I/O-Efficient Bundled Range Aggregation

Yufei Tao Cheng Sheng

Abstract—This paper studies *bundled range aggregation*, which is conceptually equivalent to running a range aggregate query separately on multiple datasets, returning the query result on *each* dataset. In particular, the queried datasets can be *arbitrarily* chosen from a large number (hundreds or even thousands) of candidate datasets. The challenge is to minimize the query cost no matter how many and which datasets are selected. We propose a fully-dynamic data structure called *aggregate bundled B-tree (aBB-tree)* to settle bundled range aggregation. Specifically, the aBB-tree requires linear space, answers any query in $O(\log_B N)$ I/Os, and can be updated in $O(\log_B N)$ I/Os (where N is the total size of all the candidate datasets, and B the disk page size), under the circumstances where the number of datasets is $O(B)$. The practical efficiency of our technique is demonstrated with extensive experiments.

Index Terms—Aggregation, Range Search, Index

1 Introduction

Range aggregation computes an aggregate result about the data items satisfying a range predicate. It has been extensively studied (see Section 2 for a survey) due to its importance in a great variety of applications. To be specific, denote by D a dataset where each item has a *key* in the real domain. Given an interval r, let $D(r)$ be the set of items in D whose keys are covered by r. A *range count* query returns the number of items in $D(r)$. Sometimes each item may carry a real-valued *weight*. In this case, a *range sum* query returns the total weight of the items in $D(r)$. Similarly, range aggregation can also be performed using other aggregate functions. For example, a *range average* query reports the average weight of the items in $D(r)$.

1.1 Bundled Range Aggregation

This paper studies *bundled range aggregation*, which can be regarded as the simultaneous execution of a range aggregate query on multiple datasets, returning a result for each dataset.

Formally, let D be a set of N data items. Each item has a *key* and a *weight*, both of which are values in the real domain $\mathbb{R}$. Furthermore, each item also carries a *color*. Let b be the number of possible colors; each color thus can be represented as an integer in $\{1, 2, ..., b\}$. Denote by D_i ($1 \leq i \leq b$) the set of values in D having color i. Since every item has exactly one color, D_1, D_2, ..., D_b are mutually disjoint; and their union is exactly D. Each D_i is referred to as a *category*.

Definition 1 (Bundled Range Sum/Count): Given an interval r in $\mathbb{R}$ and a non-empty set $Q \subseteq \{1, 2, ..., b\}$, a bundled range sum (BRS) *query returns, for each $i \in Q$, the sum of the weights of all the items in D_i whose keys are covered by r.*

A bundled range count (BRC) *query is a special BRS query when the weights of all items are 1.* □

The query parameter Q is called a *category preference*. Note that as Q can be any non-empty subset of $\{1, ..., b\}$, the total number of possible Q is $2^b - 1$.

Our objective is to design an index structure to answer any BRS/BRC query with a small number of I/Os. Furthermore, the structure should be *fully dynamic*, namely, it must allow an item to be inserted or deleted when the item appears or disappears in D. Apparently, once the BRS and BRC problems are settled, we can immediately solve *bundled range average* queries[1], which return, for each $i \in Q$, the average weight of all the items in D_i whose keys fall in r.

Computation model and assumptions. We consider the standard *external memory model* [2]. In this model, a computer is equipped with memory of M words, and a disk of an unbounded size. The disk is formatted into disjoint *pages*, each of which consists of B consecutive words. An *I/O* either reads a page of data into memory, or conversely, writes B words from memory to a page. The values of M and B satisfy $M \geq 2B$, namely, the memory can accommodate at least two pages. The *space* of a structure is defined as the number of pages occupied, whereas the *time* of an algorithm is defined as the number of I/Os performed. CPU time is for free. Every integer or real value can be stored using a single word.

The value of B in practice ranges from 1k (words) to 64k. We make the assumption that the number b of colors is no more than B by a constant factor, i.e., $b = O(B)$. In other words, our techniques are not designed to support arbitrarily many categories, but instead, a large number

- *Yufei Tao is with the Chinese University of Hong Kong, Hong Kong. E-mail: taoyf@cse.cuhk.edu.hk.*
- *Cheng Sheng is with Google Switzerland. E-mail: jeru.sheng@gmail.com.*

1. Note that an average can simply be obtained by dividing a sum by a count.

method	space	query	update	remark
one-for-all	$O(N/B)$	$O(f \log_f N)$	$O(\log_f N)$	f can be any value in $[3, B]$
one-for-each	$O(N/B)$	$O(\|Q\| \log_B N)$	$O(\log_B N)$	–
ours	$O(N/B)$	$O(\log_B N)$	$O(\log_B N)$	the update cost is amortized

TABLE 1
Comparison of the previous results and ours (assuming $b = O(B)$)

of them. As is evident from the next subsection, this is a valid assumption in numerous real-life applications.

1.2 Applications and motivation

Bundled range aggregation is motivated by the fact that, in applications where data are naturally divided into *categories*, a user is often interested in *some*, but not all, of the categories. For example, consider a crime database that stores, for each city in the US, the number of crimes each day. Here, every city is a category, in which each item is a pair of (*date*, *crime number*), where *date* is the item's search key, and *crime number* is its weight. A bundled range sum query can be:

> Find the total number of crimes during 1-10 Jan. 2010 in *each* of the 50 state capitals in the US.

The category preference Q of the above query has 50 categories (one for each state capital). For each category, the query returns a value, which sums up the crime numbers of the days during 1-10 Jan. 2010 of the corresponding capital. In general, a unique feature of bundled range aggregation is that it offers vast flexibility to a user in selecting the queried categories. In particular, the set of those categories (i.e., Q) can be completely *ad-hoc*, as they do not need to be restricted to any hierarchy at all. In the above scenario, for instance, a category preference can be *any non-empty subset* of all the 1248 cities[2] in the US.

The query given earlier is conceptually equivalent to 50 individual range sum queries, each of which is issued on a capital. Executing those queries separately, however, may incur heavy I/O penalty, especially when the number of cities concerned (i.e., the size $|Q|$ of Q) is substantially higher. This motivates the question whether there exists a mechanism for processing bundled range aggregation that remains highly efficient even when $|Q|$ is very large.

Applications similar to the previous one are abundant in practice. For example, in a stock database, each stock forms a category; and an item is a pair of (*date*, *volume*), which records the trading volume of the stock on a particular day. In this case, a meaningful *bundled range average* query is "retrieve the average daily trading volume of each of the 500 stocks in the S&P500 index during 1-31 Jan. 2010". As yet another example, consider a tax database, where each category is a state; and an item is the amount of tax paid by an individual in that state. Then, a *bundled range count* query can be used to "find the number of people that paid at least 10000 dollars of tax in California, Pennsylvania, Florida, and New York, respectively".

2. Data (year 2002) from U.S. Census Bureau, counting all cities with populations at least 25000.

1.3 How well can aggregate B-trees do?

The B-tree is a well-known structure for solving the classic problem of range reporting. With slight augmentation, a B-tree can be changed into an *aggregate B-tree (aB-tree)* [16], which supports range aggregation effectively. As reviewed in Section 2, the aB-tree solves any range sum (hence also, count and average) query in $O(\log_B N)$ I/Os, and can be updated in $O(\log_B N)$ I/Os per insertion/deletion, where N is the dataset cardinality. There are two straightforward ways to apply aB-trees for bundled range aggregation:

- **One aB-tree on all categories (*one-for-all*).** That is, the items of all categories are indexed with a single aB-tree. This method has the drawback that, when the number of categories is large, a great amount of aggregate values need to be squeezed into each internal node. As a result, the size of a node increases significantly, which in turn severely compromises query and update efficiency. As analyzed in Section 3, *one-for-all* requires $O(f \log_f N)$ I/Os to answer a query and handle an update, where f can be any value between 3 and B.
- **One aB-tree for each category (*one-for-each*).** Another approach is to create a separate aB-tree for each category. Given a bundled range sum query, we simply search the $|Q|$ aB-trees on the categories in Q, respectively. The query cost is bounded by $O(|Q| \log_B N)$ I/Os.

1.4 Our main results

We are not aware of any previous study dedicated to bundled range aggregation. This problem, given its importance both as a standalone operator and as the building brick of more complex problems, deserves specialized efforts to improve the above straightforward methods. This paper fills the gap by showing that, when $b = O(B)$, the problem can be nicely solved by an elegant structure named the *aggregate bundled B-tree (aBB-tree)*. Specifically:

- An aBB-tree consumes linear space, namely, $O(N/B)$ pages, where N is the total number of items in all categories.

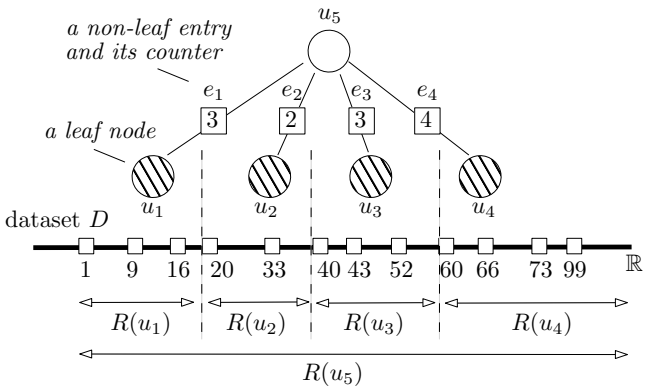


Fig. 1. An aB-tree

- It answers a bundled range sum/count/average query in $O(\log_B N)$ I/Os[3].
- It is fully dynamic. Each insertion/deletion can be performed in amortized $O(\log_B N)$ I/Os.

Table 1 compares the performance of the aBB-tree to that of the solutions mentioned in Section 1.3. The aBB-tree is simple enough to be incorporated into a commercial DBMS. It is merely a traditional B-tree, where each internal node is associated with several *sequential* pages. The overall effect is as if there *was* an aB-tree dedicated to each category, but physically all those aB-trees are *bundled* together in a space-efficient manner that permits efficient queries and updates.

The rest of the paper is organized as follows. Section 2 formalizes bundled range aggregation and reviews the related techniques. Section 3 presents the aBB-tree and its query algorithm. Section 4 elaborates the insertion and deletion procedures. Section 5 experimentally evaluates the efficiency of the aBB-tree. Finally, Section 6 concludes the paper with a summary of our findings.

2 PRELIMINARIES

Aggregate B-tree (aB-tree). When the dataset D has only $b = 1$ category, BRS (BRC) retrieval degenerates to the traditional range sum (count) problem, which is nicely solved by the aB-tree [16]. Next, we review this structure by explaining how to use it to answer range count queries (extensions to range sum are straightforward)

The aB-tree is essentially a B-tree with the only difference that, each internal entry e is augmented with a *counter*, which equals the number of data items in the subtree of e. To illustrate, Figure 1 shows an aB-tree, which has four leaf nodes $u_1, ..., u_4$, and a root u_5. For example, node u_1 contains three items 1, 9, and 16, while node u_2 includes two items 20 and 33. Therefore, their parent entries e_1 and e_2 (in u_5) carry counters 3 and 2, respectively. Counter maintenance does not incur extra I/O overhead (asymptotically), on top of the cost in updating the B-tree itself [16]. In other words, every insertion/deletion can be handled in $O(\log_B N)$ I/Os.

Each node u is associated with a *range* $R(u)$. If u is a leaf node, $R(u)$ equals $[x, x')$, where x (resp. x') is the smallest item stored in u (resp. the leaf succeeding u). In the special case where no leaf succeeds u, $x' = \infty$. If u is an internal node, $R(u)$ is the union of the ranges of all the child nodes of u. For example, $R(u_1)$ equals $[1, 20)$, noticing that 20 is the smallest item in u_2. Similarly, $R(u_2) = [20, 40)$, whereas $R(u_5)$ is $[1, \infty)$.

To answer a range count query with interval r, the query algorithm first initializes the temporary result res to 0, and then starts by processing the root. In general, let u be the node being processed (i.e., at the beginning, u is the root). If u is a leaf node, the algorithm simply adds to res the number of items in u that fall in r. Otherwise (i.e., u is an internal node), it adds to res the counters of all those entries e in u, such that the child node v of e has a range $R(v)$ contained in r. After this, the algorithm recursively processes the child nodes v of u whose $R(v)$ intersects, but is not contained in, r. Standard analysis shows that at most two nodes are accessed at each level of the tree. As each node occupies only $O(1)$ pages, the total query cost is $O(\log_B N)$.

Other related work. Our BRC problem should not be confused with *colored range counting* [8]. Let D be a set of items, each of which is a real value, and associated with a color. Given an interval r, a colored range count query returns the number of distinct colors of the items that appear in r. In other words, it is the colors that are counted, instead of the items. Moreover, only a single value is returned as the query result, unlike a BRC query which should report a value for each category in Q.

Range aggregation has been widely investigated in the database area, e.g., [1], [4], [6], [7], [9], [11], [12], [14], [16], [17], [18], to mention just a few. Those approaches, however, are specific to their own contexts; for our problems in Definition 1, none of them yields a notable advantage over the *one-for-each* method (as explained in Section 1.3). Finally, it is worth mentioning that our work is irrelevant to research on *categorical* data (see, for

3. The actual query complexity is $O(\log_B N + |Q|/B)$. But since $|Q| <= b = O(B)$, the term $|Q|/B$ is really $O(1)$.

example, [13], [15]). The term "category" in this paper refers to a *group* of items but each item is a value in an ordered domain, as opposed to a categorical domain.

3 THE AGGREGATE BUNDLED B-TREE

In the sequel, we will first analyze the defects of *one-for-all*, and then describe a static version of the aggregate bundled B-tree (aBB-tree). In Section 4, we will make the aBB-tree dynamic by elaborating the insertion and deletion algorithms. For simplicity, our discussion focuses on bundled range counting (BRC), because extensions to bundled range sum (BRS) queries are straightforward.

One-for-all. As reviewed in the previous section, in an aB-tree, an entry e of an internal node stores only one counter. This is no longer sufficient for solving the BRC problem. Instead, b counters are necessary: the i-th ($1 \leq i \leq b$) counter of e equals the number of items of color i in the subtree of e.

By default, each leaf node of the aB-tree stores $\Theta(B)$ data items, while each non-root internal node has $\Theta(B)$ child nodes. If b counters are associated with each entry, an internal node will need to store $O(bB)$ counters in total. Since every counter requires a word, an internal node with $O(bB)$ counters occupies $O(b)$ pages. To answer a BRC query, we need to access $O(\log_B N)$ internal nodes; therefore, the total query cost becomes $O(b \log_B N)$. Similarly, each update of the aB-tree incurs the same overhead.

To alleviate the deficiency, a radical approach is to manually decrease the fanout of internal nodes. In general, if each internal node has at most f child nodes, the node contains at most fb counters, which fit in $O(fb/B) = O(f)$ pages (applying $b = O(B)$). Since the capacity of a leaf node remains $\Theta(B)$, the height of the tree is $O(\log_f(N/B))$, rendering the query cost to be $O(f \log_f(N/B))$.

The value of f is at least 3 (i.e., an internal node has 2 child nodes at minimum). As $f \log_f(N/B)$ is monotonic with f, it is minimized at $f = 3$. In other words, the lowest possible query time complexity of *one-for-all* is $O(\log_3(N/B))$. This, however, is still more expensive than the query/update cost $O(\log_B(N/B))$ of the proposed aBB-tree by a factor of $O(\log_3 B)$. In any case, the update cost is proportional to the height of the tree, namely, $O(\log_f(N/B))$.

The aBB-tree. Underlying the aBB-tree is a B-tree $\mathcal{T}$ that indexes all the items of D (including all categories) by their keys. We refer to $\mathcal{T}$ as the *base tree*. Let u be a node of $\mathcal{T}$. As defined in Section 2, u naturally corresponds to a range $R(u)$ in $\mathbb{R}$. We denote by $D(u) = D \cap R(u)$ as the set of data items stored in the subtree of u. Now, consider u as an internal node with child nodes $v_1, ..., v_B$, listed in such a manner that the items in $D(v_i)$ precede those in $D(v_j)$, for any $1 \leq i < j \leq B$. We refer to v_i as a *left sibling* of v_j. Note that v_i has $i - 1$ left siblings.

Each internal entry e of $\mathcal{T}$ is associated with b counters, one for each color. These counters are defined in a *prefix sum* manner. Specifically, let u be an internal node, and v a child node of u whose parent entry is e. Then, the i-th ($1 \leq i \leq b$) counter of e, denoted as $counter(e)[i]$, equals the total number of color-i items in the subtrees of v and *all the left siblings* of v. As u has up to B entries, it is associated with at most bB counters this way.

The at most bB counters of u are stored separately in $O(b)$ *sequential* pages – referred to as the *counter pages* – such that the b counters (i.e., $counter(e)[i]$ for $i = 1, ..., b$) of each entry e are placed consecutively in ascending order of i. As each counter has a fixed size (i.e., a single word), for any e, we can find the starting page of the counters of e in a single I/O. In other words, once u has been pinpointed, all the b counters of e can be read in $O(1)$ I/Os.

Figure 2 illustrates an example, where the base tree $\mathcal{T}$ has 12 leaf nodes $u_1, ..., u_{12}$, and 4 internal nodes $u_{13}, ..., u_{16}$. The bottom of Figure 2 shows the ranges of all nodes in $\mathcal{T}$ except the root. The underlying dataset D contains items of $b = 2$ colors: white and gray. In the sequel, we refer to white as color 1, and gray as color 2. The white (gray) counter 13 (11) of e_{14} indicates that 13 white (11 gray) items are in the subtrees of e_{13} and e_{14}. Similarly, the white (gray) counter 18 (19) of e_{15} is essentially the total number of white (gray) items in the entire tree.

Space. The base tree $\mathcal{T}$ itself occupies $O(N/B)$ pages, and has $O(N/B^2)$ internal nodes. As each internal node needs $O(b)$ counter pages, all the internal nodes occupy $O(\frac{N}{B^2}b) = O(N/B)$ pages in total. The overall space cost is therefore $O(N/B)$.

Query. Consider a BRC query with search interval $r = [x, y]$ and category preference Q. Denote by $Q[i]$, $1 \leq i \leq |Q|$, the i-th color of Q. To answer the query, we resort to an array res of size $|Q|$, such that $res[i]$ is set to 0 initially, and will equal the number of color-$Q[i]$ data items in r when the query algorithm finishes.

The algorithm first descends two root-to-leaf paths to the leaf nodes z_1 and z_2 such that $R(z_1)$ and $R(z_2)$ contain x and y, respectively. Let u be the lowest common ancestor of z_1 and z_2. Denote by π_1 (resp. π_2) the set of nodes on the path from u to z_1 (resp. z_2). For each internal node v on $\pi_1 \cup \pi_2$, we carry out the following steps to update res. Let $v_1, ..., v_B$ be the child nodes of v.[4] Define:

- j_1 to be the integer such that v_{j_1} is on π_1. If no child of v is on π_1, then $j_1 = 0$.
- j_2 to be the integer such that v_{j_2+1} is on π_2. If no child of v is on π_2, then $j_2 = B$.

If we denote by e_{j_1} and e_{j_2} the parent entries of v_{j_1} and

4. The ordering is such that any item in $D(v_{j_1})$ precedes the entire $D(v_{j_1})$, for all $1 \leq j_1 < j_2 < B$. The same convention applies whenever we list the child nodes of a node.

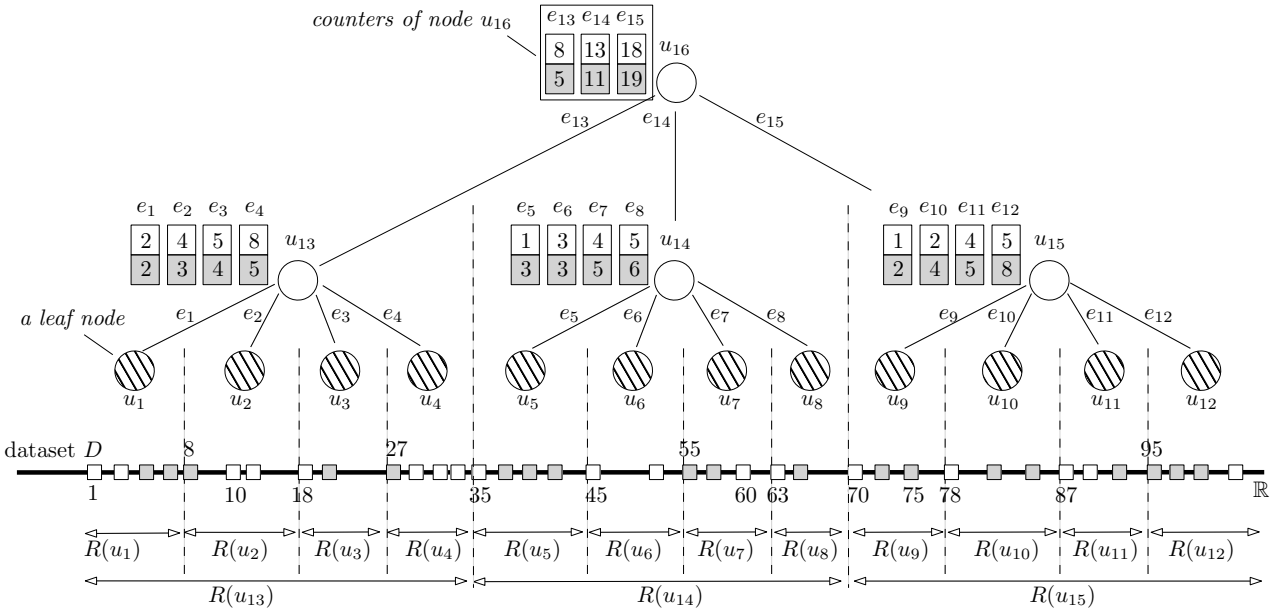


Fig. 2. An aBB-tree

v_{j_2} respectively, then res is updated as:

$$res[i] = res[i] + \begin{cases} counter(e_{j_2})[Q[i]] & \text{if } j_1 = 0 \\ counter(e_{j_2})[Q[i]] - counter(e_{j_1})[Q[i]] & \text{otherwise} \end{cases}$$

for every $i \in [1, |Q|]$. Finally, at z_1 and z_2, we simply go through all the items stored there, and update res accordingly.

For example, given a query with $r = [10, 75]$ and $Q = \{1, 2\}$ on the example of Figure 2, we have $\pi_1 = \{u_{16}, u_{13}, u_2\}$ and $\pi_2 = \{u_{16}, u_{15}, u_9\}$. It is the 5 nodes in $\pi_1 \cup \pi_2$ that are accessed. At u_{16}, for instance, e_{j_1} and e_{j_2} correspond to e_{13} and e_{14}, respectively.

Clearly, there are $O(\log_B N)$ nodes in $\pi_1 \cup \pi_2$. At each internal node $v \in \pi_1 \cup \pi_2$, we spend $O(b/B)$ I/Os to fetch the b counters of e_{j_1}, and those of e_{j_2}. Then, array res can be updated in $O(b/B)$ I/Os[5]. This means that we pay only $O(b/B) = O(1)$ I/Os per internal node. Processing z_1 and z_2 obviously incurs no more than $O(b/B) = O(1)$ I/Os. The overall query time is therefore $O(\log_B N)$.

Remark. Note that $counter(e_{j_1})[Q[i]]$ and $counter(e_{j_2})[Q[i]]$ for $i = 1, ..., |Q|$ can also obviously be retrieved using $|Q|$ I/Os. In this way, the query cost amounts to $O(|Q| \log_B N)$. This means that one can also attain the query performance of *one-for-each* (see Section 1.3) using an aBB-tree. This alternative query algorithm is more efficient when $|Q| < b/B$.

Difficulty of updates. Defining counters in a "prefix sum" style allows us to achieve $O(\log_B N)$ query cost. They, however, make dynamic maintenance of the aBB-tree a challenging issue. To see why, notice that, at each node, inserting into (or deleting from) one branch may invalidate the counters of $O(B)$ other branches. For example, if a white item is inserted in the subtree of e_{13}, then the white counters of e_{13}, e_{14} and e_{15} all need to be increased by 1. When b is large, these counters may be scattered in different pages, so that $O(b)$ I/Os may be needed to update them. As this happens at all internal levels, the total insertion cost would be as high as $O(b \log_B N)$ I/Os! In the next section, we give alternative strategies to improve the update cost to $O(\log_B N)$.

5. These I/Os can be avoided if res fits in memory, which is true in practice. Here, we aim to provide formal description that holds for any $M \geq 2B$.

4 Dynamic Maintenance

In this section, we show how to perform insertions and deletions in the aBB-tree using $O(\log_B N)$ amortized I/Os. This is achieved using a *patching* approach as presented in Section 4.1. The concrete update algorithms are described in Section 4.2. We will focus on BRC retrieval but the extensions to BRS queries are straightforward.

4.1 Patching

Rationale. Recall that each internal node in the base tree $\mathcal{T}$ is associated with $O(bB)$ counters stored in $O(b)$ pages. An obstacle in updating the aBB-tree efficiently, as discussed in Section 3, is that one insertion/deletion may affect $O(B)$ counters at *each* internal level. As those counters may be kept at different pages, immediately modifying them can incur $O(b)$ I/Os at each internal level, resulting in $O(b \log_B N)$ update cost overall.

We avoid such deficiency by updating the counters in a delayed manner. For each internal node u in $\mathcal{T}$, we do not update any of its counters until $\Omega(B)$ insertions/deletions have happened in its subtree. In other

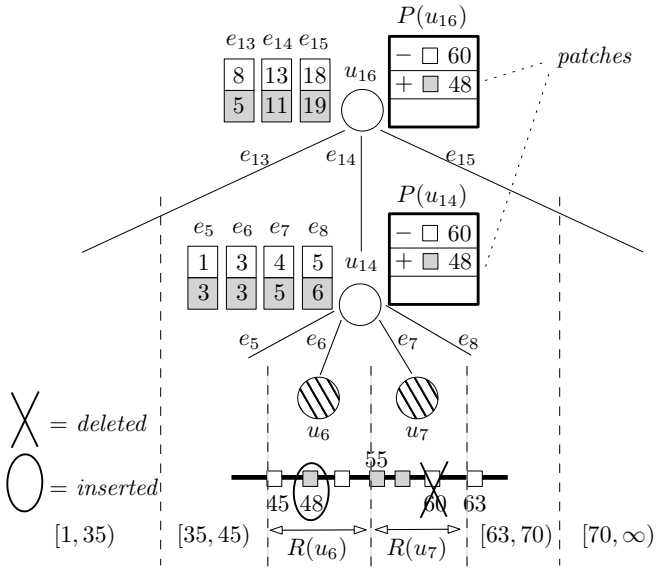
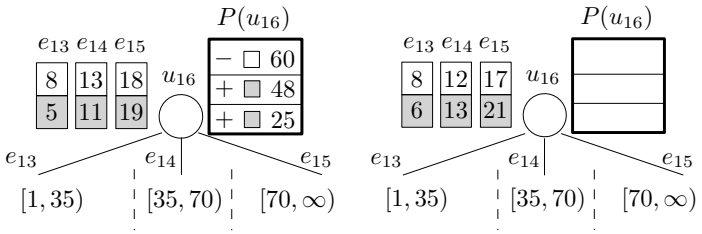


Fig. 3. Illustration of patches



(a) Before the overhaul (b) After the overhaul

Fig. 4. A counter overhaul

words, those counters may stay inaccurate for some time (but precise results are still guaranteed using a method described later). Once $\Omega(B)$ updates have occurred beneath u, we perform a *counter overhaul* (to be elaborated shortly) to correct all the counters of u in $O(b)$ I/Os. The cost of a counter overhaul can be amortized over those $\Omega(B)$ updates, so that each update bears only $O(b/B) = O(1)$ I/Os.

Patch. Motivated by the above, we associate each internal node u with one extra page called the *patch* of u, denoted as $P(u)$. We use $P(u)$ to remember all the items that have been inserted in or deleted from the subtree of u since the last counter overhaul on u. Let $F = \Omega(B)$ be the maximum number of items that can be stored in a page. Once $P(u)$ gets *full* (namely, when it contains F items), a counter overhaul is triggered.

To illustrate, imagine that we delete item 60 from node u_7 in Figure 2, and insert a gray item 48 into node u_6. The relevant parts of the resulting aBB-tree are given in Figure 3, where each internal node is now accompanied by a patch. Each updated item is recorded in the patches of *all* the internal nodes on the path from the root to the leaf containing the item. Hence, the deletion of 60 and insertion of 48 are recorded in the patches of both u_{16} and u_{14}. Note that none of the counters of u_{16} and u_{14} has been altered.

Counter overhaul. Next, we clarify how to perform a counter overhaul when the patch $P(u)$ of node u is full. Let $v_1, ..., v_B$ be the child nodes of B, whose parent entries are $e_1, ..., e_B$, respectively. We pin $P(u)$ in memory, while updating the $O(b)$ counter pages of u one by one. Specifically, after fetching a counter page, we modify all the counters there to their accurate values, and then write the counter page back to the disk. Note that, given a counter $counter(e_j)[i]$ for any $j \in [1, B]$ and $i \in [1, b]$, once its current value has been brought into memory, its accurate value can be obtained without any I/O. For this purpose, we only need to obtain the numbers α and β of insertions and deletions respectively in $P(u)$ that fall into the subtrees of $v_1, ..., v_j$. After this, $counter(e_j)[i]$ can be modified to $counter(e_j)[i] + \alpha - \beta$.

To illustrate, consider that a gray item 25 is inserted in the tree of Figure 3, and hence, is recorded in the patch $P(u_{16})$ of the root u_{16} (see Figure 4a). Assume that a patch can contain up to $F = 3$ items, so $P(u_{16})$ is full, necessitating a counter overhaul on u_{16}. Its child nodes u_{13}, u_{14}, u_{15} have ranges $[1, 35)$, $[35, 70)$, and $[70, \infty)$, respectively (as can also be observed in Figure 2). Let us look at the first record in $P(u_{16})$ (removal of white item 60). As 60 is covered by $R(u_{14}) = [35, 70)$, its removal

should reduce the white counters of e_{14} and e_{15} by 1. Similarly, the second record in $P(u_{16})$ should increase the gray counters of e_{14} and e_{15} by 1, while the last record increases all the 3 gray counters of u_{16} by 1. Figure 4b illustrates the updated counters, and the emptied $P(u_{16})$.

In general, a counter overhaul on node u needs to access u and $P(u)$, and then read/write the $O(b)$ counter pages of u at most once. Hence, it can be finished in $O(b)$ I/Os.

Space. Since (i) each patch occupies only 1 page and (ii) there are $O(N/B^2)$ internal nodes, all the patches occupy only $O(N/B^2)$ space. Hence, the overall space cost of the aBB-tree is still $O(N/B)$.

Query. Our algorithm in Section 3 needs to be slightly modified to account for the fact that, a counter of a node u must be verified (and corrected if necessary) using the items in the patch $P(u)$. Counter verification imposes only 1 extra I/O (for reading the patch) at each internal node accessed. The query cost thus remains $O(\log_B N)$.

To illustrate, consider a query with $r = [10, 75]$ and $Q = \{1\}$ issued on the aBB-tree in Figure 3. At the root u_{16}, the query algorithm needs to retrieve the white counters of e_{13} and e_{14}. The two counters are stored as 8 and 13 respectively in the counter pages of u_{16}, but they may be erroneous due to the updates logged in $P(u_{16})$. To find out, the algorithm reads $P(u_{16})$, and modifies the white counter of e_{14} to 12 according to the first record in $P(u_{16})$. Note that the modification happens only in memory; namely, the counter of e_{14} still remains 13 in disk. The algorithm then proceeds as described in Section 3 with the verified counters (the white counter of e_{13} needs no correction).

4.2 Update algorithms

This subsection elaborates the insertion and deletion algorithms of the aBB-tree. These algorithms extend those of a traditional B-tree with extra steps to maintain the counter pages of the internal nodes in the base tree $\mathcal{T}$. Overall, several principles are followed: (i) $\mathcal{T}$ is updated in exactly the same way as a normal B-tree. (ii) The item being inserted/deleted is automatically recorded in the patches of all the nodes along the insertion/deletion path. (iii) Counter pages are modified only when a patch becomes full (in which case, a counter overhaul is performed), a node is split, or two nodes are merged. As (i) and (ii) are straightforward, and for (iii) the details of a counter overhaul have been given in Section 4.1, what remains unclear is how to update counters in a node split and merge, respectively. Next, we discuss these scenarios respectively.

Split. Figure 5 formally presents the node split algorithm. Next, we illustrate it using an example. Consider that node u_{14} in Figure 6a is to be split. The algorithm starts by *forcing* a counter overhaul on u_{14} and its parent node u_{16} – that is, perform a counter overhaul on them

Algorithm *Split*(u)

/* u is the node being split. This algorithm assumes that u is not the root. Otherwise, the split algorithm is a straightforward adaptation of this one. */

1. $u_{parent} \leftarrow$ the parent node of u
2. force counter overhauls on u and u_{parent}
   /* if u is a leaf node, no counter overhaul on u */
3. $e \leftarrow$ the parent entry of u
4. split u into u' and u'' as in a normal B-tree
   /* assume u' is on the left of u'' */
5. **for each** entry $\hat{e}$ in u'
6. set all counters of $\hat{e}$ in u' directly to those of $\hat{e}$ in u
7. $e^* \leftarrow$ the right most entry in u';
   $e^{**} \leftarrow$ the right most entry in u''
8. **for each** entry $\hat{e}$ in u''
9. **for** $i \leftarrow 1$ to b
10. $counter(\hat{e})[i] \leftarrow counter(e)[i] - counter(e^*)[i]$
11. remove u, and add u' and u'' as children of u_{parent}
12. $e' \leftarrow$ parent entry of u';
    $e'' \leftarrow$ parent entry of u''
13. set all counters of e'' directly to those of e
14. **for** $i \leftarrow 1$ to b
15. $counter(e')[i] \leftarrow counter(e'')[i] - counter(e^{**})[i]$

Fig. 5. Node split algorithm

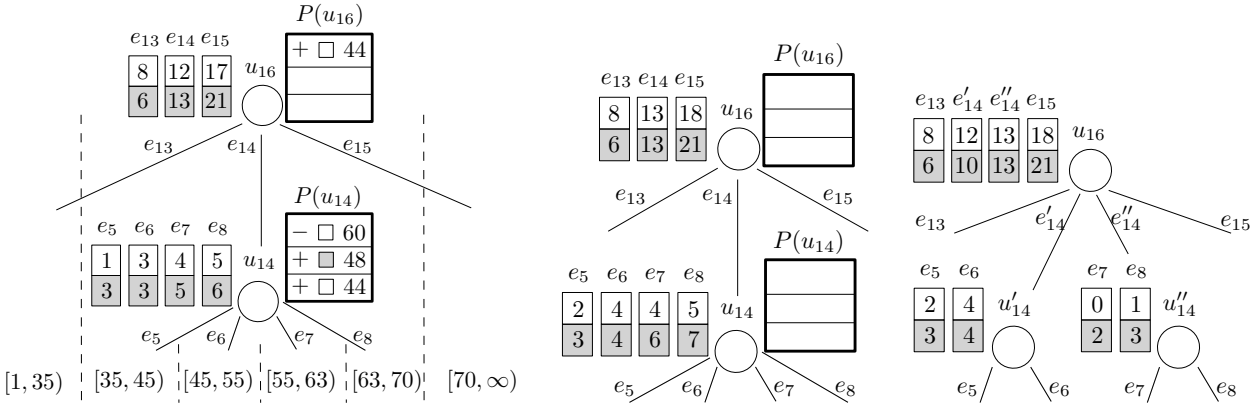
respectively, even if their patches are not full yet. Figure 6b shows the situation after the (forced) overhauls.

Next, we split u_{14} splits into u'_{14} and u''_{14} (in the same way as in a standard B-tree), and decide the counters of the new nodes appropriately. As shown in Figure 6c, the counters of e_5 and e_6 in u'_{14} are copied directly from those in the original node u_{14}, whereas adjustments are needed to derive the counters in u''_{14}. For instance, the white counter of e_7 equals 0 in u''_{14} because, from the white counters of e_6 and e_7 in u_{14}, we can infer that no white item exists in the subtree of e_7. The other counters in u''_{14} are obtained based on analogous reasoning.

The generation of u'_{14} and u''_{14} creates parent entries e'_{14} and e''_{14} in u_{16}. To complete the split, we compute the b counters of e'_{14} and e''_{14}, respectively. As given in Figure 6, the counters of e_{14} (which has disappeared due to the removal of u_{14}) in Figure 6b are taken directly as the counters of e''_{14}. The counters of e'_{14} are calculated as follows: its white (gray) counter 12 (10) is the difference between the white (gray) counter of e''_{14} and that of e_8 in u''_{14}.

In general, a split involves at most 2 counter overhauls, and reading/writing the counter pages of at most 4 nodes. Therefore, its cost is bounded by $O(b)$ I/Os.

Merge. The merge algorithm, formally presented in Figure 7, can be easily understood as reversing the steps in a split. For example, think backwardly of Figures 6c and 6b as nodes u'_{14} and u''_{14} merging into u_{14}. This demands



(a) Node u_{14} is to split (b) After counter overhauls (c) After the split

Fig. 6. Handling a node split

Algorithm ***Merge***(u', u'')

/* u' and u'' are the nodes to be merged. This algorithm assumes that their parent node is not the root. Otherwise, the split algorithm is a straightforward adaptation of this one. */

1. $u_{parent} \leftarrow$ the parent node of u' (hence also u'')
2. force counter overhauls on u', u'', and u_{parent}
   /* if u' and u'' are leaf nodes, no counter overhaul on them */
3. $e' \leftarrow$ the parent entry of u';
   $e'' \leftarrow$ the parent entry of u''
4. $e^* \leftarrow$ the right most entry in u'
5. merge u' and u'' into u as in a normal B-tree
   /* assume u' is on the left of u'' */
6. **for each** entry $\hat{e}$ in u
7. **if** $\hat{e}$ is from u' **then**
8. set all counters of $\hat{e}$ in u directly to those of $\hat{e}$ in u'
9. **else**
10. **for** $i \leftarrow 1$ to b
11. $counter(\hat{e})[i] \leftarrow counter(\hat{e})[i] + counter(e^*)[i]$
12. remove u', u'', and add u as a child of u_{parent}
13. $e \leftarrow$ parent entry of u
14. set all counters of e directly to those of e''

Fig. 7. Node merge algorithm

creating the counter pages of u_{14}, and modifying the counter pages of u_{16}. We omit the details because they should have become straightforward at this point.

Update cost. We prove:

Lemma 1: The aBB-tree can be updated in $O(\log_B N)$ I/Os amortized.

Proof: An update on the aBB-tree includes (i) maintaining the base tree $\mathcal{T}$, (ii) modifying the patches and counter pages of the relevant internal nodes. The cost of (i) is $O(\log_B N)$ I/Os by the standard analysis of the B-tree. As for (ii), if no node split/merge is involved in the update, our algorithm simply adds the item (being inserted/deleted) to the patches of the internal nodes along the insertion/deletion path, and carries out at most one counter overhaul at each internal level. This requires $O(\log_B N)$ amortized I/Os per insertion/deletion (the cost of counter overhauls can be amortized in the way explained in Section 4.1, so that each update accounts for $O(1)$ I/Os at each internal level). In the sequel, we assume that a split or merge has occurred. A charging argument will be used to show that the total cost of handling splits and merges can be amortized over all the updates, such that each update bears $O(\log_B N)$ I/Os.

The B-tree, as implemented in [10], has the property that, after a node u is created (by a split or a merge), it can generate an overflow/underflow only after $\Omega(B)$ insertions or deletions have occurred in its subtree[6]. Refer to the set of those updates as the *pocket set* of u. Notice that every update is in the pocket sets of at most $O(\log_B N)$ nodes (i.e., those on the insertion/deletion path of the update).

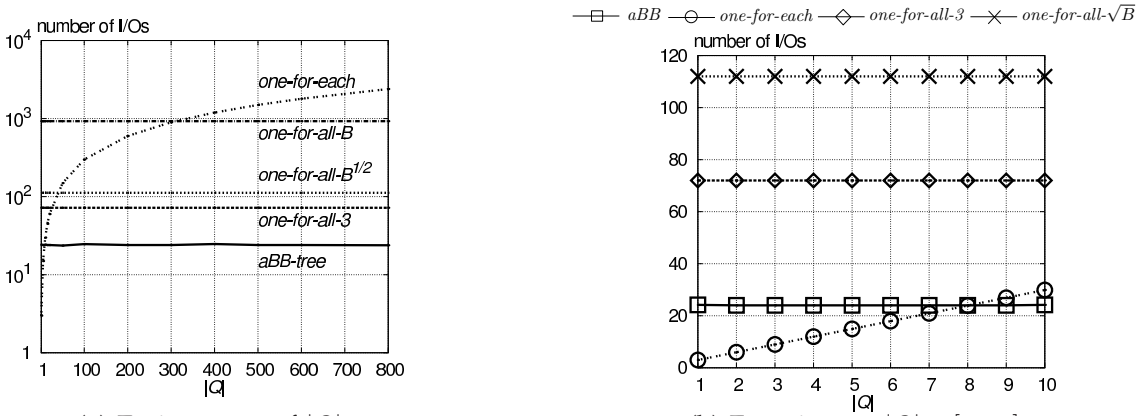
As explained in Section 4.2, our algorithm handles a split (merge) in $O(b)$ I/Os. We charge this cost over the $\Omega(B)$ updates in the pocket set of the node that generated the overflow (underflow). Each update thus bears only $O(b/B) = O(1)$ I/Os. Since an update belongs to the pocket set of $O(\log_B N)$ nodes, the total cost it needs to bear is at most $O(\log_B N)$. □

Main result. We thus have arrived at:

Theorem 1: For $b = O(B)$, there exists a structure that consumes $O(N/B)$ space, and solves any BRS/BRC query in $O(\log_B N)$ I/Os. The structure can be updated with $O(\log_B N)$ amortized I/Os per insertion/deletion.

Remark. Our patching method is reminiscent of the *buffer-tree* technique [3] (see also [5]). The two approaches

6. A simple way to do so is to set the minimum number of entries in a node to, e.g., $B/4$.



(a) Entire range of $|Q|$ (b) Focusing on $|Q| \in [1, 10]$

Fig. 8. Query cost vs. the number $|Q|$ of categories in the preference set Q ($N = 80$m, $b = 800$)

are similar in that both associate internal nodes with additional pages in order to perform updates in a batched manner. The buffer-tree technique, however, aims at progressively pushing updates to lower levels of a tree efficiently. In our context, an update immediately reaches all levels (just like updating a normal B-tree); the purpose of patching is to refresh aggregate information (i.e., the counters), a feature that is absent from [3], [5].

5 EXPERIMENTS

We will compare the aBB-tree to *one-for-all* and *one-for-each*, both of which (as introduced in Section 1.3) are based on aB-trees. Regarding *one-for-all*, we implemented three versions, which differ in the maximum fanouts of internal nodes:

- *one-for-all-3*: The maximum fanout is 3.
- *one-for-all-*$\sqrt{B}$: The maximum fanout is $\sqrt{B}$.
- *one-for-all-*B: The maximum fanout is B.

We evaluated all the methods by their query, update, and space overhead, using both synthetic and real data. The page size was fixed to 4096 bytes.

Data and queries. We generated datasets where items' keys and weights are uniformly distributed in $[0, 2^{30})$ and $[0, 100)$, respectively. Each dataset can be characterized by two parameters: the number b of colors (a.k.a. categories), and the total number N of items. Every item is assigned a random color; hence, each category has approximately the same number of items. In the sequel, we use $u_N v_b$ to denote a dataset with $N = u$ million and $b = v$. For example, $40_N 400_b$ represents a dataset with $N = 40$ million items and $b = 400$ categories. We varied N from 4 to 80 million, and b from 100 to 800.

We also experimented with a real dataset called *S&P500*, which contains the daily trading volumes of every stock in the S&P500 index from 1 Jan. 1980 to 4 Mar. 2010 (if a stock entered the market after 1 Jan. 1980, its entire history is included). Each stock is a category (i.e., totally $b = 500$ categories), in which an item is of the form (*date, volume*), where *date* is the item's key, and

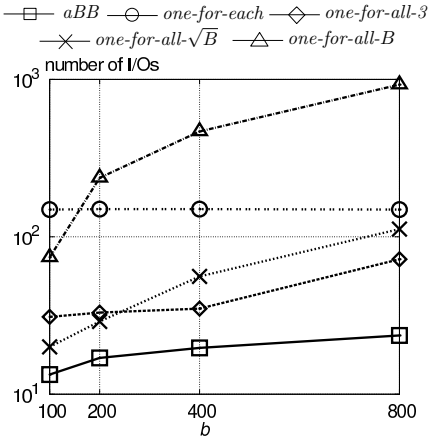


Fig. 9. Query cost vs. the total number of categories b ($N = b \cdot 10^5$, $|Q| = 50$)

volume is its weight. The total number N of items (of all stocks) is 2.57 million.

A *q-workload* is defined to be a set of 100 bundled range sum (BRS) queries whose category preferences Q have q colors (i.e., $|Q| = q$). The search interval r of a query is decided as $[\min\{x, y\}, \max\{x, y\}]$, where x and y are two random integers in the underlying key domain. The Q of the query is a random size-q subset of all the b possible colors. A (q, ℓ)-*workload*, on the other hand, is similar except that the interval r of each query has the same length ℓ, and has its position uniformly decided in the key domain. Note that a query on dataset *S&P500* has the semantics of retrieving the total trading volume during the period of r for each of the stocks in Q. We gauge the query cost of a method as the average number of I/Os it performs in answering a query in the workload. CPU time is ignored because in all cases it accounts for a fraction (less than one thousandth) of the I/O time.

Query characteristics. Let us start by studying the query behavior of each method using synthetic data. The first experiment assesses the impact of q on query efficiency. For this purpose, we used each method to answer q-

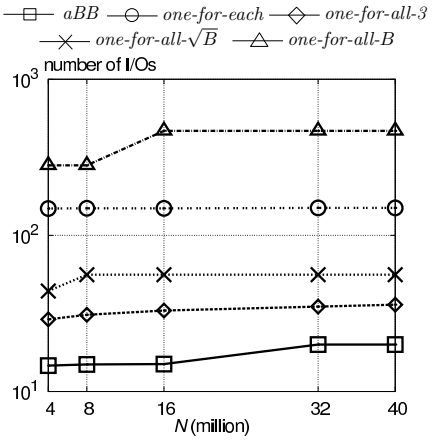


Fig. 10. Query cost vs. the dataset cardinality N ($b = 400$, $|Q| = 50$)

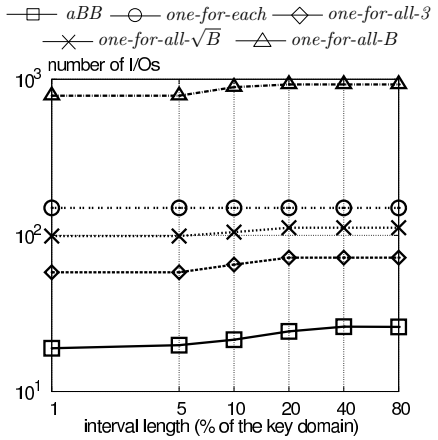


Fig. 11. Query cost vs. the length ℓ of the query interval ($b = 800$, N = 80m, $|Q| = 50$)

workloads on dataset 80_N800_b, by varying q in its entire range (i.e., from 1 to b). Figure 8a illustrates the query overhead as a function of q, while Figure 8b zooms into $q \in [1, 10]$ for better clarity. The performance of the aBB-tree and *one-for-all* was not affected by q, whereas the cost of *one-for-each* escalated linearly with this parameter. In Figure 8, *one-for-each* is outperformed by the aBB-tree at $q = 8$; and the speedup of the aBB-tree increases to two orders of magnitude when q reaches 800. Among the three versions of *one-for-all*, the one with the smallest fanout exhibits the best query efficiency, as explained by our analysis in Section 3. Note that *one-for-all-B* is omitted from Figure 8b due to its prohibitively high query overhead.

The next experiment aims at revealing a defect of *one-for-all*, that is, its query cost scales poorly with the number b of categories. We fixed the number of items in each category to 100k, but increased b from 100 to 800. For each dataset, Figure 9 reports the cost of all methods in processing a 50-workload. The overhead of the aBB-tree and *one-for-each* is stable for all values of b, while the performance of *one-for-all* deteriorates very rapidly. Note that for $b \leq 200$, *one-for-all-*$\sqrt{B}$ outperformed *one-for-all-3*. This phenomenon deserves further explanation because it seems to contradict our claim that $f = 3$ is the best fanout. In fact, this is merely an artifact of asymptotic analysis, which holds when the input parameters (in our case, b) are sufficiently large. Otherwise, for small b, the hidden constant (which is *not* taken into account by big-O) does not permit drawing a decisive conclusion on the superiority of *one-for- all-3* and *one-for-all-*$\sqrt{B}$.

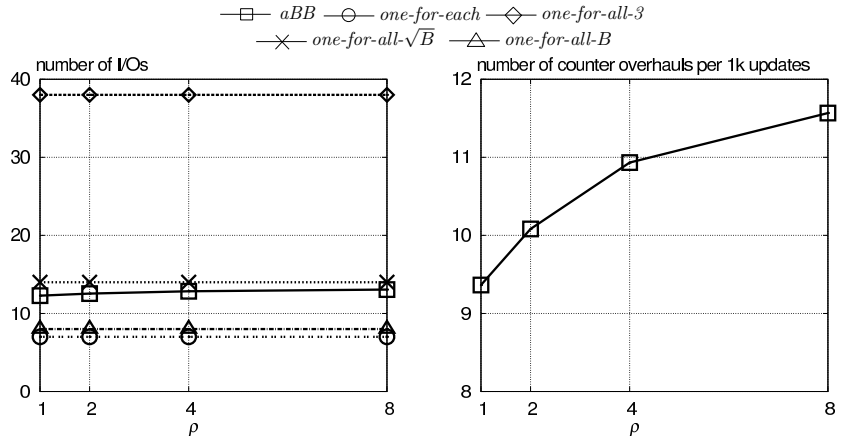
Let us now inspect the scalability of the three approaches with respect to the cardinality N, when the number b of categories is fixed. For this purpose, we set b to 400, and increased N from 4 to 40 million. Again, for each dataset, we compared alternative methods in answering a 50-workload; the results are presented in Figure 10. The cost of all solutions is only slightly affected, because their query complexities are logarithmic to N.

To examine the influence of the length ℓ of the query interval, we deployed dataset 80_N800_b and compared the efficiency of the competing methods on $(50, \ell)$-workloads by varying ℓ from 1% to 80% of the key domain's length. The results, as shown in Figure 11, exhibit the same relative superiority as has been observed in the preceding figures.

Update characteristics. We now proceed to study the update behavior of each method, again using synthetic data. Let us define a *ρ-sequence* as a sequence of updates, in which there are ρ times more insertions than deletions, where ρ is called the *ins-del ratio*. Specifically, each ρ-sequence consists of 1 million updates generated as follows. First, each update is an insertion with probability $\rho/(1 + \rho)$, and a deletion with the remaining probability. Second, an insertion adds to the dataset an item whose key and weight are uniformly distributed in their respective domains, while a deletion removes a random item in the current dataset. The update cost of a method is measured as the average number of I/Os in handling an update in a ρ-sequence.

The first experiment examines the effect of ρ. For this purpose, we obtained four ρ-sequences by doubling ρ from 1 to 8, and applied each sequence to dataset 40_N400_b (e.g., after executing an 8-sequence on 40_N400_b, the dataset cardinality becomes 40.9 million). Figure 12a compares the update cost of all methods as a function of ρ. All methods were able to handled an update in less than 15 I/Os on average, whereas the overhead of *one-for-each-3* was significantly higher. This is expected because *one-for-each-3*, due to its low fanout, has a much greater height.

Let us analyze the cost of the aBB-tree in greater detail. In the above experiment, the height of the aBB-tree remained 4 (including 3 internal levels). It thus follows that



(a) Update cost (b) Frequency of counter overhauls

Fig. 12. Influence of the ins-del ratio ρ on update cost (dataset 4_N400_b)

aBB	*one-for-each*	*one-for-all*
10	3.8	56

(a) Avg. num. of I/Os per update

aBB	*one-for-each*	*one-for-all*
69	42	122

(b) Space (mega bytes)

TABLE 2
Update and space comparison (dataset *S&P500*)

each insertion/deletion required at least 11 I/Os[7] even if no structural change (i.e., node split/merge) happened. In other words, counter overhauls, the most expensive procedure in handling a split/merge, contributed very few amortized I/Os per update. To see why, Figure 12b gives, as a function of ρ, the average number of counter overhauls in every 1000 updates in the experiments of Figure 12a. Evidently, counter overhauls were infrequent. For example, at $\rho = 1$, only less than 10 overhauls took place every 1000 updates. Given that each overhaul performed around 280 I/Os[8], merely $10 \cdot 280/1000 = 2.8$ I/Os were amortized over each update. This is the main reason behind the efficiency of our patching technique in Section 4.1.

Space scalability. Figure 13 plots the space consumption of each technique as a function of N, when the number b of categories is fixed to 400. The space of all methods grew linearly, as predicted by their space complexities. *One-for-each* required the least space because, unlike the other methods, it does not need to store the color (i.e., category id) of any item. *One-for-all-3*, on the other hand, occupied the most space because it has much more internal nodes. The space consumption of the other methods was approximately the same.

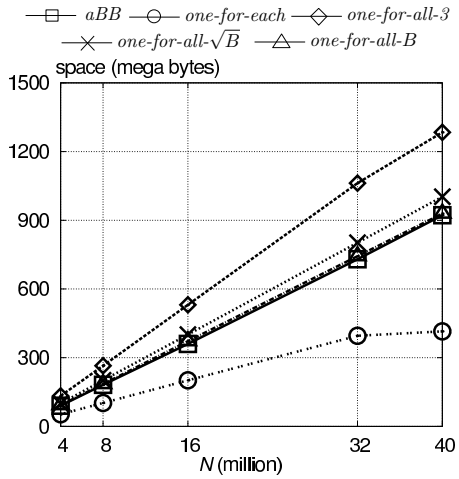


Fig. 13. Space vs. the dataset cardinality N ($b = 400$)

Performance on real data. Finally, we examine the efficiency of the ABB-tree, *one-for-all-3*, and *one-for-each* on the real dataset *S&P500* (*one-for-all-*$\sqrt{B}$ and *one-for-all-B* were not considered because their query cost is much more expensive). Figure 14 compares their query cost when the size $|Q|$ of the category preference varies from 1 to 50. All methods demonstrated the same behavior as in Figure 8. Table 2a gives the average number of I/Os per update in constructing the index of each method on *S&P500*, while Table 2b shows the amount of space occupied by that index. These results are consistent with the earlier observations on synthetic data. It is worth mentioning that the maximum cost of the aBB-tree in handling an update was over 300 I/Os, which again

7. This includes reading two internal nodes (3 I/Os), reading and writing their patches respectively (6 I/Os), and reading and writing a leaf node (2 I/Os).

8. The average fanout of an internal node in an aBB-tree was 357. Hence, on average $357b/1024$ counter pages were associated with an internal node, as a page (of 4k bytes) can store 1024 counters. For $b = 400$, $357b/1024 = 139.4$; and a counter overhaul performed twice as many I/Os (for reading and writing each counter page respectively).

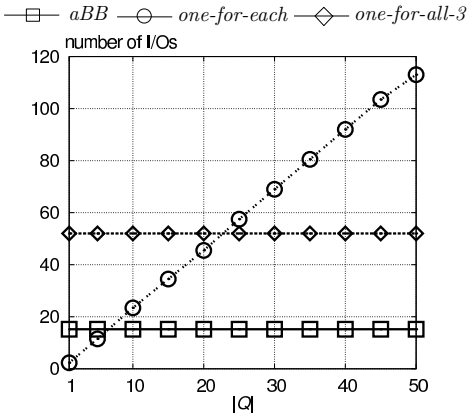


Fig. 14. Query cost vs. the number $|Q|$ of categories in the preference set Q (dataset *S&P500*)

indicates how infrequently counter overhauls occurred.

6 Conclusions

Bundled range aggregation performs a traditional range aggregate query on multiple datasets simultaneously. Its usefulness is reflected by the vast flexibility in selecting the queried datasets, which can be arbitrarily chosen from hundreds or even thousands of candidate datasets in a completely ad-hoc manner. The challenge is to solve the problem with cost significantly lower than answering the query on each chosen dataset separately, while at the same time, allowing updates to be carried out efficiently. Under the assumption that the number b of datasets is $O(B)$, this work settles the problem with a structure called the aggregate bundled B-tree (aBB-tree). The aBB-tree consumes linear space $O(N/B)$, answers any bundled range sum/count/average query in $O(\log_B N)$ I/Os, and supports an insertion/deletion in $O(\log_B N)$ amortized I/Os, where N is the total cardinality of all the datasets, and B is the page size. We have also presented extensive experimental results to demonstrate the practical behavior of the aBB-tree and its superiority over alternative solutions to the problem. We leave it as an open problem to support bundled range aggregation efficiently when b is far greater than B.

Acknowledgements

This work was supported by grants GRF 4165/11, GRF 4164/12, and GRF 4168/13 from HKRGC. The authors would like to thank the anonymous reviewers for their insightful comments.

References

[1] P. K. Agarwal, L. Arge, J. Yang, and K. Yi. I/O-efficient structures for orthogonal range-max and stabbing-max queries. In *Proc. of European Symposium on Algorithms (ESA)*, pages 7–18, 2003.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] S.-J. Chun, C.-W. Chung, J.-H. Lee, and S.-L. Lee. Dynamic update cube for range-sum queries. In *Proc. of Very Large Data Bases (VLDB)*, pages 521–530, 2001.

[5] J. V. den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. of Very Large Data Bases (VLDB)*, pages 406–415, 1997.

[6] S. Geffner, D. Agrawal, A. E. Abbadi, and T. R. Smith. Relative prefix sums: An efficient approach for querying dynamic olap data cubes. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 328–335, 1999.

[7] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *Proc. of International Conference on Database Theory (ICDT)*, pages 143–157, 2003.

[8] P. Gupta, R. Janardan, and M. H. M. Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *J. Algorithms*, 19(2):282–317, 1995.

[9] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *Proc. of ACM Management of Data (SIGMOD)*, pages 73–88, 1997.

[10] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982.

[11] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *ICDE*, pages 166–175, 2002.

[12] C. K. Poon. Optimal range max datacube for fixed dimensions. In *Proc. of International Conference on Database Theory (ICDT)*, pages 158–172, 2003.

[13] N. Sarkas, G. Das, N. Koudas, and A. K. H. Tung. Categorical skylines for streaming data. In *Proc. of ACM Management of Data (SIGMOD)*, pages 239–250, 2008.

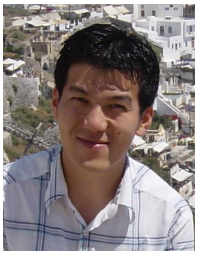
[14] R. R. Schmidt and C. Shahabi. How to evaluate multiple range-sum queries progressively. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 133–141, 2002.

[15] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. E. Hambrusch. Indexing uncertain categorical data. In *Proc. of International Conference on Data Engineering (ICDE)*, pages 616–625, 2007.

[16] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal*, 12(3):262–283, 2003.

[17] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *ACM Transactions on Database Systems (TODS)*, 33(2), 2008.

[18] D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient aggregation over objects with extent. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 121–132, 2002.



Yufei Tao is a full professor at the Chinese University of Hong Kong. He also holds a Visiting Professor position, under the World Class University (WCU) program of the Korean government, at the Korea Advanced Institute of Science and Technology (KAIST). He is an associate editor of ACM Transactions on Database Systems (TODS), and of IEEE Transactions on Knowledge and Data Engineering (TKDE). He is/was a PC co-chair of ICDE 2014, a PC co-chair of SSTD 2011, an area PC chair of ICDE 2011, and a senior PC member of CIKM 2010, 2011, 2012.

Cheng Sheng received his Ph.D. degree in computer science from the Chinese University of Hong Kong in 2012 and BSc degree in computer science from Fudan University in 2008. He is currently a software engineer at Google Switzerland. His research focuses on algorithms in database systems with nontrivial theoretical guarantees.