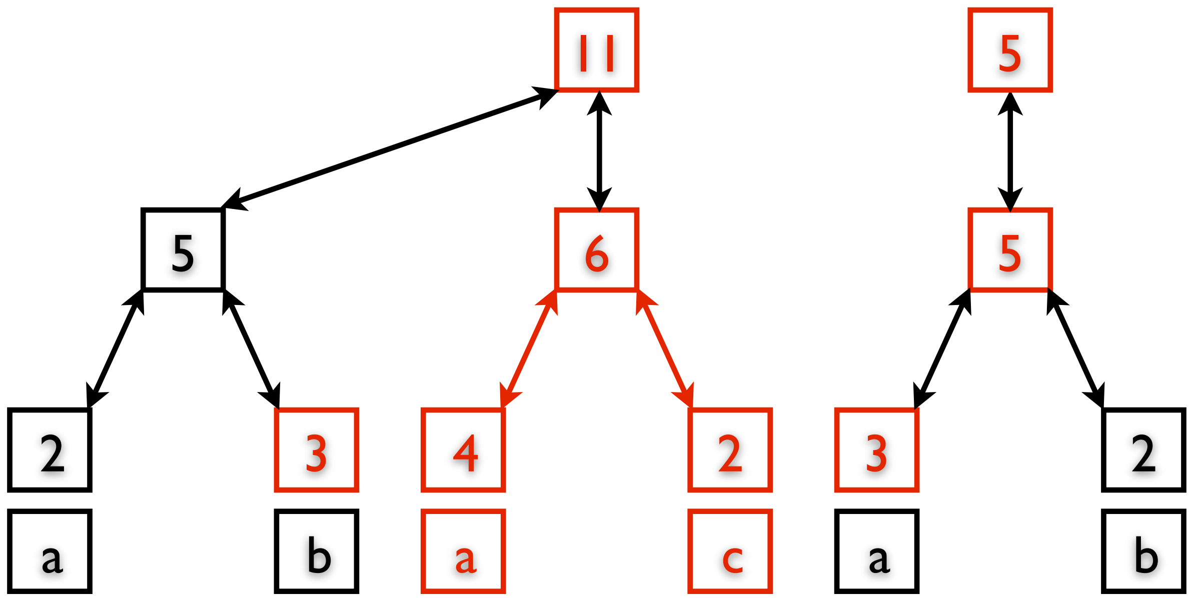
Bulk Inserts



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

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

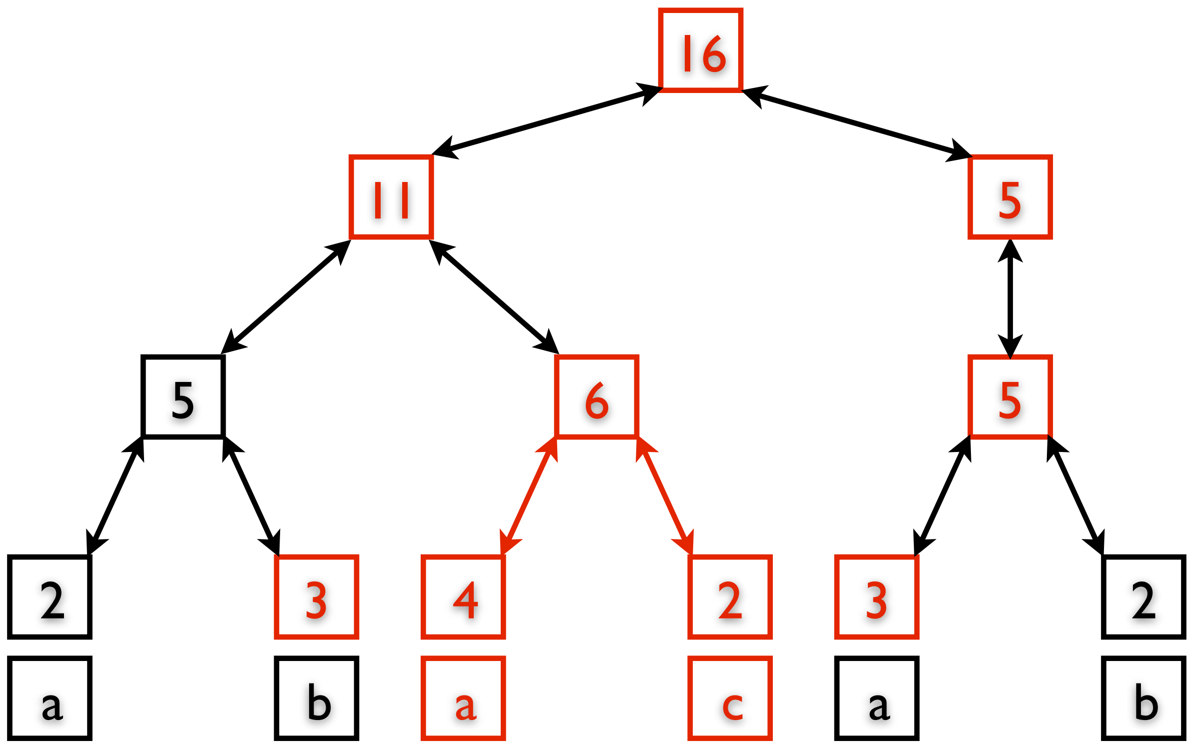
Bulk Inserts



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

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Bulk Inserts



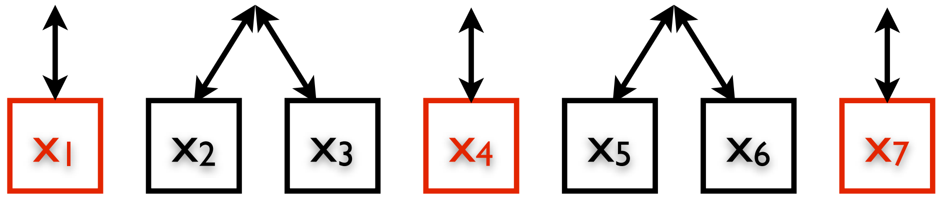
Time Complexity of Inserting k runs = $O(k + h - h_l)$
$= O(k + h)$

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.

Balancing Count Indexes



About 2/3rd of the nodes are paired-up at each level.

∴ Height h of the count index = $O(\log n)$.

⇒

Operation over n runs	Time
Look ups	$O(\log n)$
Inserting / Deleting a run	$O(\log n)$
Inserting a sequence of k runs	$O(k + \log n)$

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

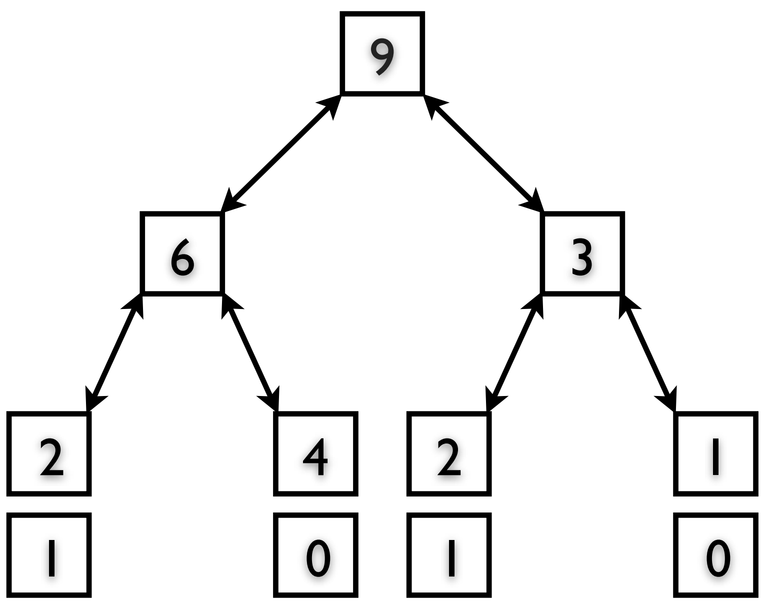
1. Count Indexes are balanced i.e. height $h = O(\log n)$ where n is number of leaves.

2. Incrementally maintaining n runs takes $\Omega(\log n)$ time. Hence count indexes are optimal.

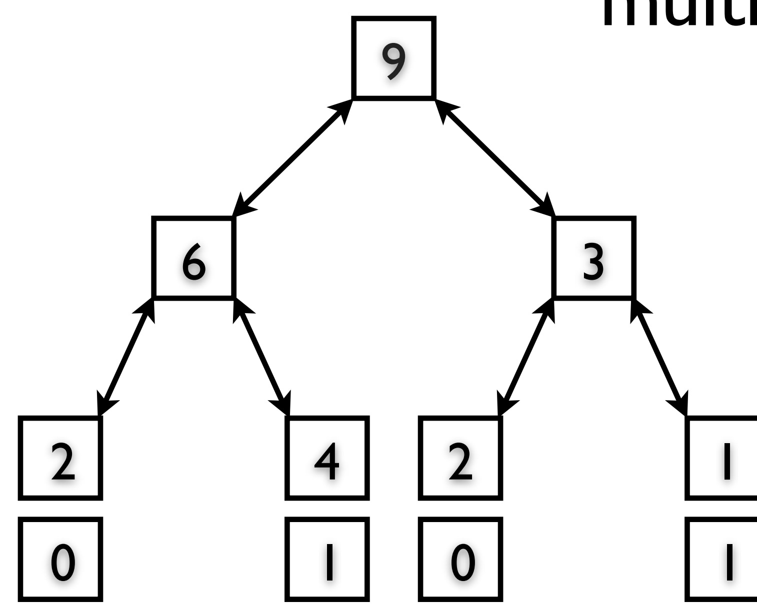
Incrementally Maintaining Run-length Encoded Attributes in Column Stores

- To Bit-Map Encoded sequences

few distinct values
multiple runs



Bitmap 'IL' : 110000110

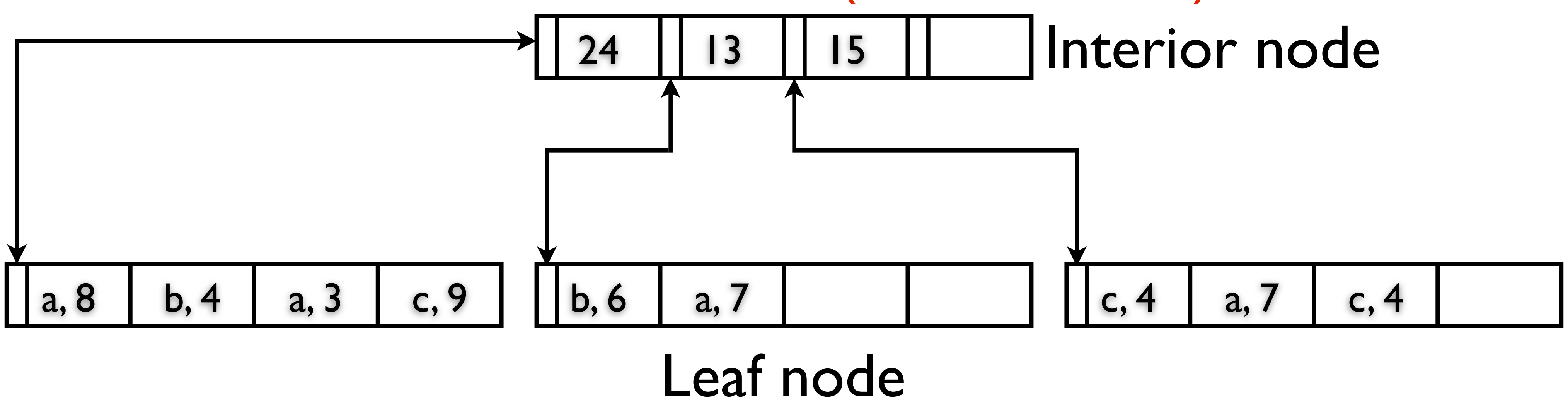


Bitmap 'NY' : 001111001

State
IL
IL
NY
NY
NY
NY
IL
IL
NY

Incrementally Maintaining Run-length Encoded Attributes in Column Stores

- 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.

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

Thank You!

End of Presentation