Eurographics Conference on Visualization (EuroVis) 2021
R. Borgo, G. E. Marai, and T. von Landesberger
(Guest Editors)

Volume 40 (2021), Number 3

Supplementary Material to the Publication Parameterized Splitting of Summed Volume Tables

Christian Reinbold and Rüdiger Westermann

Computer Graphics & Visualization Group, Technische Universität München, Garching, Germany

1. Details of the Heuristic

In our work we did not elaborate on two technical aspects of the heuristic, namely the interpolation scheme to derive the subarray size z and the ratio correlation to partition λ into λ_a and λ_s. We now make up for it so that the heuristic can be reimplemented accurately. Further, we present pseudo code to ease implementation.

1.1. Subarray size

In order to derive the heuristic, we computed sets of optimal parameter trees via an expensive Branch-and-Bound approach for arrays of small shape. When investigating these, we noticed that the subarray size parameter at the root node correlates with the fetch estimate computed for a parameter tree. For instance in Fig 1, all optimal parameter trees for a binary 2D array of shape 64^2 are represented via points (x,y), with x being the fetch estimate and y being the subarray size parameter at the root node. For all investigated array shapes, most points roughly follow a logarithmic curve. We found out that the function

$$f(\lambda) := \frac{\log_2(2\cdot\lambda/(0.35\cdot n_k)+1)}{\log_2(2|n|/n_k+1)},$$

where n is the shape of the input array and k is the split dimension, allows to interpolate between the smallest and largest occurring subarray size z. Clearly, $z \geq 1$. Since one can have up to three subarrays when having only one split position (keep in mind the conjugate trick, see Sec. 4.2), distributed alignment yields maximal subarray sizes of $\lceil (n_k-1)/3 \rceil$. Setting $z = (1-f(\lambda))\cdot 1 + f(\lambda)\cdot\lceil (n_k-1)/3\rceil$ yields the green curve of Fig. 1. After rounding to the nearest subarray shape that can actually arise in distributed aligned splits, we obtain our final estimate of z. The orange curve of Fig. 1 depicts which subarray size is chosen for which fetch estimate. Note that it roughly follows the distribution of optimal parameter trees.

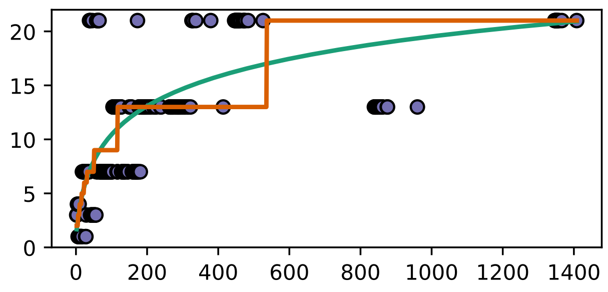

Figure 1: *Scatterplot relating fetch estimates (x-axis) to the subarray size parameter at the root node (y-axis) for all optimal parameter trees for a binary 2D array of shape 64^2. Blue points represent optimal parameter trees. Curves indicate the heuristic used for estimating subarray size from the fetch estimate* ***(green)*** *before and* ***(orange)*** *after discretization.*

1.2. Finding fetch estimates for recursion

Again, by looking at optimal parameter trees, we made the observation that the fetch estimate of the first aggregate subtree can be approximated as well. This is easily seen by considering some ratios. Let T be an optimal parameter tree for an input array of shape n and let T_a be the aggregate subtree of T describing the split hierarchy of an aggregate array of shape $n^{(a)}$. We noticed that

$$\frac{\mathrm{FETCH}(T_a)}{\mathrm{FETCH}(T)} \approx 2\cdot\frac{|n^{(a)}|}{|n|}.$$

In Fig. 2, optimal parameter trees for a binary 64^2-shaped array are represented via points (x,y) where x represents the ratio $|n^{(a)}|/|n|$ and y the ratio $\mathrm{FETCH}(T_a)/\mathrm{FETCH}(T)$. The orange curve hints at the conjectured linear correlation.

Since the control parameter λ of the heuristic represents a threshold for the fetch estimate, it is reasonable that the control parameter λ_a is chosen such that the conjectured correlation holds true. This can be achieved by setting $\lambda_a := 2\cdot\ell/n_k\cdot\lambda$, where ℓ is the number of split positions. It can be computed from the subarray size z via $\ell = \lceil (n_k - z)/(2\cdot z+1)\rceil$. As a last step, we round λ_a to the nearest integral number and clamp it to the interval $[1, \min(|n^{(a)}|,\ \lambda-1)]$. This ensures a) that λ_a cannot surpass the maximal amount of $|n^{(a)}|$ fetches for the aggregate array, and b) that both λ_a and the number of fetches $\lambda_s := \lambda - \lambda_a$ that remain for the subarrays is at least one. Algorithm 1 depicts the pseudo code of the heuristic after incorporating the considerations of this section.

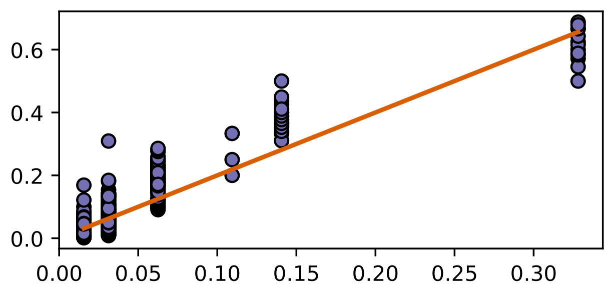

Figure 2: *Scatterplot relating the ratio $|n^{(a)}|/|n|$ (x-axis) to the ratio $\mathrm{FETCH}(T_a)/\mathrm{FETCH}(T)$ (y-axis) for all optimal parameter trees for a binary 2D array of shape 64^2. Blue points represent optimal parameter trees. The orange curve indicates the conjectured linear correlation between both ratios.*

2. A tighter bound for fetch operations

Although the recursive formula for estimating the fetch operations in Sec. 5.2 is a useful tool for deriving the heuristic, it is not the actual number we are interested in. Instead, we wish to know the maximal number of fetch operations we require when querying *an arbitrary* prefix sum. Eq. (3) overestimates this number since it assumes that one has to query a prefix sum from the full aggregate array and a full subarray simultaneously to get the recursion going. This is not quite true in the scenario of Fig. 3 which—for demonstration purposes—does not utilize the trick of conjugating subarrays. Here, we have $\mathrm{FETCH}(T_a) = \mathrm{FETCH}(T_{s_1}) = 2$ and $\mathrm{FETCH}(T_{s_2}) = 1$. $\mathrm{FETCH}(T)$ evaluates to 4. In the estimate formulation it is assumed that whenever we query into a subarray of size 2, we have to query both aggregates as well. However, we only have to query aggregates at *preceding* split positions. Thus, the second aggregate entry never is fetched as it is not followed by a subarray of size 2. Consequently, querying into one of the first two subarrays requires at most 3 fetches. It is only in the case of querying into the last subarray of size 1 that we require both aggregate entries. But then querying the smaller subarray requires only one fetch, summing up to 3 as well. If we query into an aggregate array instead of a subarray, one would require at most 2 fetches. As a result, one requires at most 3 fetch operations instead of 4 as indicated by the fetch estimate.

In order to obtain a tighter bound, we have to make sure that we only pair fetch operations of a subarray with the fetch operations that are required to evaluate prefix sums of the aggregate array up to the *preceding* split position; or up to the *subsequent* split position in the case of a conjugated subarrays. To resolve this issue, we introduce a more complex recursion formula $P(T,v)$ that takes a parameter tree T and a corner $v \in \mathbb{N}^d$, and then returns an upper bound for fetch operations required to retrieve any prefix sum over a subarray contained in the *box of* v, that is the volume spanned by the origin and the corner v. Any box spanned by the origin and a point in the box of v is called a *subbox of* v.

Algorithm 1: heuristic for approximately solving the optimal parameter tree problem.

```
1  Function $\mathcal{H}$(control parameter $\lambda$, array shape $n \in \mathbb{N}^d$):
       Output: Parameter tree $T$ with at most $\lambda$ fetches for
               array of shape $n$
2      $T \leftarrow$ new Node();
       /* Covering leaf cases                    */
3      if $\lambda = 1$ then                          // SVT block
4          attach parameter $I = \{1,2,\ldots,d\}$ to $T$;
5          return $T$;
6      else if $\lambda \geq |n|$ then               // verbatim block
7          attach parameter $I = \emptyset$ to $T$;
8          return $T$;
9      end
       /* Defining split parameters              */
10     $k \leftarrow \arg\max_i(n_i)$;
11     $f \leftarrow \log_2(2\cdot\lambda/(0.35 n_k)+1)/\log_2(2|n|/n_k+1)$;
12     $z_{\text{continuous}} \leftarrow (1-f)\cdot 1 + f\cdot\lceil(n_k-1)/3\rceil$;
13     $z \leftarrow$ round $z_{\text{continuous}}$ to the nearest subarray size that
         can occur with distributed aligned splits along a split
         dimension of length $n_k$;
14     attach parameters $k$, $z$ to $T$;
       /* Creating subtrees via recursion        */
15     $\ell \leftarrow \lceil(n_k-z)/(2\cdot z+1)\rceil$;
16     $n^{(a)} \leftarrow n|_{k=\ell}$;
17     $n^{(s_1)} \leftarrow n|_{k=z}$;
18     $n^{(s_2)} \leftarrow n|_{k=z-1}$;
19     $\lambda_a \leftarrow \mathrm{round}(2\cdot\ell/n_k\cdot\lambda)$;
20     Clamp $\lambda_a$ to $[1, \min(|n^{(a)}|, \lambda-1)]$;
21     $\lambda_s \leftarrow \lambda - \lambda_a$;
22     $T_a \leftarrow \mathcal{H}(\lambda_a,\ n^{(a)})$;
23     $T_{s_1} \leftarrow \mathcal{H}(\lambda_s,\ n^{(s_1)})$;
24     $T_{s_2} \leftarrow \mathcal{H}(\lambda_s,\ n^{(s_2)})$;
25     Attach $T_a$, $T_{s_1}$ and $T_{s_2}$ to $T$;
26     return $T$;
```

2.1. Leaf nodes

If T consists of a single leaf node, evaluation of $P(T,v)$ is straightforward. We just have to compute the number of entries along the axes for which values have not been cumulated before storing them. Hence, we read off the I parameter of the leaf node and set

$$P(T,v) = \prod_{i\in\{1,\ldots,d\}\setminus I} v_i.$$

2.2. Split nodes

If the root of T is an internal node, computations become more involved. We differentiate between upper bounds of fetches for each subarray shape $n^{(s)}$ arising in the split operation via an operation $P_s(T,v,n^{(s)})$. Similarly, we define a function $P_a(T,v)$ that returns an upper bound of fetches if one queries into an aggregate array.

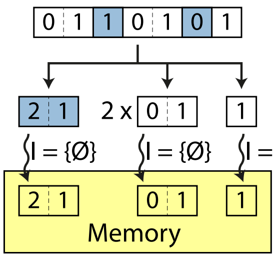

Figure 3: *Decomposition of a 1D array of size 7 according to a parameter tree describing a single split with subarray size 2. All entries at leaf nodes are stored verbatim.*

Clearly, $P(T,v)$ can be computed by taking the maximum over the value of P_a and the value of P_s for each subarray shape. In the scenario of Fig. 3, P_s evaluates to 3 for both subarray shapes and P_a evaluates to 2. Thus, P evaluates to the maximum of 3 as well.

From now on let us assume that T is an internal node with an aggregate subtree T_a and one or more subarray subtrees T_s^i, $i \in \{1,2,\dots,\#\text{subarray shapes}\}$. The split dimension of T is denoted by k. Further we assume a fixed array F of shape n that splits into an aggregate subarray F_a of shape n_a and subarrays F_s^i of different shapes n_s^i such that the subarray subtree T_s^i relates to shape n_s^i.

2.2.1. Query to aggregate arrays

$P_a(T,v)$ is computed as follows: First, the index of the last split position contained in the box of v is computed. In the notation of Sec. 4 of the paper this is $i = \max(\{m \mid c_m \leq v_k\} \cup \{0\})$. By Eq. (1), the box of $v\mid_{k=i}$ resembles the region in F_a that has to be queried in order to compute a prefix sum of a subbox of v in F. Since we do query into an aggregate array, the subarray offset j of Eq. (1) and thus the number of fetches to evaluate the second summand in Eq. (1) always is 0. By the definition of P, we can set

$$P_a(T,v) = P(T_a, v\mid_{k=i}).$$

Note that P now depends on P_a, which itself depends on P again, but invoked on a subtree. Hence, we have setup a recursion formula.

Now we address the evaluation of $P_s(T,v,n_s^i)$ for a subarray shape n_s^i. We simplify notation by fixing the i for which P_s is evaluated and leave out the superscript, that is $F_s \mathrel{\hat{=}} F_s^i$, $n_s \mathrel{\hat{=}} n_s^i$ and so on. A subarray $\mathcal{S}$ is called *complete*, if all its entries projected to the split dimension are contained in the box of v. In other words, $\mathcal{S}$ is complete if and only if for each voxel w covered by $\mathcal{S}$ it holds that $w_k \leq v_k$.

2.2.2. Query to complete subarrays

First, we consider the case of querying into a *complete* subarray of shape n_s. W.l.o.g. we can assume that the queried subarray is the "last" complete subarray $\mathcal{S}_{\text{last}}$ which is most distant from the origin and thus has the highest number of preceding split positions. If we would query into another subarray $\mathcal{S}$, we can query into $\mathcal{S}_{\text{last}}$ instead and would obtain an upper bound at least as high as when querying into $\mathcal{S}$. The upper bound of fetches for the second summand of Eq. (1) remains the same, and the upper bound of fetches for the first summand can only become bigger to due more preceding split positions. Now, we set i to the index of the split position preceding $\mathcal{S}_{\text{last}}$. If $\mathcal{S}_{\text{last}}$ is *not* going to be conjugated, we compute

$$p_{\text{complete}} = P(T_a, v\mid_{k=i}) + P(T_s, v\mid_{k=j}),$$

where $j := (n_s)_k$ is the subarray size along the split dimension. If $\mathcal{S}_{\text{last}}$ is going to be conjugated, Eq. (2) instead of Eq. (1) is used for prefix sum computation, which is why we then have

$$p_{\text{complete}} = P(T_a, v\mid_{k=i+1}) + P(T_a, v\mid_{k=j})$$

instead. Note that i is incremented by one.

2.2.3. Query to incomplete subarrays

Second, we discuss the case of querying into an incomplete subarray of shape n_s, if there is any. Clearly, there can be only one of them and it has to follow the last complete subarray $\mathcal{S}_{\text{last}}$. Let us call it $\mathcal{S}_{\text{inc}}$ and, again, denote by i the index of its preceding split position. In comparison to the complete case, we cannot set $j := (n_s)_k$ anymore, but have to keep in mind that we query only those voxels of $\mathcal{S}_{\text{inc}}$ that are also contained in the box of v. Hence, we set j to the size along the split dimension obtained after intersecting $\mathcal{S}_{\text{inc}}$ with the box of v. If $\mathcal{S}_{\text{inc}}$ is *not* going to be conjugated, then it holds that

$$p_{\text{incomplete}} = P(T_a, v\mid_{k=i}) + P(T_s, v\mid_{k=j}).$$

If, on the other hand, $\mathcal{S}_{\text{inc}}$ is going to be used as a conjugated subarray in Eq. (2), we have to keep in mind that even if the width j is only one, we have to sum over the whole subarray due to flipping. Hence, then we have

$$p_{\text{incomplete}} = P(T_a, v\mid_{k=i+1}) + P(T_a, v\mid_{k=(n_s)_k}).$$

If there exists no incomplete subarrays at all, we set $p_{\text{incomplete}} = 0$. With both complete and incomplete subarrays covered, we can compute P_s via

$$P_s(T,v,n_s) = \max(p_{\text{complete}},\ p_{\text{incomplete}}).$$

2.2.4. Performance considerations

This concludes the formulation of the recursion formula. At this point, formally proving its upper bound property is a mere exercise in precise mathematical formulation and rephrasing of the sections above in a definition-theorem-proof style. However, we still have to improve performance. If j is the number of different subarray shapes (in our case $j = 2$), P has to invoke itself $2j+1$ times with the aggregate subtree and $2j$ times with a subarray subtree. Since this factors multiply for each level of the tree, computation times quickly get out of hand. Performance can be improved significantly by introducing a cache storing results of previous evaluations of P. In doing so, the algorithm even can be used in the bisection method that tweaks the control parameter λ of the heuristic. Timings given in Table 2 include the costs for repeatedly computing the upper bound as proposed here. Further, note that performance does not depend on the actual array size because subarray shapes as well as indices i, j can be derived from split parameters and the input array shape in constant time. Pseudo code for the complete implementation is shown in Algorithm 2.

Algorithm 2: Computation of a tighter bound for fetch operations respecting queries into subboxes.

```
Input: Parameter tree $T$,
       index $v \in \mathbb{N}^d$ into the input array,
       (Optional) cache datastructure $\mathcal{C}$
Output: Upper bound $P(T,v)$ for fetch operations required
        to retrieve prefix sums over any subbox of $v$ when
        using the SVT representation defined by $T$
Initialize empty cache $\mathcal{C}$ if not provided;
if $(T,v) \in \mathcal{C}$ then
    return cached result $P(T,v)$;
end
if root node of $T$ is leaf then
    Read off $I$ parameter from root of $T$;
    result $\leftarrow \prod_{i \in \{1,\ldots,d\} \setminus I} v_i$;
    Insert result into $\mathcal{C}$ at position $(T,v)$;
    return result;
else
    Read off split dimension index $k$ from root of $T$;
    $i \leftarrow$ index of last split position contained in the box of $v$;
    upper $\leftarrow P(T_a, v \mid_{k=i}, \mathcal{C})$;
    foreach subarray subtree $T_s$ of $T$ do
        $n_s \leftarrow$ array shape associated to $T_s$;
        Find last complete subarray $\mathcal{S}_{\text{last}}$;
        $i \leftarrow$ index of last split position preceding $\mathcal{S}_{\text{last}}$;
        if $\mathcal{S}_{\text{last}}$ is conjugated then
            $p \leftarrow P(T_a, v \mid_{k=i+1}, \mathcal{C}) + P(T_a, v \mid_{k=(n_s)_k}, \mathcal{C})$;
        else
            $p \leftarrow P(T_a, v \mid_{k=i}, \mathcal{C}) + P(T_a, v \mid_{k=(n_s)_k}, \mathcal{C})$;
        end
        upper $\leftarrow \max(\text{upper}, p)$;
        Search for incomplete subarray $\mathcal{S}_{\text{inc}}$;
        if $\mathcal{S}_{\text{inc}}$ exists then
            $i \leftarrow$ index of last split position preceding $\mathcal{S}_{\text{inc}}$;
            $j \leftarrow$ size along split dimension $k$ of intersection
              of $\mathcal{S}_{\text{inc}}$ and the box of $v$;
            if $\mathcal{S}_{\text{inc}}$ is conjugated then
                $p \leftarrow P(T_a, v \mid_{k=i+1}, \mathcal{C}) +$
                      $P(T_s, v \mid_{k=(n_s)_k}, \mathcal{C})$;
            else
                $p \leftarrow P(T_a, v \mid_{k=i}, \mathcal{C}) + P(T_s, v \mid_{k=j}, \mathcal{C})$;
            end
            upper $\leftarrow \max(\text{upper}, p)$;
        end
    end
    Insert upper into $\mathcal{C}$ at position $(T,v)$;
    return upper;
end
```

3. SVT representations store partial sums

When discussing construction costs of the proposed SVT representations in Sec. 5.4, we assume that each value stored in memory is a partial sum of the input array F and thus can be sampled from a SVT of F. This claim is intuitive insofar as values of aggregate arrays are obtained by summing consecutive values of the array being split. In the following, we give a formal proof of this claim in Theorem 3.6. However, we require some additional notation first.

Definition 3.1. Let $c = (c_1, c_2, \ldots, c_n)$ be a sequence of n natural numbers. We denote by $|c|$ its length n. For another sequence d of natural numbers such that $d_i \leq n$ for all $i \in \mathbb{N}_{\leq |d|}$, the concatenation $c \circ d$ of c and d is the sequence of length $|d|$ given by $(c \circ d)_i = c_{d_i}$. For $m \in \mathbb{N}_{<|c|}$, the sequence $c \ll m$ of length $|c| - m$ is given by $(c \ll m)_i = c_{i+m}$.

Definition 3.2. Let c be a monotone sequence of natural numbers. The range of c at index $i \in \mathbb{N}_{\leq |c|-1}$ is

$$\text{range}(c,i) = \begin{cases} \{n \in \mathbb{N} \mid c_i < n \leq c_{i+1}\} & \text{if c is increasing} \\ \{n \in \mathbb{N} \mid c_{i+1} < n \leq c_i\} & \text{if c is decreasing.} \end{cases}$$

Lemma 3.3. *Let c be a monotone sequence of natural numbers and d be an* ***increasing*** *sequence of natural numbers such that $c \circ d$ is defined. Then for all $i \in \mathbb{N}_{\leq |d|}$ and $m \in \mathbb{N}_0$ it holds that*

$$\bigcup_{j=d_i}^{d_{i+1}-1} \text{range}(c,j) = \text{range}(c \circ d, i)$$

$$\text{range}(c, i+m) = \text{range}(c \ll m, i)$$

Proof. Follows immediately from the definition of range. □

Definition 3.4. Let c be a finite (that is $|c| < \infty$) monotone sequence of natural numbers with. The inverse $\neg c$ of c is the monotone sequence given by $(\neg c)_i = c_{|c|+1-i}$.

Lemma 3.5. *Let c be a finite monotone sequence of natural numbers. Then for all $i \in \mathbb{N}_{\leq |c|-1}$ it holds that*

$$\text{range}(\neg c, i) = \text{range}(c, |c| - i).$$

Proof. Follows immediately from the definition of the inverse and range. □

Theorem 3.6. *Let A be an array of shape $n \in \mathbb{N}^d$ arising in a split hierarchy of the input array F. Then there exist d monotone sequences $a^{(1)}, \ldots, a^{(d)}$ of length $|a^{(i)}| = n_i + 1$ such that*

$$A[v] = \sum_{v'_i \in \text{range}(a^{(i)}, v_i)} F[v'],$$

where $v \in \mathbb{N}^d$ is any multi index indexing into A. In particular, $A[v]$ is a partial sum of a hyperbox of F that starts at the corner

$$(\min(a^{(1)}_{v_1}, a^{(1)}_{v_1+1}) + 1, \ldots, \min(a^{(d)}_{v_d}, a^{(d)}_{v_d+1}) + 1)$$

and ends at

$$(\max(a^{(1)}_{v_1}, a^{(1)}_{v_1+1}), \ldots, \max(a^{(d)}_{v_d}, a^{(d)}_{v_d+1}))$$

Proof. The second claim immediately follows from the definition of range. The first claim is proven via induction regarding the number of performed split operations before arriving at A. For the initial case, $A = F$ and the claim holds true by setting $a^{(i)} = (0, 1, \ldots, n_i)$.

Next, the inductive step is shown. Assume that A arises by splitting the array B of shape $m \in \mathbb{N}^d$ along split dimension k and split positions

$$0 = c_0 < c_1 < \cdots < c_\ell \leq m_k$$

with the array B being split. By the induction hypothesis, B can be written as

$$B[v] = \sum_{v'_i \in \mathrm{range}(b^{(i)}, v_i)} F[v']$$

with appropriate monotone sequences $b^{(i)}$.

Case 1: A is the aggregate array returned by the split process. We define the monotone sequence d of length $\ell+1$ via $d_m = c_{m-1}+1$. Then, by Lemma 3.3, we have

$$\begin{aligned} A[v] &= \sum_{i=c_{v_k-1}+1}^{c_{v_k}} B[v\,|_{k=i}] \\ &= \sum_{i=d_{v_k}}^{d_{v_k+1}-1} \sum_{v'_k \in \mathrm{range}(b^{(k)}, i)} \sum_{\substack{v'_j \in \mathrm{range}(d^{(j)}, v_j) \\ j \neq k}} F[v'] \\ &= \sum_{v'_k \in \mathrm{range}(b^{(k)} \circ d, v_k)} \sum_{\substack{v'_j \in \mathrm{range}(d^{(j)}, v_j) \\ j \neq k}} F[v'] \end{aligned}$$

Thus, the claim holds true by setting $a^{(k)} = b^{(k)} \circ d$ and $a^{(j)} = b^{(j)}$ for all $j \neq k$.

Case 2: A is a non-conjugated subarray returned by the split process. Choose i such that c_i is the last split position preceding A. By Lemma 3.3, it is

$$\begin{aligned} A[v] &= B[v\,|_{k=c_i+v_k}] \\ &= \sum_{v'_k \in \mathrm{range}(b^{(k)}, c_i+v_k)} \sum_{\substack{v'_j \in \mathrm{range}(b^{(j)}, v_j) \\ j \neq k}} F[v'] \\ &= \sum_{v'_k \in \mathrm{range}(b^{(k)} \ll c_i, v_k)} \sum_{\substack{v'_j \in \mathrm{range}(b^{(j)}, v_j) \\ j \neq k}} F[v'] \end{aligned}$$

Again, the claim holds true by setting $a^{(k)} = b^{(k)} \ll c_i$ and $a^{(j)} = b^{(j)}$ for all $j \neq k$.

Case 3: A is a conjugated subarray returned by the split process. Choose i such that c_i is the first split position succeeding A. By Lemma 3.3 and 3.5, it is

$$\begin{aligned} A[v] &= B[v\,|_{k=c_i+1-v_k}] \\ &= \sum_{v'_k \in \mathrm{range}(b^{(k)}, c_i+1-v_k)} \sum_{\substack{v'_j \in \mathrm{range}(b^{(j)}, v_j) \\ j \neq k}} F[v'] \\ &= \sum_{v'_k \in \mathrm{range}(\neg b^{(k)}, m_k-c_i+v_k)} \sum_{\substack{v'_j \in \mathrm{range}(b^{(j)}, v_j) \\ j \neq k}} F[v'] \\ &= \sum_{v'_k \in \mathrm{range}(\neg b^{(k)} \ll (m_k-c_i), v_k)} \sum_{\substack{v'_j \in \mathrm{range}(b^{(j)}, v_j) \\ j \neq k}} F[v'] \end{aligned}$$

The claim holds true by setting $a^{(k)} = \neg b^{(k)} \ll (m_k - c_i)$ and $a^{(j)} = b^{(j)}$ for all $j \neq k$. Note that $m_k \geq c_i$. □