Incrementally Maintaining Run-length Encoded Attributes in Column Stores

Abhijeet Mohapatra
Stanford University
abhijeet@stanford.edu

Michael Genesereth
Stanford University
genesereth@stanford.edu

ABSTRACT

Run-length encoding is a popular compression scheme which is used extensively to compress the attribute values in column stores. Out of order insertion of tuples potentially degrades the compression achieved using run-length encoding and consequently, the performance of reads. The in-place insertions, deletions and updates of tuples into a column store relation with n tuples take O(n) time. The linear cost is typically avoided by amortizing the cost of updates in batches. However, the relation is decompressed and subsequently re-compressed after applying a batch of updates. This leads to added time time complexity. We propose a novel indexing scheme called *count indexes* that supports O(log n) in-place insertions, deletions, updates and look ups on a run-length encoded sequence with n runs. We also show that count indexes efficiently update a batch of tuples requiring almost a constant time per updated tuple. Additionally, we show that count indexes are optimal. We extend count indexes to support O(log n) updates on bitmapped sequences with n values and adapt them to block-based stores.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms

Algorithms, Theory, Performance

Keywords

column store, run-length encoding, incremental maintenance, in-place updates, indexing

1. INTRODUCTION

Since Stonebraker et al's seminal paper [20] in 2005, column stores have become the preferred platform for data warehousing and analytics. A case study in [16] shows that present enterprise systems are best served by column stores. Although column stores have recently received increased attention, the concept of decomposed storage [5] dates back to the 1980s. [1] presents an excellent overview of the column store technology starting from the decomposed storage model [5]. Column stores support fast reads of data by vertically partitioning a relation and efficiently compressing attribute value using techniques such as run-length encoding and bitmap-encoding. The performance of the reads depends on the compression achieved. It has been shown in [17] that the compression schemes such as run-length encoding and bitmap encoding are sensitive to the ordering of tuples. Hence, out of order insertions of the tuples potentially degrade the compression achieved and consequently, the performance of the reads. Therefore in order to incrementally maintain relations while achieving good compression, column stores require the support for efficient in-place or offset-based inserts, deletes and updates of the tuples.

On an average, the time taken to insert or delete a tuple at a given offset is linear in the number of the tuples of a relation. This is because, in addition to the updated tuple, the storage keys or identifiers of the successive tuples have to be updated as well. In this paper, we consider the problem of efficiently supporting in-place insertions, deletions and updates of tuples in column stores. Previous work in [3, 13, 15, 20] have proposed different techniques to amortize the cost of applying updates to a column store relation in batches. The central theme underlying these techniques (which are discussed in Section 4) is to buffer the updates in a differential store and later merge the updates with a read-optimized store using a table scan. Although such a differential update mechanism amortizes the time to apply updates in bulk, it does not avoid the linear cost of a table scan. In addition, the tuples in the read-optimized store are decompressed and subsequently re-compressed after applying the updates. This requires additional time. We show that we can support in-place inserts, deletes and updates of tuples in a column store relation in time that is sub-linear in the number of tuples and thus avoid a linear scan of the tuples. In this paper, we study the problem of efficiently updating run-length encoded attributes and extend our solution to bitmapped attributes.

Updating run-length encoded attributes: Tuples of a relation in a column store are ordered by sort keys and every attribute is stored separately as a sequence of values. These sequences are either run-length encoded or bitmap encoded. The bitmapped attributes are subsequently run-legth encoded to save space. The run-length encoding scheme compresses successive repetitions of values or runs in a sequence. A run of a value v can be succinctly represented either as a (v, n) or a (v, o) pair, where n is the run-length and o is the offset of the last (or the first) occurrence of v in the run. For presentation purposes, we shall call the former representation of runs as *count-based* and the latter representation as *offset-based*. We illustrate the two representation schemes using

IDEAS12 2012, August 8-10, Prague [Czech Republic]
Editors: Bipin C. Desai, Jaroslav Pokorny, Jorge Bernardino

the following example.

Example 1.1. Consider the sequence of values a, a, b, b, b, a, a, a, b, b. If we use the count-based representation of runs to encode this sequence we would get $(a, 2)$, $(b, 3)$, $(a, 3)$, $(b, 2)$. Instead, if we use the offset-based representation of runs, the sequence would be encoded as $(a, 2)$, $(b, 5)$, $(a, 8)$, $(b, 10)$.

Each representation scheme has its advantages and disadvantages. Since the offsets of the runs increase monotonically, the values in the offset-based representation can be efficiently looked up using augmented Binary Search Trees [2, 10, 12] in time that is logarithmic in the number of the runs. However, if a run is inserted or deleted, the offset of every successive run must be updated as well. Hence, the time taken to update a run-length encoded sequence is linear in the number of runs using the offset-based representation.

In contrast to the offset-based representation, a run can be inserted or deleted using the count-based representation of a run-length encoded sequence in constant time if the offset of updated run is known. However, in order to look up a value at a given offset, the offset of the preceding run must be computed as well. Since the run-lengths are not monotonic in general, the variants of binary trees cannot be used to efficiently look up the value at a given offset. Hence, the time to look-up a value in a run-length encoded sequence is linear in the number of runs using the count-based representation.

Thus choosing between the offset-based and the count-based representation of runs trades-off the time taken to look up and to update runs of values in a sequence. We summarize this trade-off in Table 1. In this paper, we show that we can leverage auxiliary data structures along with an appropriate choice of a representation scheme to optimize this trade-off. Prior work [4, 6, 8, 9, 11, 14, 19, 21, 22] on spatio-temporal aggregation for range queries has studied the problem of incrementally maintaining the partial sums for an array. However, the proposed data structures are not dynamic. For instance, operations such as the in-place insertions and the deletions of new array values are not supported. We distinguish count indexes from the prior work on incrementally maintaining spatio-temporal aggregates in Section 4.

Table 1: Trade-off between the look up and update complexity for run-length encoded sequences

n = number of the runs in the sequence		
Representation Scheme	Cost of operation	
	Look up	Update
Count-based	O(n)	O(1)
Offset-based	O(log n)	O(n)

Contributions: We consider the problem of efficiently updating run-length encoded attributes as a first step towards efficiently updating column stores. We propose a novel indexing scheme called count indexes (in Section 2) that supports offset-based look ups as well as updates in a run-length encoded sequence with n runs in O(log n) time. Thus, count indexes efficiently trade-off the time to look up a value in a sequence with the time taken to update the sequence. We generalize count indexes (in Section 3) to efficiently update bitmap encoded columns. Our contributions in this paper can be summarized as follows:

1. In Section 2.1, we describe a procedure to efficiently create count indexes on a sequence with n runs in O(n) time.
2. In Sections 2.2 and 2.3, we show that count indexes support offset-based look ups and in-place insertion, deletion and update of runs in O(log n) time. We also show that a sequence of k runs can be inserted into a count index on n runs in O(k + log n) time, thus lowering the time complexity of updating a batch of tuples using the techniques proposed in [3, 13, 15, 20].
3. In Section 2.4 we show that the problem of incrementally maintaining a run-lenth encoded sequence with n runs requires Ω(log n) time. Hence our technique of incrementally maintaining run-length encoded attributes using count indexes is optimal.
4. In Section 3, we extend count indexes to support O(log n) updates on bitmap encoded sequences of size O(n). We also adapt count indexes to block-based stores and present the corresponding algorithms for inserting and deleting runs.

2. COUNT INDEXES

In Section 1, we observed that choosing a particular representation scheme to run-length encode a sequence trades-off the time taken to look-up and to update a value at a supplied offset. In this section, we present a novel indexing scheme called *count indexes* that optimizes this trade-off. We define a count index as follows.

Definition 2.1. A *count index* is a binary tree on a sequence of integers. The integers are stored in the leaf nodes. Every interior node in the count index stores the sum of its children's values. Thus the root of the count index stores the total sum of integers in the sequence.

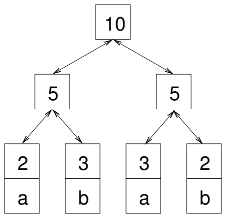


Figure 1: Count index on the sequence $a, a, b, b, b, a, a, a, b, b$

Figure 1 shows a count index on the sequence given in Example 1.1. While the count index is constructed on the run lengths only, we display the values (a and b) as well for presentation purposes. The root node in Figure 1 stores the number of values in the sequence (= 10).

Given a run-length encoded sequence with n runs, we can look up or update runs of values at specified offsets using count indexes in O(log n) time. In the remainder of this section we discuss the algorithms for index creation, look up and maintenance.

2.1 Index Creation

As discussed above, a count index is a binary tree on a sequence of integers. Every node in a count index stores the the sum of the values of its children. To construct a count index on a sequence of n integers, we divide the integers into groups of two. If n is odd, we create a group for the last count. We then sum the counts of the integers in each of the $\lceil \frac{n}{2} \rceil$ groups and recursively build the count index on these $\lceil \frac{n}{2} \rceil$ sums. The detailed algorithm is presented in Figure 2. The pointers to the parent, the left and the right children of a node are stored in $parent$, $lchild$ and $rchild$ respectively.

```
**CreateIndex** ($[c_i]$, $n$)
  **if** $n = 1$ **then**
    return $c_i$
  **else**
    **for** $i = 1$ to $n - 1$ **do**
      $j \leftarrow \frac{i+1}{2}$
      $d_j.parent \leftarrow d_j$
      $d_j.lchild \leftarrow c_i$      $d_j.rchild \leftarrow c_{i+1}$
      $d_j.count = c_i.count + c_{i+1}.count$
      $c_i.parent = c_{i+1}.parent \leftarrow d_j$
      $i \leftarrow i + 2$
    **end for**
    **if** $n \equiv 0 \pmod 2$ **then**
      return **CreateIndex** ($[d_i]$, $\frac{n}{2}$)
    **else**
      $d_j.parent \leftarrow d_j$
      $d_j.count = c_i.count$
      $d_j.lchild \leftarrow c_i$      $d_j.rchild \leftarrow$ NULL
      $c_i.parent \leftarrow d_j$
      return **CreateIndex** ($[d_i]$, $\frac{n+1}{2}$)
    **end if**
  **end if**
```

Figure 2: Algorithm to create a count index on a sequence of n integers $[c_i]$

Time Complexity: In the first pass, we divide the n integers into groups of two and compute the sum of the integers in each group. This takes n units of time. Similarly, the second pass takes $\lceil \frac{n}{2} \rceil$ units of time, the third pass takes $\lceil \frac{n}{4} \rceil$ units of time and so on. The algorithm terminates after log n passes. The time complexity of the index creation algorithm is therefore O($n + \frac{n}{2} + \frac{n}{4} + \ldots + 1$) = O($2 \times n$) = O($n$).

In Section 1, we observed that choosing one representation of a run-length encoded sequence over the other trades-off the time taken to update the sequence with the time taken to look up values at the supplied offsets. In the remainder of the section, we show that count indexes optimize this trade-off by supporting O(log n) offset based look ups and updates on a sequence with n runs.

2.2 Index Look Up

The run lengths in a run-length encoded sequence are not monotonic in general. Therefore we cannot directly perform a binary search on the run lengths to look up the leaf node corresponding to a given offset. However, if we start at the root of the count index, we can determine which of its children contains the leaf node corresponding to the given offset in constant time. Suppose we want to look up the value at offset p. Let the counts of the left child and the right child of the root be l_v and r_v respectively. If $p \leq l_v$, the value at the offset p is located in the left child of the root. Otherwise, the value at the offset p is located in the right child. The left or the right child of the root can be looked up in a similar manner. The look up procedure terminates when we arrive at the leaf which corresponds to the supplied offset. Thus the look up procedure terminates in time that is proportional to the height of the count index. The detailed algorithm for looking up values in a count index is presented in Figure 3.

To illustrate how the look-up algorithm works, let us revisit the count index (Figure 1) on the sequence given in Example 1.1. Suppose we want to look up the value at the 8^{th} position from the start of the sequence. We look up the counts of the left and right children of the root of the count index. The count of the left child (= 5) is less than 8. Therefore, we look up the value at the offset 3 (= 8 - 5) in the right child (say r) of the root. Since the count of the left child of r is 3, our look up procedure terminates and returns the value a. The look-up procedure is shown in Figure 4.

```
**SearchCountIndex** ($p$, $node$)
  **if** $node$ has no children **then**
    return value at $node$
  **else**
    $lchild \leftarrow$ left child of $node$
    $v \leftarrow$ value at $lchild$
    **if** $v > p$ **then**
      **SearchCountIndex** ($p$, $lchild$)
    **else**
      **SearchCountIndex** ($p - v$, $rchild$)
    **end if**
  **end if**
```

Figure 3: Algorithm to look-up values in a count index

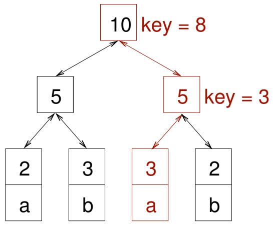


Figure 4: Example of a look up on a count index

Proof of Correctness: We can establish the correctness of the look up algorithm shown in Figure 3 using induction on the height of the count index.

Base Case: (Count index with a root and 2 children i.e. height = 2) Suppose we want to look up the value in the p^{th} position from the start of the sequence. Let l_v and r_v be the counts of the left and right children of the root node. If the offset is positive and less than the size of the sequence, we have:

$$1 \leq p \leq l_v + r_v \tag{1}$$

Therefore, either $p \leq l_v$ or $p - l_v \leq r_v$.

Inductive Hypothesis: We assume that the look up algorithm correctly returns the value at any given offset in a count index of height $\leq k$.

Induction Step: Suppose we look up the value at the offset p in a count index of height $k + 1$. Let l_v and r_v be the counts of the left and the right children of the root node. Following the argument made in the base case, either $p \leq l_v$ or $p - l_v \leq r_v$. Since the children of the root node are at a height of k, the look up algorithm correctly returns the value at the offset p.

Time Complexity: Consider a count index on a sequence of n integers. The algorithm in Figure 3 examines exactly one node at any level of the count index. If the height of the count index is h, the time complexity of the look up procedure is O(h). Using Claim 2.3 (which is presented later) we show that the height h of a count index with n leaves is O(log n). Therefore count indexes support offset based look ups over a run-length encoded sequence in time that is logarithmic in the number of the runs.

Supporting Range Predicates: Given a count index with n leaf nodes, we can also look up the values within a supplied range of offsets in O(log n) time by creating links between the neighboring leaf nodes. Suppose we require values in between the offsets p and q from the start ($p \leq q$). We look up the value at the offset p using the algorithm shown in Figure 3. On reaching the leaf node that corresponds to the offset p, we traverse the leaf's neighbors on the right till we reach the leaf node corresponding to the offset q.

2.3 Index Maintenance

It is essential for any index to be efficiently updatable when new values are inserted or old values are deleted. When values in a sequence are updated, the corresponding count index is affected in one of the following ways:

1. The count of an existing leaf node is updated.
2. A new leaf node is inserted or an existing leaf node is deleted from the count index.

It is easy to handle Case 1 by updating the ancestors of the leaf node in time that is proportional to the height of the count index. However, the count index can be potentially unbalanced by Case 2. Since the counts of the leaf nodes are not monotonic, we cannot use traditional tree rotation schemes which are used to balance AVL Trees [2] and Red-Black Trees [12]. In this section, we discuss techniques to efficiently balance a count index when new values are inserted or old values are deleted. We note that in all of the examples which follow, we discuss the insertion and the deletion of single values or counts into a count index. We can, however, use the index update procedures described below to insert or delete runs of any length (≥ 1).

Deletions: Suppose we want to delete a leaf from a count index. There are two cases depending on whether the leaf has a sibling or not.

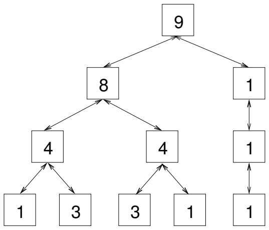
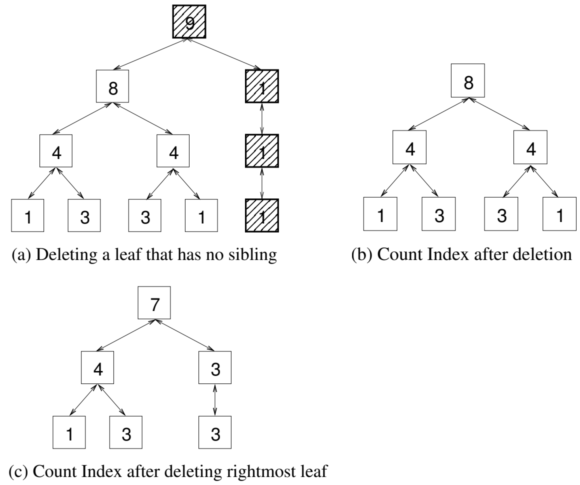


Figure 5: Count Index on a sequence with five run-lengths: 1, 3, 3, 1 and 1

1. (Leaf has no siblings): In this case, we delete the leaf and its ancestors. If the new root has only one child, we make its child the new root node and iterate down the tree.
2. (Leaf has a sibling): In this case, we check whether the leaf (say c_l) or its sibling (say c_s) has a neighbor without any sibling of its own. If neither the leaf or its sibling have such a neighbor, then we delete the leaf node and update the value at its parent. Otherwise, let c_n be the neighbor of the leaf (or its sibling) that has no sibling of its own. We delete c_l and assign c_n to be the new sibling of c_s.

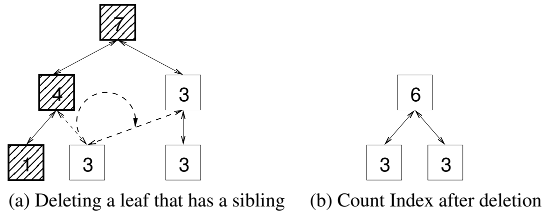
Consider a count index with five run-lengths: 1, 3, 3, 1 and 1 (shown in Figure 5). We note that although the count index appears to be unbalanced, its height is in fact O($\log_2$(no. of nodes) $=$ $\log 5)$. We show that this is always the case using Claim 2.3. Suppose we want to delete the rightmost count from the count index shown in Figure 5. Since the leaf node corresponding to the rightmost count (= 1) does not have a sibling (Case 1), we delete it and all of its ancestors which are shown as hatched nodes in Figure 6(a). The state of the count index after deleting its rightmost leaf is shown in Figure 6(b). Suppose that we delete the rightmost



(a) Deleting a leaf that has no sibling (b) Count Index after deletion

(c) Count Index after deleting rightmost leaf

Figure 6: Deleting a leaf that has no siblings



(a) Deleting a leaf that has a sibling (b) Count Index after deletion

Figure 7: Deleting a leaf that has a sibling

leaf again. Since the rightmost leaf has a sibling and no neighbors without a sibling, we delete the leaf node and update the value of its parent. The resulting count index is shown in Figure 6(c). Now suppose we delete the leftmost count (= 1). Since the leftmost leaf has a sibling (with a count of 3) and a neighbor without any siblings (Case 2), we delete the leftmost node and pair up the sibling with the neighboring leaf. This procedure is shown in Figure 7(a) and (b). In Figure 7(a), the hatched nodes are deleted.

Claim 2.1. *Deleting a leaf from a count index takes time that is linear in the height of the count index.*

PROOF. Let $f(k)$ and $g(k)$ be the time taken to delete a node at a height of k using cases 1 and 2 respectively. Then the following recurrence relations hold.

$$f(k) \leq 1 + \max\{f(k-1), g(k-1)\} \quad (2)$$
$$g(k) = f(k-1) + 1 \quad (3)$$

On solving for $f(k)$ and $g(k)$, we get $f(k) = g(k) = c \times k + d$ where c and d are constants. Therefore, a leaf can be deleted from a count index of height h in O(h) time.

Updating the Root: When we delete a leaf node which causes one of the root's children to be deleted as well (an example is shown in Figure 6(a)), the root node of the count index has to be updated. As a general rule, after the deletion procedure terminates, we check whether the root of the count index has two children or not. If the root has two children, no further update is required. However, if the root has only one child, we set its child to be the new root node and recurse down the count index.

Inserts: The process of inserting new leaves into a count index is similar to the process of deleting leaves. Suppose we want to insert a leaf node w between two leaves: u and v. There are two cases depending on whether u or v have siblings or not.

1. (One of u or v has no siblings): Without loss of generality, we can assume that u has no siblings. We make w the sibling of u and update the count of their parent.

2. (Both u and v have siblings): If u and v are siblings of each other then we make w the sibling of u by updating the count at v. Next, we insert a leaf with the same count as v at the leaf level and set its parent to have the same count as itself.

Claim 2.2. *Inserting a leaf into a count index takes time that is linear in the height of the count index.*

PROOF. Let $f(k)$ and $g(k)$ be the time taken to insert a node at a height of k using cases 1 and 2 respectively. Then, the following recurrence relations hold.

$$f(k) = f(k-1) + 1 \tag{4}$$

$$g(k) \leq 1 + \max\{f(k-1), g(k-1)\} \tag{5}$$

On solving for $f(k)$ and $g(k)$, we get $f(k) = g(k) = c \times k + d$ where c and d are constants. Therefore, a leaf can be inserted into a count index of height h in O(h) time.

Balancing Count Indexes: Using Claims 2.1 and 2.2, we have established that the node insertions and deletions in a count index of height h take O(h) time. We still need to show that an updated count index is balanced i.e. $h =$ O($\log n$). Count indexes can be potentially unbalanced due to nodes which are not paired up with their neighbors. We call such unpaired nodes as *holes*. Since a node in a count index stores the sum of its children's counts, we cannot use traditional tree rotation techniques for tree balancing as is used in AVL Trees [2] and Red-Black Trees [12].

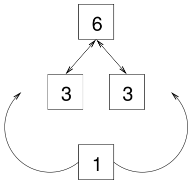


Figure 8: Inserting counts at a 'hole' position

Consider a count index with n leaf nodes. The maximum number of holes at the leaf level of the count index is $\lceil \frac{n}{2} \rceil$. When we insert or delete a node from the count index, we propagate the holes to higher levels where they either pair up with an existing hole or cause a node to be split (while inserting a node) or cause two nodes to be merged (while deleting a node). The count index shown in Figure 7(b) has two hole positions at the leaf level (shown in Figure 8). When we insert a count (say 1) into either of these holes, the hole propagates to the next level and pairs up with the node which was previously the root. The final state of the count index is shown in Figure 9.

Claim 2.3. *The height of a count index is logarithmic in the number of its leaves. i.e. the count index is always balanced.*

PROOF. Suppose that there are N nodes that are paired up in some level of a count index. There can be a hole adjacent to each pair. The maximum number of holes in each level is $\lceil \frac{N}{2} \rceil$. In every level of a count index at most one-third of the nodes can potentially be holes and at least two-third of the nodes are paired up. Hence, the height of a count index with n leaf nodes is at most $\log\left(\frac{3}{2}\right) \times \log n$ = O($\log n$).

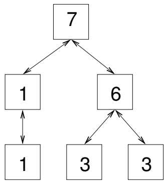


Figure 9: State of the count index after inserting

Therefore, we can incrementally maintain a run-length encoded sequence using a count index in time that is logarithmic in the number of the runs in the sequence.

Bulk Inserting Sequences: Consider a count index with n leaf nodes. Suppose that we would like to insert a sequence of k runs. Naively, we could insert the k runs using the insert procedure in O($k \times \log(n+k)$) time. However, we could do better if we made use of the fact that the k runs are adjacent. We present below a procedure that efficiently inserts a sequence of k runs ($\lceil c_i \rceil$) into a count index T with n leaves.

- We construct a count index T' on the sequence of k runs $\lceil c_i \rceil$.
- We merge T and T'. (Figure 10)
  - Let u and v be the leaves in T between which the sequence $\lceil c_i \rceil$ is inserted.
  - If u and v are siblings, we split them up converting them into holes.
  - We pair up the left-most leaf of T' with u if both nodes are holes. Similarly we pair up the right-most leaf of T' with v if both nodes are holes.
  - We insert the rest of the leaves of T' into T without further changes and iterate at the next higher level of T and T'.
- At a height of $\log k$, we insert the root of T' into T.

Time Complexity of Bulk Inserts: The construction of T' which is a count index on k runs takes O(k) time. To insert the root of T' into T at a height of $\log k$, we require O($\log \frac{n}{k}$) time. Let us denote the time taken to merge two count indexes with n and k leaves as $f(n, k)$. The bulk insert procedure results in the following recurrence relations.

$$f(n,k) = f\left(\frac{n}{2}, \frac{k}{2}\right) + \mathrm{O}(k) \tag{6}$$

$$f(n,1) = \mathrm{O}(\log n) \tag{7}$$

On solving for $f(n,k)$ we get $f(n,k) = \mathrm{O}(k + \log n)$. Thus using the above procedure, we can amortize the cost of inserting a sequence of runs in time that is almost linear in the number of the inserted runs.

2.4 Optimality of Incremental Maintenance using Count Indexes

We have shown in Sections 2.2 and 2.3 that count indexes support O($\log n$) offset-based look ups and in-place inserts, deletes and updates of runs into a run-length encoded sequence with n runs. We now use a lower bound result on the complexity of incrementally maintaining *partial sums* which has been proved in [8, 9, 19] to show that the problem of incrementally maintaining a run-length encoded sequence with n runs requires $\Omega(\log n)$ time . Given an

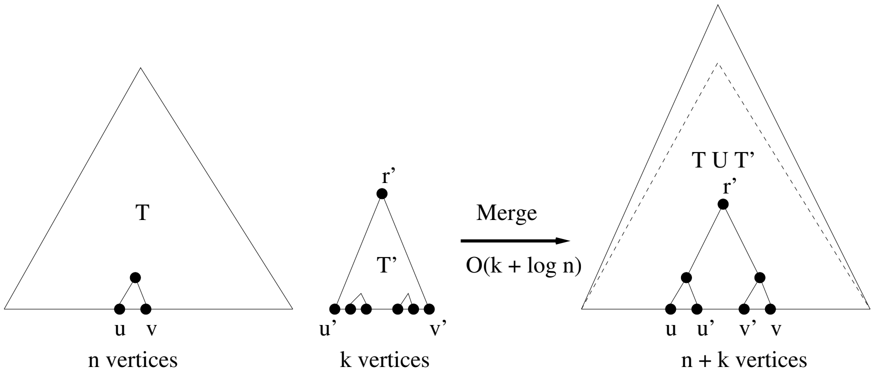


Figure 10: Bulk Inserting a sequence of runs into a count index

array of numbers $\{a[i]\}$ and a parameter k, the partial sums problem computes $\sum_{i=0}^{k} a[i]$ subject to updates of the form $a[i]$ += x, where x is a number. [8, 9, 19] have independently established that incrementally maintaining partial sums while supporting in-place updates requires $\Omega(\log n)$ time.

The problem of incrementally maintaining partial sums over an array $\{a[i]\}$ of length n can be reduced to the problem of incrementally maintaining run-length encoded sequences. We create a run-length encoded sequence with n runs. Every i corresponds to a run and the run-length of the i^{th} run is equal to $a[i]$. An update $a[i]$ += x to the partial sums problem is equivalent to incrementing the run-legth of the run at the offset $\sum_{j=0}^{i} a[i]$ by x in the run-length encoded sequence. The computation of the partial sum at $a[k]$ is equivalent to the computation of the starting offset of the $i + 1^{th}$ run. Therefore, incrementally maintaining a run-length encoded sequence with n runs requires $\Omega(\log n)$ time. In fact, the problem of incrementally maintaining run-length encoded sequences is more general than the problem of incrementally maintaining partial sums because the partial sums problem does not support in-place insertion or deletion of array elements. Thus, leveraging count indexes to incrementally maintain a run-length encoded sequence is optimal.

3. EXTENSIONS

In the previous section we showed that count indexes can efficiently look up and update a run-length encoded sequence with n runs in O(log n) time. Bitmap encoding is another compression technique that is used in column stores to compress the attributes with a few distinct values. Bitmaps are typically sparse and are further compressed using run-length encoding. In this section, we extend count indexes to efficiently update bitmap encoded sequences. We also discuss how count indexes can be adapted to block-based stores where the number of I/Os determine the time complexity of updates.

Updating Bitmap Encoded Sequences: Bitmap encoding a sequence creates a bit-vector for every distinct value in the sequence. If the sequence contains the value v at the i^{th} position, then the bit-vector corresponding to the value v contains the bit 1 at the i^{th} position. Bitmaps are typically sparse and are run-length encoded. When new values are added to the sequence, additional bits are appended to the end of a bitmap. When values are deleted from a bitmap, the corresponding bits are not deleted. Instead, the deleted values are tombstoned (see [10] for details). The insertion of a new value or the deletion of an existing value at a given offset in a bitmap takes time that is linear in the size of the bitmap. We can significantly improve the update complexity by extending count indexes to operate on bitmap encoded sequences. For every distinct value in an input sequence, we create a count index on the corresponding run-length encoded bitmap. The bitmaps can be looked up and updated at a given offset in time that is logarithmic in the number of runs in the bitmap. We present an example to show how count indexes can be extended to operate on bitmapped sequences.

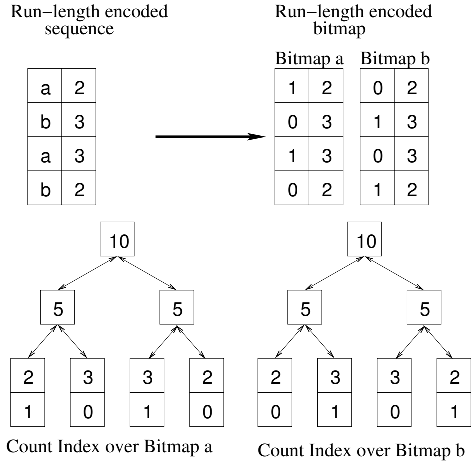

Run−length encoded sequence
Run−length encoded bitmap
Bitmap a
Bitmap b
a 2
b 3
a 3
b 2
1 2
0 3
1 3
0 2
0 2
1 3
0 3
1 2
10
5
5
2 1
3 0
3 1
2 0
10
5
5
2 0
3 1
3 0
2 1
Count Index over Bitmap a
Count Index over Bitmap b

Figure 11: Count Indexes on Run-length Encoded Bitmaps

Example 3.1. Suppose we bitmap encode the sequence as given in Example 1.1. we get two bitmaps [1 1 0 0 0 1 1 1 0 0] for the value 'a' and [0 0 1 1 1 0 0 0 1 1] for 'b'. Run-length encoding these bitmaps would produce the sequences: [(1, 2), (0, 3), (1, 3), (0, 2)] and [(0, 2), (1, 3), (0, 3), (1, 2)] respectively. Two count indexes are constructed on the resulting sequences. These indexes (shown in Figure 11) are identical to each other.

In general, if an input sequence has n runs and k distinct values, then we could look up and update the bitmap encoded sequence using count indexes in time that is proportional to $k \times \log \frac{n}{k}$ in the worst case. However, we can independently execute the look up and update algorithms over each of the k bitmaps in parallel in O(log n) time.

Adapting Count Indexes to Block-Based Stores: In Section 2, we stored a single run count in each leaf node of a count index and the sum of two child nodes in their parent. In order to adapt count indexes to block-based stores where each block of data is a *disk page* instead of a single value, we modify count indexes to store

Figure 12: Incremental Maintenance of Count Indexes in Block-based Stores

(a)

```
Insert(Node $N$, Count $C$, Value $V$)
if $N$ is not full then
    Insert $\langle V, C \rangle$ into $N$
    Update the sum of the counts of $N$ and its ancestors
else
    Create a new node $M$
    Move $\langle V, C \rangle$ and half of the values or pointers and
    the counts present in $N$ over to $M$
    Update the sum of the counts of $N$ and $M$
    $P \leftarrow$ parent of $N$
    $C' \leftarrow$ Sum of the counts of $M$
    Insert($P$, $C'$, $M$)
end if
```

(b)

```
Delete(Node $N$, Count $C$, Value $V$)
if $N$ is exactly half-filled then
    Delete $\langle V, C \rangle$ into $N$
    if all neighbors of $N$ are half-filled then
        Merge a neighbor of $N$ say $M$ with $N$
        $P \leftarrow$ parent of $M$
        Suppose $C' \leftarrow$ Sum of counts of $M$
        Delete($P$, $C'$, $M$)
    else
        $M$ is a neighbor of $N$ which is more than half-filled
        Let $Q$ be a pointer contained in $M$ which is closest
        to $N$
        Let the sum of counts of $Q$ be $C'$
        Delete($M$, $C'$, $Q$)
        Insert($N$, $C'$, $Q$)
    end if
    Update the sum of the counts of $N$ and its ancestors
else
    Delete $\langle V, C \rangle$ from $N$
    Update the sum of the counts of $N$ and its ancestors
end if
```

multiple counts in each node.

Suppose each disk page is S Bytes long and the width of the attribute to be stored is W Bytes long. We assume that the run-lengths can be represented as 4 Byte integers. In each leaf node of our modified count index, we store a maximum of N_l attribute values, their respective counts and the sum of their N_l counts where $N_l = \frac{S-4}{W+4}$. In each interior node, we now store a maximum of N_i pointers (8 Bytes long) to its children and the sum of their respective counts (4 Bytes each) where $N_i = \frac{S-4}{W+12}$. To adapt count indexes to block-oriented stores, we also establish a new index invariant: every non-root node is at least *half-full* i.e. every leaf node stores at least $\lceil \frac{N_l}{2} \rceil$ attribute values and counts and every interior nodes stores at least $\lceil \frac{N_i}{2} \rceil$ pointers to its children.

The modified algorithms for inserting and deleting a run are presented in Figures 12(a) and (b) respectively.

4. RELATED WORK

The problem of efficiently updating compressed columns has ben well-studied in the database community in the context of optimizing column store updates. While column stores such as [16] support append-only inserts, others such as [3, 13, 15, 20] merge differential updates from a write-optimized store into a read-optimized store. An append-only insert scheme such as [16] is easy to implement. However such a scheme could arbitrarily degrade the compression achieved on the non-sort key columns by order-sensitive schemes such as run-length encoding or bitmap encoding [17] and thus lead to inefficient look ups.

Column stores such as [3, 13, 15, 20] maintain a differential store in addition to a read-optimized store. The updates are buffered in the differential store and are later merged with the read-store. This technique amortizes the cost of updates by applying the updates in batches. Recently, [13] has proposed a data structure called Positional Delta Trees (PDTs) which efficiently buffer updates in such a differential store. PDTs record the deltas of the offsets of the updated tuples within a segment. The idea of operating on deltas of offsets is common to both PDTs and count indexes. However there are differences that separate the two. Count indexes are designed to operate on a single store only (one which contains the compressed representations of the tuples) while supporting efficient reads and updates. On the other hand, a PDT visualizes a database as a two-pronged store: a read-optimize store and a write-optimized store (one which contains the uncompressed representation of the tuples). Second, count indexes are more space efficient than PDTs. If k values are inserted into and later deleted from a sequence of n values, the space required to store the corresponding PDT would be O($n+k$), whereas a count index would require only O(n) space. Further, a PDT requires a merge scan to bulk insert values or runs of values. Therefore the time complexity of updating a relation is O(n) where n is the total number of tuples. On the other hand, a count index requires O($k \times \log n$) time in the worst case to insert k runs. If the number of updated tuples is a small fraction of the total number of tuples ($k \ll n$), O($k \times \log n$) time updates are much better than O(n) updates. If the k runs are adjacent, then they can be inserted into a count indexed in O($k + \log n$) time which is almost linear in the number of updated tuples. Additionally, while applying updates using PDTs the columns are decompressed and later re-compressed, which can be expensive. In contrast, count indexes operate directly on the compressed sequences of values and can be incrementally maintained without decompressing the sequence.

Incremental maintenance of aggregates: Prior work [4, 6, 8, 9, 11, 14, 18, 19, 21, 22] on spatio-temporal aggregation for range queries have studied the problem of efficiently computing the partial sums $\sum_{i=0}^{k} a[i]$ for an array $\{a[i]\}$ subject to the updates of the form $a[i]$ += x. As shown in Section 2.4, the problem of incrementally maintaining partial sums is closely related to the problem of incrementally maintaining run-length encoded sequences. However, the data structures [8, 9, 11, 14, 18, 19, 21, 22] that have been proposed for problem of incrementally maintaining partial sums are not dynamic. For instance, the insertions and the deletions of values in the array are not supported on proposed data structures. These operations are equivalent of supporting in-place insertion and deletion of runs in a run-length encoded sequence. There is another distinction between incrementally maintaining run-length encoded sequences and the problem of efficiently computing aggregates over spatial and temporal data. Consider a set of points in the 2-D plane which is shown in Figure 13. Suppose we are interested in efficiently computing the sum of the weights of the points within a given bounding box, say $[2, 1] \times [4, 5]$. To efficiently compute the sum of the weights we can leverage previously proposed data structures such as Orthogonal Range Trees and k-D Trees [6].

Table 2: Performance of count indexes on various encoding schemes

N = number of values and n = number of runs

Encoding Scheme	create	look-up	update	bulk insert (k contiguous runs)
Run-length Encoding	$O(n)$	$O(\log n)$	$O(\log n)$	$O(k{+}\log n)$
Bitmap Encoding (parallel operations)	$O(n)$	$O(\log n)$	$O(\log n)$	$O(k{+}\log n)$

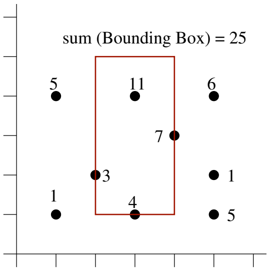


Figure 13: Aggregation of weights of points on a 2-D plane

When we insert a new point say (3, 2) into our point set, the value of the other points is not affected. This is because there is an implicit ordering of the points in a 2-D plane based on their x and y coordinates which are monotonically increasing. However, if we choose a different ordering of the points such as the *lexicographic rank* of a point in the point set, the insertion of a new point affects the rank of other points. The tuples in a column store are typically ordered lexicographically using sort-keys and the offset of a tuple is given by the lexicographic rank of the sort-key value for that tuple. This introduces a linear time complexity for the in-place insertion and deletion of the tuples.

There has also been prior work (e.g. [7]) on efficiently updating cumulative frequencies. [7] has proposed a data structure called *Fenwick Trees* that supports O(log n) time look ups and in-place updates of the frequencies. However, unlike count indexes, Fenwick trees have to be reconstructed if any new frequencies are inserted or if any existing frequencies are deleted. Count index is the first data structure that supports O(log n) in-place inserts, deletes and updates of run and off-set based look ups on an O(n) sized run-length encoded or bitmap encoded sequence.

5. CONCLUSION

In this paper, we have proposed a novel indexing scheme called *count indexes* that supports efficient offset-based look ups and in-place inserts, deletes and updates of runs in a run-length encoded sequence. Count indexes significantly reduce the time complexity of updating values at supplied offsets in a run-length encoded sequence with n runs from O(n) to O(log n). We have also shown that the lower bound for incrementally maintaining run-length encoded sequences with n runs is $\Omega(\log n)$. Therefore our technique of incrementally maintaining run-length encoded sequences using count indexes is optimal. We have also generalized count indexes to incrementally maintain bitmap encoded sequences and have adapted count indexes to block-based stores.

Column stores compress the attribute values using run-length encoding and bitmap encoding. Hence count indexes can be potentially leveraged as an auxiliary data structure to efficiently support single value updates in column stores relations. In addition, count indexes can bulk insert a sequence of tuples in time that is almost constant per inserted tuple. This lowers the time complexity of updating a batch of tuples from being linear in the total number of tuples using previous techniques such as [3, 13, 15, 20] to be linear in the number of updated tuples.

To conclude, we summarize the performance of count indexes on different compression schemes in Table 2.

6. REFERENCES

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *PVLDB*, 2009.

[2] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 1962.

[3] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, 2002.

[4] B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proceedings of the fifth annual symposium on Computational Geometry*, 1989.

[5] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. SIGMOD, 1985.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. 2001.

[7] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 1994.

[8] M. L. Fredman. A lower bound on the complexity of orthogonal range queries. *J. ACM*, 1981.

[9] M. L. Fredman. The complexity of maintaining an array and computing its partial sums. *J. ACM*, 1982.

[10] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. 2008.

[11] S. Govindarajan, P. K. Agarwal, and L. Arge. Crb-tree: An efficient indexing scheme for range-aggregate queries. ICDT, 2002.

[12] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, 1978.

[13] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. SIGMOD, 2010.

[14] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. *SIGMOD*, 1997.

[15] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. SIGMOD, 2007.

[16] J. Krueger, M. Grund, A. Zeier, and H. Plattner. Enterprise application-specific data management. EDOC, 2010.

[17] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowledge Engineering*, 2010.

[18] I. Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Transactions on Knowledge. and Data Engineering*, 2005.

[19] M. Pǎtraşcu and E. D. Demaine. Tight bounds for the partial-sums problem. SODA, 2004.

[20] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. VLDB, 2005.

[21] Y. Wang, M. Zhang, J. Li, and L. Zhang. An improved ag-tree based on column store. In *International Conference on Artificial Intelligence and Education*, 2010.

[22] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *VLDB*, 2003.