Optimal Splitters for Temporal and Multi-version Databases

Wangchao Le[1] Feifei Li[1] Yufei Tao[2,3] Robert Christensen[1]
[1]University of Utah [2]Chinese University of Hong Kong [3]Korea Advanced Institute of Science and Technology
{lew, lifeifei}@cs.utah.edu taoyf@cse.cuhk.edu.hk robertc@eng.utah.edu

ABSTRACT

Temporal and multi-version databases are ideal candidates for a distributed store, which offers large storage space, and parallel and distributed processing power from a cluster of (commodity) machines. A key challenge is to achieve a good load balancing algorithm for storage and processing of these data, which is done by partitioning the database. We introduce the concept of *optimal splitters* for temporal and multi-version databases, which induce a partition of the input data set, and guarantee that the size of the maximum bucket be minimized among all possible configurations, given a budget for the desired number of buckets. We design efficient methods for memory- and disk-resident data respectively, and show that they significantly outperform competing baseline methods both theoretically and empirically on large real data sets.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management – *Systems*

Keywords

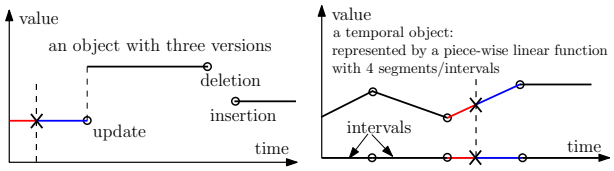
Optimal splitters, temporal data, multi-version databases

1. INTRODUCTION

Increasingly, user applications request the storage and processing of historical values in a database, to support various auditing, provenance, mining, and querying operations for better decision making. Given the fast development in distributed and parallel processing frameworks, applications can now afford collecting, storing, and processing large amounts of multi-versioned or temporal values from a long-running history. Naturally, it leads to the development of multi-version databases and temporal databases. In these databases, an object o is associated with multiple disjoint temporal intervals in the form $[s, e]$, each of which is associated with the valid value(s) of o during the period of $[s, e]$.

Consider two specific examples as shown in Figure 1. Figure 1(a) shows a multi-version database [6, 17], and Figure 1(b) shows a temporal database, where each object is represented by a piecewise linear function. A multi-version database keeps all the historical values of an object. A new interval with a new value is created whenever an update or an insertion to an object has occurred. An existing interval with a (now) old value terminates when an item has been deleted or updated.

On the other hand, for large temporal or time-series data, we can represent the value of a temporal object as an arbitrary function $f : \mathbb{R} \rightarrow \mathbb{R}$ (time to value). In general, for arbitrary temporal data, f can be expensive to describe and process. In practice, applications often approximate f using a piecewise linear function g [3, 7, 13]. The problem of approximating an arbitrary function f by a piecewise linear function g has been extensively studied (see [3, 7, 13, 19] and references therein). Other functions can be used for approximation as well, such as a piece-wise polynomial function, for better approximation quality. The key observations are: 1) more intervals lead to better approximation quality, but also are more expensive to represent; 2) adaptive methods, by allocating more intervals to regions of high volatility and less to smoother regions, are better than non-adaptive methods with a fixed segmentation interval.



(a) Multi-version database. (b) Temporal database.

Figure 1: Databases with Intervals.

How to approximate f with g is beyond the scope of this paper, and we assume that the data has already been converted to a set of intervals, where each interval is associated with a mapping function (piece-wise linear, or piece-wise polynomial) by *any* segmentation method. In particular, we require *neither* them having the same number of intervals *nor* them having the aligned starting/ending time instances for intervals from different objects. Thus it is possible that the data is collected from a variety of sources after each applying different preprocessing modules.

Lastly, large interval data may also be produced by any time-based or range-based partitioning of an object, such as a log file or a spatial object, from a big data set.

That said, we observe that in the aforementioned applications users often have to deal with big interval data. Meanwhile, storing and processing big data in a cluster of (commodity) machines, to tap the power of parallel and distributed computation, is becoming increasingly important. Therefore, storing and processing the large number of intervals in a distributed store is a paramount concern, in enabling the above applications to leverage the storage space and the computation power from a cluster.

Since most analytical tasks and user queries concerning interval data in a multi-version or temporal database are time-based, e.g.,

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

find the object ids with valid values in $[50, 100]$ at time-instance t, the general intuition is to partition the input data set into a set of buckets based on their time-stamps. Intervals from one bucket are then stored in one node and processed by one core from a cluster of (commodity) machines. By doing so, user queries or transactions concerning a time instance or a time range can be answered in selected node(s) and core(s) independently without incurring excessive communication, which also dramatically improves the spatial and temporal locality of caching in query processing.

A challenge is to achieve load-balancing in this process, i.e., no single node and core should be responsible for storing and processing too many intervals. In particular, *given the number of buckets to create, the size of the maximum bucket should be minimized.* This is similar to the concept of *optimal splitters* in databases with points [21] or array data sets [14, 18]. However, the particular nature of interval data sets introduces significant new challenges.

Specifically, a partitioning boundary (known as a splitter) may split an interval into two intervals. As a result, buckets on both sides of a splitter need to contain an interval that intersects with a splitter; see examples in Figure 1 where the dashed line with a cross represents a splitter. Furthermore, intervals from different objects may overlap with each other, which complicates the problem of finding the optimal splitters. Finally, a good storage scheme should also be capable to handle ad-hoc updates gracefully. In contrast, in a point or array data set, any element from the original data set will only lead to one element in one of the buckets; and elements do not overlap with each other.

Our contributions. Given n objects with a total of N intervals from all objects, and a user-defined budget k for the number of buckets to create, a baseline solution is a dynamic programming formulation with a cost of $O(kN^2)$. However, this solution is clearly not scalable for large data sets. Our goal is to design I/O and computation efficient algorithms that work well regardless if data fits in main memory or not. A design principle we have followed is to leverage on existing indexing structures whenever possible (so these methods can be easily adopted in practice). Specifically, we make the following contributions:

- We formulate the problem of finding optimal splitters for large interval data in Section 2.
- We present a baseline method using a dynamic programming formulation in Section 3.
- We design efficient methods for memory-resident data in Section 4. Our best method finds optimal splitters of a data set for any budget values k in only $O(N \log N)$ cost.
- We investigate external memory methods for large disk-based data in Section 5. Our best method finds optimal splitters of a data set for any budget value k in only $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ IOs, where B is the block size and M is the memory size.
- We extend our methods to work for the queryable version of the optimal splitters problem in Section 6, where k is the query parameter. We also discuss how to make our methods dynamic to handle ad-hoc updates in Section 6.
- We examine the efficiency of the proposed methods with extensive experiments in Section 7, where large real data sets with hundreds of millions of intervals were tested.

We survey related work in Section 8, and conclude in Section 9.

2. PROBLEM FORMULATION

Let the database D be a set of objects. Each object has an *interval attribute* (e.g., time) whose universe U is the real domain representable in a computer. An object's interval attribute contains a sequence of non-overlapping intervals, as demonstrated in Figure 1. Given an interval $[s, e]$, we refer to both s and e as its *endpoints*, and s (e) as its *starting* (*ending*) *value*. Although the intervals from one object are disjoint, the intervals of different objects may overlap. Let N be the total number of intervals from *all* objects in D. Denote by I the set of these N intervals.

The objective is to partition I into smaller sets so that they can be stored and processed in a distributed and parallel fashion. A *size-k partition P over I*, denoted as $P(I, k)$, is defined as follows:

1. P contains m *splitters*, where $0 \leq m \leq k$. Each splitter is a vertical line that is orthogonal to the interval dimension at a *distinct* value $\ell \in U$. We will use ℓ to denote the splitter itself when there is no confusion. Let the splitters in P be $\ell_1, \ldots, \ell_m$ in ascending order, and for convenience, also define $\ell_0 = -\infty$, $\ell_{m+1} = \infty$. These splitters induce $m + 1$ buckets $\{b_1, \ldots, b_{m+1}\}$ over I, where b_i ($1 \leq i \leq m + 1$) represents the interval $[\ell_{i-1}, \ell_i]$.
2. If $s \neq e$, an interval $[s, e]$ of I is assigned to a bucket b_i ($1 \leq i \leq m + 1$), if $[s, e]$ has an intersection of *non-zero length* with the interval of b_i. That is, the intersection cannot be a point (which has a zero length).

   If $s = e$, $[s, e]$ degenerates into a point, in which case we assign $[s, e]$ to the bucket whose interval contains it. In the special case where $s = \ell_i$ for some $i \in [1, m]$ (i.e., the point lies at a splitter), we follow the convention that $[s, e]$ is assigned to b_{i+1}.
3. We will regard b_i as a set, consisting of the intervals assigned to it. The size of bucket b_i, denoted as $|b_i|$, gives the number of such intervals.

Define the *cost of a partition* P with buckets $b_1, \ldots, b_{m+1}$ as the size of its maximum bucket:

$$c(P) = \max\{|b_1|, \ldots, |b_{m+1}|\}. \quad (1)$$

Since the goal is to partition I for storage and processing in distributed and parallel frameworks, a paramount concern is to achieve load-balancing, towards which a common objective is to minimize the maximum load on any node, so that there is no single bottleneck in the system. The same principle has been used for finding optimal splitters in partitioning points [21] and array datasets [14, 18]. That said, an optimal partition for I is formally defined as follows.

Definition 1 An optimal partition of size k for I is a partition $P^*(I, k)$ with the smallest cost, *i.e.*,

$$P^*(I, k) = \operatorname*{argmin}_{P \in \mathcal{P}(I,k)} c(P) \quad (2)$$

where $\mathcal{P}(I, k)$ is the set of all the size-k partitions over I.

$P^*(I, k)$ is thus referred to as an *optimal partition*. If multiple partitions have the same optimal cost, $P^*(I, k)$ may represent any one of them. In what follows, when the context is clear, we use P^* and P to represent $P^*(I, k)$ and $P(I, k)$, respectively.

Note that it is an equivalent definition if one defines $P(I, k)$ in step 2 such that, bucket b_i gets assigned only the *intersection* of $[s, e]$ with b_i – namely, only a *portion* of $[s, e]$ is assigned to b_i, instead of the entire $[s, e]$. This, however, does not change the number of intervals stored at b_i, which therefore gives rise to the same partitioning problem. Keeping $[s, e]$ entirely in b_i permits conceptually cleaner and simpler discussion (because it removes the need of remembering to take intersection). Hence, we will stick to this problem definition in presenting our solutions.

Consider the example from Figure 2, where I contains 9 intervals from 3 objects. When $k = 2$, the optimal splitters are $\{\ell_1, \ell_2\}$,

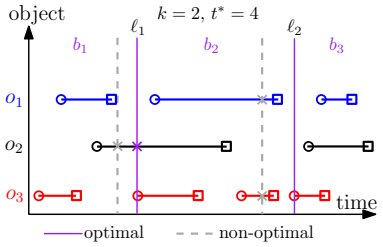


Figure 2: An example.

which yields 3 buckets b_1, b_2, b_3 with 3, 4, and 3 intervals respectively; hence, $c(P^*) = 4$. An alternative partition of 2 splitters is also shown in Figure 2 with dashed lines. It induces 3 buckets with 3, 4, and 5 intervals, so its cost 5 is worse than the aforementioned optimal partition. Note that we use t^* to represent $c(P^*)$.

Depending on whether k is given apriori or not, there are two versions of the optimal splitters problem. In the first case, k is fixed and given at the same time with the input set I. This fits the scenario where the number of nodes/cores in a cluster available for processing I is already known. In the second case, a user is not certain yet how many nodes/cores should be deployed. Thus, he/she wants to explore the cost of the optimal partition for different budget values of k. In this case, k is unknown, and we are allowed to pre-process I in order to quickly find the optimal splitters for any subsequent queries over I with different k values. Formally:

Problem 1 *The static interval splitters problem is to find P^* and $c(P^*)$ for an interval set I and a fixed budget value k.*

Problem 2 *The queryable interval splitters problem is to find P^* and $c(P^*)$ over an interval set I, for any value k that is supplied at the query time as a query parameter.*

Clearly, any solution to the queryable version can be applied to settle the static version, and vice versa (by treating I and k as a new problem instance each time a different k is supplied). However, such brute-force adaptation is unlikely to be efficient in either case. To understand why, first note that, in Problem 2, the key is to pre-process I into a suitable structure so that all subsequent queries can be answered fast (the pre-processing cost is *not* a part of a query's overhead). In Problem 1, however, such preprocessing cost must be counted in an algorithm's overall running time, and can be unworthy because we only need to concentrate on a single k, thus potentially avoiding much of the computation in the preprocessing aforementioned (which must target all values of k).

We investigate the static problem in Sections 3, 4, and 5, and the queryable problem in Section 6. In the queryable problem, handling updates in I becomes an important issue, and presents additional challenges, which are also tackled in Section 6.

3. A BASELINE METHOD

Let us denote the N intervals in I as $[s_1, e_1], \ldots, [s_N, e_N]$. Suppose, without loss of generality, that $s_1 \leq \ldots \leq s_N$. Let $S(I) = \{s_1, \ldots, s_N\}$, *i.e.*, the set of starting values of the intervals in I. When I is clear from the context, we simply write $S(I)$ as S.

Where to place splitters? For any splitter ℓ, denote by $\ell(1)$ the splitter that is placed at the smallest starting value in S that is larger than or equal to ℓ. If no such starting value exists, $\ell(1)$ is undefined. Note that if ℓ itself is a starting value, $\ell(1) = \ell$.

It turns out that, to minimize the cost of a partition over I, it suffices to place splitters only at the values in S. This is formally stated in the next two lemmas:

Lemma 1 *Consider any partition P with distinct splitters $\ell_1, \ldots, \ell_m$ in ascending order, such that $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \ldots, \ell_{m-1}$. Then, it always holds that $c(P') = c(P)$.*

PROOF. Let b_m and b_{m+1} be the last two buckets in P (which are separated by ℓ_m), and b'_m be the last bucket of P' (which is to the right of ℓ_m). When $\ell_m(1)$ is undefined, there are no new intervals starting to the right of ℓ_m. Hence, it is not hard to show that $b_{m+1} \subseteq b_m = b'_m$. This proves the lemma because all the other buckets in P exist directly in P'. □

If s_{max} is the largest starting value in S, the above lemma suggests that we can drop all those splitters greater than s_{max} without affecting the cost of the partition. Hence, it does not pay off to have such splitters.

Lemma 2 *Consider any partition P with distinct splitters $\ell_1, \ldots, \ell_m$ in ascending order, such that $\ell_m(1)$ exists. Let ℓ_i be the largest splitter that does not belong to S (i.e., $\ell_j \in S$ for all $j < i \leq m$). Define P' as a partition with splitters*

- *$\ell_1, \ldots, \ell_{i-1}, \ell_i(1), \ell_{i+1}, \ldots \ell_m$, if $\ell_i(1) \neq \ell_{i+1}$;*
- *$\ell_1, \ldots, \ell_{i-1}, \ell_{i+1}, \ldots \ell_m$, otherwise.*

Then, it always holds that $c(P') \leq c(P)$.

PROOF. We consider only the first bullet because the case of the second bullet is analogous. Let b_i (b_{i+1}) be the bucket in P that is on the left (right) of ℓ_i. Similarly, let b'_i (b'_{i+1}) be the bucket in P' that is on the left (right) of $\ell_i(1)$. We will show that $b'_i \subseteq b_i$ and $b'_{i+1} \subseteq b_{i+1}$, whose correctness implies $c(P') \leq c(P)$, because all the other buckets in P still exist in P', and vice versa.

We prove only $b'_i \subseteq b_i$ because a similar argument validates $b'_{i+1} \subseteq b_{i+1}$. Given an interval $[s, e]$ assigned to b'_i, we will show that it must have been assigned to b_i, too. Note that, by definition of $\ell_i(1)$, *no starting value can exist in* $[\ell_i, \ell_i(1))$. This means that $s < \ell_i$ because s would not be assigned to b'_i if $s = \ell_i(1)$.

If $s = e$, then s must fall in $[\ell_{i-1}, \ell_i)$, which means that s is also assigned to b_i. On the other hand, if $s \neq e$, a part of $[s, e]$ needs to intersect (ℓ_{i-1}, ℓ_i) so that $[s, e]$ can have intersection of non-zero length with b'_i. This means that $[s, e]$ also has non-zero-length intersection with b_i, and therefore, is assigned to b_i. □

This shows that if a partition has a splitter that is not in S, we can always "snap" the splitter to a value in S, without increasing the cost of the partition (may decrease the cost of the partition).

Dynamic programming. Given a splitter ℓ, we define $I^-(\ell)$, $I^+(\ell)$, and $I^o(\ell)$ as the subset of intervals in I whose starting values are less than, greater than, and equal to ℓ, respectively:

$$\begin{aligned} I^-(\ell) &= \{[s_i, e_i] \in I \mid s_i < \ell\} \\ I^+(\ell) &= \{[s_i, e_i] \in I \mid s_i > \ell\} \\ I^o(\ell) &= \{[s_i, e_i] \in I \mid s_i = \ell\}. \end{aligned}$$

An interval $[s, e]$ is said to *strongly cover* a splitter ℓ if $s < \ell < e$ (note that both inequalities are strict). Let $I^\times(\ell)$ be the subset of intervals from I that strongly cover ℓ:

$$I^\times(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}.$$

The following fact paves the way to a dynamic programming algorithm solving Problem 1:

Lemma 3 *If $k = 0$, $c(P^*(I, k)) = |I|$. For $k \geq 1$, $c(P^*(I, k)) =$*

$$\min\left\{|I|,\ \min_{\ell \in S}\{\max\{c(P^*(I^-(\ell), k-1)), \lambda\}\}\right\}. \quad (3)$$

where $\lambda = |I^o(\ell)| + |I^+(\ell)| + |I^\times(\ell)|$.

PROOF. The case of $k = 0$ is obvious, so we concentrate on $k \geq 1$. A partition of size-k over I is allowed to use m splitters, where m ranges from 0 to k. If $m = 0$, then apparently the partition has cost $|I|$. The rest of the proof considers $m \geq 1$.

Lemmas 1 and 2 indicate that it suffices to consider partitions where all splitters fall in S. Let us fix a starting value $x \in S$. Consider an arbitrary partition $P(I,k)$ *whose last splitter* ℓ *is at position* x. Let $m \in [1,k]$ be the number of splitters in $P(I,k)$. Let $P(I^-(x),k-1)$ represent the partition over $I^-(x)$ with the first $m-1$ splitters of $P(I,k)$. Denote by $b'_1,...,b'_m$ the buckets of $P(I^-(x),k-1)$ in ascending order.

Let $b_1,...,b_{m+1}$ be the buckets of $P(I,k)$ in ascending order. As $P(I,k)$ shares the first $m-1$ splitters with $P(I^-(x),k-1)$, we know $b_i = b'_i$ for $1 \leq i \leq m-1$. Next, we will prove:

- Fact 1: $b_m = b'_m$
- Fact 2: $|b_m + 1| = |I^o(x)| + |I^+(x)| + |I^\times(x)|$.

These facts indicate:

$$\begin{aligned} c(P(I,k)) &= \max\{|b_1|,...,|b_{m+1}|\} \\ &= \max\{|b'_1|,...,|b'_m|,|b_{m+1}|\} \\ &= \max\{c(P(I^-(x),k-1)),|b_{m+1}|\} \\ &\geq \max\{c(P^*(I^-(x),k-1)),|b_{m+1}|\} \end{aligned}$$

where the equality can be achieved by using an optimal partition $P^*(I^-(x),k-1)$ to replace $P(I^-(x),k-1)$. Then, the lemma follows by minimizing $c(P(I,k))$ over all possible x.

Proof of Fact 1. We will prove only $b_m \subseteq b'_m$ because a similar argument proves $b'_m \subseteq b_m$. Let $[x',x]$ be the interval of b_m. Accordingly, the interval of b'_m is $[x',\infty)$. Consider an interval $[s,e]$ of I that is assigned to b_m. If $s = e$, then it must hold that $x' \leq s < x$ (note that $s \neq x$; otherwise, $[s,e]$ is assigned to b_{m+1}), which means that $[s,e] \in I^-(x)$, and hence, $[s,e]$ is assigned to b'_m. On the other hand, if $s \neq e$, then $[s,e]$ has a non-zero-length intersection with $[x',x]$, implying that $[s,e]$ intersects (x',x). Hence, $s < x$, and $[s,e]$ has a non-zero-length intersection with $[x',\infty)$. This proves that $[s,e]$ is also assigned to b'_m.

Proof of Fact 2. By definition, an interval $[s,e]$ in I is assigned to b_{m+1} in three disjoint scenarios: 1) $s > x$, 2) strongly covers x, and 3) $s = x$. The numbers of intervals in these scenarios are given precisely by $I^+(x)$, $I^\times(x)$, and $I^o(x)$, respectively. □

We are now ready to clarify our dynamic programming algorithm, which aims to fill in an N by k matrix, as shown in Figure 3. Cell $[i,j]$ (at the i-th row, j-th column) records the optimal cost for a sub-problem $c(P^*(I(i),j))$, where $I(i)$ denotes the set of intervals in I with ids 1, ..., i (we index intervals in I in ascending order of their starting values, break ties first by ascending order of their ending values, and then arbitrarily). Thus, Cell $[i,j]$ represents the optimal cost of partitioning $I(i)$ using up to j splitters.

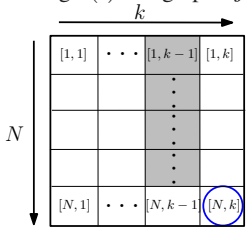


Figure 3: The DP method.

Specifically, we fill the matrix starting from the top-left corner to the bottom-right corner. To fill the cost in Cell $[i,j]$, according to Lemma 3, the last splitter in a partition $P(I(i),j)$ may be placed at any value in $\{s_1,...,s_i\}$. Hence, one needs to check $i-1$ cells from the $(j-1)$-th column, i.e., from $[1,j-1]$ to $[i-1,j-1]$. For instance, in Figure 3, to find the value for the cell $[N,k]$, we need to check $N-1$ cells from the previous column (the gray cells). We refer to this approach as the *DP* method.

The above algorithm can be slightly extended in the straightforward manner to remember also the partitions found, so that the DP method outputs both P^* and $c(P^*)$.

Cost analysis. Sorting I by starting values takes $O(N \log N)$ time. Next, we focus on the cost of DP. Given a splitter ℓ_j, using ideas to be explained in Section 4, we can determine λ in $O(1)$ time with the help of a structure that can be built in $O(N \log N)$ time. Hence, to fill in Cell $[i,j]$, it requires to check $i-1$ cells in the preceding column plus $O(1)$ cost for obtaining λ. Completing an entire column thus incurs $O(\sum_{i=1}^{N} i) = O(N^2)$ time. As k columns need to be filled, the overall running time is $O(kN^2)$.

4. INTERNAL MEMORY METHODS

The DP method clearly does not scale well with the database size N due to its quadratic complexity. In this section, we develop a more efficient algorithm for Problem 1, assuming that the database can fit in memory.

The decision version of our problem is what we call the *cost-t splitters problem*: determine whether there is a size-k partition P with $c(P) \leq t$, where t is a positive integer given as an input parameter. If such P exists, t is *feasible*, or otherwise, *infeasible*. If t is feasible, we define $\bar{t}$ as an arbitrary value in $[1,t]$ such that there is a size-k partition P with $c(P) = \bar{t}$, i.e,

$$\bar{t} = \text{an arbitrary } x \in [1,t] \text{ s.t. } \exists P \in \mathcal{P}(k,I), c(P) = x. \quad (4)$$

When t is infeasible, define $\bar{t} = 0$. An algorithm solving the cost-t splitters problem is required to output $\bar{t}$, and if $\bar{t} > 0$ (i.e., t is feasible), also a P with $c(P) = \bar{t}$. The following observation follows immediately the above definitions:

Lemma 4 *If t is infeasible, then any $t' < t$ is also infeasible.*

A trivial upper bound of t is N. Hence, the above lemma suggests that we can solve Problem 1 by carrying out binary search to determine the smallest feasible t in $[1,N]$. This requires solving $O(\log N)$ instances of the cost-t splitters problem. In the sequel, we will show that each instance can be settled in $O(k)$ time, using a structure constructable in $O(N \log N)$ time. This gives an algorithm for Problem 1 with $O(N \log N + k \log N) = O(N \log N)$ overall running time.

As before, we denote the intervals in I as $[s_1,e_1],...,[s_N,e_N]$ sorted in non-descending order of their starting values; break ties by non-descending order of their ending values first, and then arbitrarily. Interval $[s_i,e_i]$ is said to have *id* i. We consider that I is given in an array where the i-th element is $[s_i,e_i]$ for $1 \leq i \leq N$.

Stabbing-count array. As will be clear shortly, the key to attacking the cost-t splitters problem is to construct a *stabbing-count array* A. For each $i \in [1,N]$, define $\sigma[i]$ as the number of intervals in I strongly covering the value s_i, i.e.,

$$\sigma[i] = |I^\times(s_i)|. \quad (5)$$

Furthermore, define $\delta[i]$ as the number of intervals in I with starting values equal to s_i but with ids less than i. Formally, if $I^o_<(s_i) = \{[s_j,e_j] \in I \mid s_j = s_i \wedge j < i\}$, then

$$\delta[i] = |I^o_<(s_i)|. \quad (6)$$

A stabbing-count array A is simply an array of size N where $A[i] = (\sigma[i],\delta[i])$, $1 \leq i \leq N$. Figure 4 shows an example

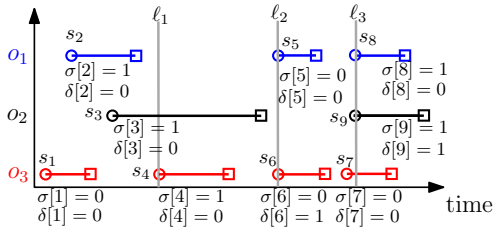


Figure 4: Stabbing-count array.

where I has $N = 9$ intervals, and the values of $\sigma[i]$, $\delta[i]$ have been shown under the interval with id i. For instance, $\sigma[4] = 1$ because there is one interval $[s_3, e_3]$ strongly covering s_4, whereas $\delta[6] = 1$ because one interval $[s_5, e_5]$ has the same starting value as $[s_6, e_6]$, and yet, has a smaller id than $[s_6, e_6]$.

Lemma 5 *The stabbing count array can be built in $O(N \log N)$ time.*

PROOF. This is done with a sweeping technique. First, use $O(N \log N)$ time to sort in ascending order the $2N$ endpoints (starting and ending values) of the N intervals in I, breaking ties as follows:

- A starting value is always put before an ending value;
- If both endpoints are starting values, the one belonging to an interval with a smaller id is put earlier (similarly, if both endpoints are ending values).

Denote by E the sorted list. For each endpoint $x \in E$, we associate it with the id i of the interval where x belongs, and if x is an ending value, also with the starting value s_i of that interval.

Next, we scan E once by its ordering. In this process, an interval $[s, e]$ is *alive* if s has been scanned, but e has not. At any moment, we keep track of 1) s_{last}: the last starting value scanned, 2) the number c of alive intervals, and 3) the number c_δ of alive intervals whose starting values equal s_{last}. As we will see, all this information can be updated in constant time per endpoint in E, and allows us to generate an entry of stabbing count array A in constant time after a starting value is scanned.

Specifically, at the beginning of the scan, $c = c_\delta = 0$ and $s_{last} = -\infty$. Let x be the next endpoint to be scanned, and i the id of the interval to which x belongs. We proceed as follows:

- Case 1: x is an ending value e_i. Decrease c by 1. Furthermore, if $s_i = s_{last}$, also decrease c_δ by 1.
- Case 2: x is a starting value. Increase c by 1. Furthermore, if $x = s_{last}$, increase c_δ by 1; otherwise, set s_{last} to x, and reset c_δ to 1. We also generate an entry $(\sigma[i], \delta[i])$ of A, where $\sigma[i] = c - c_\delta$, and $\delta[i] = c_\delta - 1$.

The scan clearly takes $O(N)$ time, thus completing the proof. □

Cost-t splitters in memory. Given the stab-counting array A, we now describe an algorithm, called t-*jump*, for solving the cost-t splitters problem in $O(k)$ time. If t is feasible, our algorithm outputs $\bar{t}$ and a partition P with splitters $\ell_1, ..., \ell_m$, where m is some integer between 0 and k, such that $c(P) = \bar{t} \leq t$. Each ℓ_j $(1 \leq j \leq m)$ is a starting value of a certain interval in I. We denote by $p(j)$ the id of that interval, namely, $\ell_j = s_{p(j)}$.

Algorithm 1 illustrates the details of t-jump. At a high level, the main idea is to place the splitters in ascending order, and particularly, in such a way that the next splitter is pushed as far away from the preceding one as possible, aiming to let the new bucket have size exactly t (hence, the name t-jump). However, as we will see, due to the overlapping among intervals, the aim is not always achievable. When it is not, we need to settle for a bucket with a smaller size, by moving the next splitter backwards, but just enough to make the new bucket's size drop below t.

Algorithm 1: t-jump (I, k, t)

```
1  if t ≥ N then
2      return t̄ = N, and an empty splitter set;
3  p(1) = t + 1 − δ[t + 1]; |b_1| = t − δ[t + 1];
4  if |b_1| = 0 then
5      return t̄ = 0;
6  t̄ = |b_1|;
7  for j = 2, ..., k do
8      x_j = p(j − 1) + t − σ[p(j − 1)];
9      if x_j > N then
10         |b_j| = N − p(j − 1) + 1 + σ[p(j − 1)];
           t̄ = max(t̄, |b_j|);
11         return t̄, and splitters s_{p(1)}, ..., s_{p(j−1)};
12     p(j) = x_j − δ[x_j];
13     if p(j) = p(j − 1) then
14         return t̄ = 0;
15         /* p(j) < p(j − 1) cannot happen */
16     |b_j| = t − δ[x_j]; t̄ = max(t̄, |b_j|);
17 |b_{k+1}| = N − p(k) + 1 + σ[p(k)];
18 if |b_{k+1}| > t then
19     return t̄ = 0;
20 t̄ = max(t̄, |b_{k+1}|);
21 return t̄, and splitters s_{p(1)}, ..., s_{p(k)}.
```

Now, let us walk through the algorithm. In the outset, Lines 1-2 deal with the trivial case where $t \geq N$ (such t is always feasible, even if no splitter is used). Lines 3-6 place the first splitter ℓ_1 at the *largest starting value that guarantees* $|b_1| \leq t$. The rationale of these lines is from the next lemma:

Lemma 6 *We have:*

- *For $\ell_1 = s_{p(1)}$ where $p(1)$ is set as in Line 3, $|b_1| = t - \delta[t+1] \leq t$. On the other hand, if $\ell_1 > s_{p(1)}$, $|b_1| > t$.*
- *If $|b_1|$ as set in Line 3 equals 0, t is infeasible.*

PROOF. To prove the first sentence in the first bullet, note that b_1 does not include the interval with id $p(1)$. The bucket includes exactly the intervals with ids 1, ..., $p(1) - 1$, namely, $t - \delta[t + 1]$ of them. To prove the second sentence, if $\ell_1 > s_{p(1)}$, b_1 includes at least the first $t + 1$ intervals, and hence, $|b_1| > t$.

Now we show that the second bullet is also true. In fact, since $t = \delta[t+1]$, the intervals with ids 1, ..., $t+1$ all have the same starting values. These intervals will be assigned to an identical bucket in any partition. This bucket must have a size at least $t + 1$. □

Hence, if $|b_1| = 0$, the algorithm terminates at Line 5, indicating that t is infeasible. Otherwise (*i.e.*, $|b_1| > 0$), we know that the cost of the current partition is $\bar{t} = |b_1|$ (Line 6).

For every $j \in [2, k]$, Lines 8-16 determine splitter ℓ_j, assuming $\ell_1, ..., \ell_{j-1}$ are already available. These lines set ℓ_j to *largest starting value that guarantees* $|b_j| \leq t$, based on the next lemma:

Lemma 7 *We have:*

- *For $\ell_j = s_{p(j)}$ where $p(j)$ is set as in Line 12, $|b_j| = t - \delta[x_j] \leq t$. On the other hand, if $\ell_j > s_{p(j)}$, $|b_j| > t$.*
- *If $p(j) = p(j-1)$ at Line 13, t is infeasible.*

PROOF. Regardless of the position of ℓ_j, b_j *must* contain $\sigma[p(j-1)]$ intervals, i.e., those in $I^{\times}(s_{p(j-1)})$ because they strongly cover $s_{p(j-1)}$, and hence, have non-zero-length intersection with b_j. If ℓ_j is placed at $s_{p(j)}$, then besides the intervals of $I^{\times}(s_{p(j-1)})$, b_j also includes those intervals with ids $p(j-1)$, ..., $p(j)-1$. By definition of the stabbing count array, $|I^{\times}(s_{p(j-1)})| = \sigma[p(j-1)]$. Hence, b_j contains in total $|I^{\times}(s_{p(j-1)})|+p(j)-p(j-1) = t-\delta[x_j]$ intervals. This proves the first sentence of the first bullet.

If $\ell_j > s_{p(j)}$, then besides $I^{\times}(s_{p(j-1)})$, b_j also includes at least the intervals with ids $p(j-1)$, ..., x_j (due to line 12 and the definition of the stabbing count array for $\delta[i]$, $s_{p[j]} = s_{x[j]}$). In this case, $|b_j|$ is at least $x_j - p(j-1) + 1 + |I^{\times}(s_{p(j-1)})| = t+1$. This proves the second sentence of the first bullet.

Next, we show that the second bullet is also true. Notice that $p(j) = p(j-1)$ implies $\delta[x_j] = t - \sigma[p(j-1)]$. By the way x_j is calculated, there are exactly $t - \sigma[p(j-1)] - 1$ starting values between $s_{p(j-1)}$ and s_{x_j}. Hence, since $\delta[x_j] > t-\sigma[p(j-1)]-1$, we know that the interval with id $p(j-1)$ is counted by $\delta[x_j]$ (i.e., the interval belongs to $I^o_<(x_j)$). This means that $s_{p(j-1)} = s_{x_j}$.

Now we have found $\sigma[p(j-1)]$ intervals of I strongly covering $s_{p(j-1)}$, in addition to $\delta(x_j)+1$ intervals whose starting values are $s_{p(j-1)}$ (the $+1$ is due to the interval with id x_j). These $\sigma[p(j-1)] + \delta(x_j) + 1 = t+1$ intervals will always be assigned to an identical bucket in any partition. Therefore, no partition can have cost at most t. □

Hence, Lines 13-15 declare t infeasible if $p(j) = p(j-1)$. Otherwise, Line 16 correctly sets $|b_j|$, and updates the current partition cost $\bar{t}$ if necessary.

Now let us focus on Lines 9-11. They handle the scenario where *less than* k splitters are needed to obtain a partition of cost at most t. Specifically, the final partition has only $j-1$ splitters. Line 10 determines the size of the last bucket, and adjusts the partition cost $\bar{t}$ accordingly. The algorithm terminates at Line 11 by returning the results.

At Line 17, we have obtained all the k splitters, and hence, the last bucket b_{k+1} has been automatically determined. The line computes its size, while Lines 18-20 check its feasibility (a negative answer leads to termination at Line 19), and update the partition cost if needed. Finally, Line 21 returns the k splitters already found, as well as the cost $\bar{t}$ of the partition.

We illustrate the algorithm with the example in Figure 4, setting $t = k = 3$. To find the first splitter ℓ_1, we compute $p(1) = 4$, and hence, place ℓ_1 at s_4. To look for the second splitter ℓ_2, Line 8 calculates $x_2 = p(1) + t - \sigma[p(1)] = 4+3-1 = 6$. However, as $\delta[6] = 1 > 0$, we move $p(2)$ back to $x_2 - \delta[x_2] = 6-1 = 5$. For the third splitter ℓ_3, we derive $p(3) = p(2) + t - \sigma[p(2)] = 5+3-0 = 8$. Finally, the algorithm checks the size of the last bucket, which is $N - p(3) + 1 + \sigma[p(3)] = 9-8+1+1 = 3$, namely, still within the target cost t. Hence, t-jump outputs $\bar{t} = 3$, and splitters $\ell_1 = s_4, \ell_2 = s_5$, and $\ell_3 = s_8$.

Lemma 8 *If t-jump does not return $\bar{t} = 0$, then the splitters output constitute a partition with cost $\bar{t} \leq t$. Otherwise (i.e.,$\bar{t} = 0$), then t must be infeasible.*

PROOF. The first sentence is easy to show, given that the sizes of all buckets have been explicitly given in Algorithm 1. Next, we focus on the case where $\bar{t} = 0$. The algorithm may return $\bar{t} = 0$ at three places: Lines 5, 14, and 19. Lemmas 6 and 7 have already shown that termination at Lines 5 and 14 is correct. It thus remains to show that Line 19 termination is also correct.

Assume, for the purpose of contradiction, that j-jump reports $\bar{t} = 0$ at Line 19, but there exists a size-k partition P' over I such that $c(P') \leq t$. Suppose that P' has splitters $\ell'_1, \ldots, \ell'_m$ in ascending order for some $m \leq k$, which define $m+1$ buckets $b'_1, \ldots, b'_{m+1}$ again in ascending order. By Lemmas 1 and 2, we can consider that $\ell'_1, ..., \ell'_m$ are all starting values in S.

We will establish the following statement: *for each $j \in [1, m]$, t-jump always places the jth smallest splitter ℓ_j in such a way that $\ell_j \geq \ell'_j$* – referred to as the *key statement* henceforth. This statement will complete the proof of Lemma 8. To see this, notice that it indicates $\ell_k \geq \ell_m \geq \ell'_m$, which in turn means $|b_{k+1}| \leq |b'_{m+1}|$. However, as Line 19 tells us $|b_{k+1}| > t$, it thus must hold that $|b'_{m+1}| > t$, thus contradicting the fact that $c(P') \leq t$.

We now prove the key statement by induction. As the base step, $\ell_1 = s_{p(1)} \geq \ell'_j$ because if not, then by Lemma 6 $|b'_1|$ must be strictly greater than t, violating $c(P') \leq t$.

Now assuming that the key statement is correct for $j = z$, next we show that it is also correct for $j = z+1$. In fact, if $\ell_{z+1} = s_{p(z+1)} < \ell'_{z+1}$, then by Lemma 7 a bucket with interval $[\ell_z, \ell'_{z+1}]$ will have a size greater than t. However, since $\ell_z \geq \ell'_z$, we know that the interval $[l'_z, l'_{z+1}]$ of bucket b'_z contains $[\ell_z, \ell'_{z+1}]$, and therefore, $|b'_{z+1}|$ must also be greater than t, contradicting $c(P') \leq t$. This completes the proof of the key statement. □

Algorithm t-jump in Algorithm 1 clearly runs in linear time to the number of splitters, i.e., $O(k)$ time. Putting everything in this section together, we have arrived at the first main result of the paper.

Theorem 1 *The static interval splitters problem can be solved in $O(N \log N)$ time in internal memory.*

5. EXTERNAL MEMORY METHODS

This section discusses how to solve the static interval splitters problem I/O-efficiently when the input set I of intervals does not fit in memory. Our analysis will be carried out in the standard *external memory* model of computation [2]. In this model, a computer has M words of memory, and a disk has been formatted into *blocks* (a.k.a. pages) of size B words. An I/O operation either reads a block from the disk to memory, or conversely, writes a block in memory to the disk. The objective of an algorithm is to minimize the number of I/Os. We assume $M \geq 3B$, i.e., the memory is large enough to store at least 3 blocks of data.

Initially, the input set I is stored in a disk-resident array that occupies $O(N/B)$ blocks. When the algorithm finishes, we should have output the $m \leq k$ splitters of the final partition to a file of $O(k/B)$ blocks in the disk. Define:

$$SORT(N) \quad = \quad (N/B)\log_{M/B}(N/B).$$

It is well-known that sorting a file of N elements entails $O(SORT(N))$ I/Os by the textbook external sort algorithm. The rest of the section serves as the proof for the theorem below:

Theorem 2 *The static interval splitters problem can be solved using $O(SORT(N))$ I/Os in external memory.*

As before, denote the intervals in I as $[s_1, e_1]$, $[s_2, e_2]$, ..., $[s_N, e_N]$ in ascending order of s_i (break ties by ascending order of their ending values first, and then arbitrarily), $1 \leq i \leq N$, where $[s_i, e_i]$ is said to have id i. Henceforth, we consider that I is stored as a disk-resident array where the i-th element is $[s_i, e_i]$. This can be fulfilled by simply sorting the original input I, whose cost is within the budget of Theorem 2.

Adapting the main-memory algorithm. The previous section has settled the static interval splitters problem in $O(N \log N)$ time when the input I fits in memory. Recall that our algorithm has two steps: it first creates the stabbing count array A in $O(N \log N)$

time, and then solves $O(\log N)$ instances of the cost-t splitters problem, spending $O(k)$ time on each instance.

In external memory, a straightforward adaptation gives an algorithm that performs $O(SORT(N) + \min(k, \frac{N}{B}) \log N)$ I/Os. Recall from Section 4 that the computation of the stabbing count array A requires only sorting $2N$ values followed by a single scan of the sorted list. Hence, the first step can be easily implemented in $O(SORT(N))$ I/Os in external memory. The second step, on the other hand, trivially runs in $O(\min(k, \frac{N}{B}) \log N)$ I/Os, by simply treating the disk as virtual memory.

This algorithm is adequate when k is not very large. The term $\min(k, \frac{N}{B}) \log N$ is asymptotically dominated by $O(SORT(N))$ when $k = O\big(\frac{N}{B \log_2(M/B)}\big)$ (note that the base of the logarithm is 2). However, the solution falls short for our purpose of claiming a clean bound $O(SORT(N))$ for the entire range of $k \in [1, N]$ (as is needed for proving Theorem 2).

In practice, this straightforward solution can be expensive when k is large, which may happen in a cluster. Note that k could be the total number of cores in a cluster, when each core is responsible for processing one bucket in a partition. So it is not uncommon to have k in a few thousand, or even tens of thousands in a cluster.

Next, we provide an alternative algorithm in the external memory that performs $O(SORT(N))$ I/Os.

Cost-t splitter in external memory. The *cost-t splitters problem* (defined in Section 4) determines whether there is a size-k partition P with cost $c(P) \leq t$; moreover, if P exists, an algorithm also needs to output such a partition (any P with $c(P) \leq t$ is fine). Assuming that the stabbing-count array A has been stored as a file of $O(N/B)$ blocks, next we explain how to solve this problem with $O(N/B)$ I/Os.

The algorithm implements the idea of our main-memory solution by scanning the arrays A and I synchronously once. Following the notations in Section 4, let $\ell_1, \ldots, \ell_m$ be the splitters of P (where $1 \leq m \leq k$), and $p(i)$ an interval id such that $\ell_i = s_{p(i)}$, $1 \leq i \leq m$. We start by setting $p(1)$ as in Line 3 of Algorithm 1, fetching the $p(1)$-th interval of I, and writing it to an output file. Iteratively, having obtained $p(i)$, for $1 \leq i \leq m-1$, we forward the scan of A to $A[p(i)]$, and compute $p(i+1)$ according to Lines 8-16 in Algorithm 1. Then, we forward the scan of I to retrieve the $p(i+1)$-th interval of I, and append it to the output file. Recall that the main-memory algorithm would declare the absence of a feasible P under several situations. In external memory, when any such situation occurs, we also terminate with the absence declaration, and destroy the output file.

The total cost is $O(N/B)$ I/Os because we never read the same block of I or A twice. The algorithm only requires keeping $O(1)$ information in memory. In particular, among $p(1), \ldots, p(i)$ (suppose $p(i+1)$ is not available yet), only $p(i)$ needs to be remembered. We will refer to $p(i)$ as the *front-line value* of the algorithm. By definition, once $p(i+1)$ is obtained, it becomes the new front-line value, thus allowing us to discard $p(i)$ from memory.

Cost-t testing. Let us consider an easier variant of the cost-t splitters problem called *cost-t testing*, which is identical to the former problem except that it does not require an algorithm to output a partition in any case (i.e., even if P exists). An algorithm outputs only a boolean answer: yes (that is, P exists), or no.

Clearly, the cost-t testing problem can also be solved in $O(N/B)$ I/Os. For this purpose, we slightly modify our algorithm for cost-t splitter: (i) eliminate the entire part of the algorithm dealing with the output file (which is unnecessary for cost-t testing), and (ii) if the algorithm declares the absence of a feasible P, we return no for cost-t testing; otherwise, i.e., the algorithm terminates without such a declaration, we return yes.

What do we gain from such a modification, compared to using the cost-t splitter algorithm to perform cost-t testing directly? The answer is the avoidance of writing $O(k/B)$ blocks. Recall that the cost-t splitter algorithm would produce during its execution an output file whose length can reach k. Doing away with the output file turns out to be crucial in attacking a concurrent extension of cost-t testing, as discussed next, which is the key to proving Theorem 2.

Concurrent testing. The goal of this problem is to solve multiple instances of the cost-t testing problem simultaneously. Specifically, given h integers satisfying $1 \leq t_1 < t_2 < \ldots < t_h \leq N$, the *concurrent testing problem* settles h instances of cost-t testing for $t = t_1, ..., t_h$, respectively. Following the result in Lemma 4, cost-t testing obeys the monotonicity that if cost-t testing returns yes (or no), then cost-t' with any $t' > t$ (or $t' < t$, respectively) will also return yes (no). Therefore, the output of concurrent testing can be a single value τ, equal to the smallest t_j ($1 \leq j \leq h$) such that t_j-testing returns yes. Note that τ does not need to always exist: the algorithm returns nothing if t_h-testing returns negatively (in which case, the t_j-testings of all $j \in [1, h-1]$ must also return no).

Assuming $h \leq cM$ where $0 < c < 1$ is to be decided later, we can perform concurrent testing in $O(N/B)$ I/Os. We concurrently execute h cost-t testings, each of which sets t to a distinct t_j, $1 \leq j \leq h$. The concurrency is made possible by several observations on the cost-t testing algorithm we developed earlier:

1. Regardless of t, the algorithm scans I and A only *forwardly*, i.e., it never reads any block that has already been passed.

2. The next block to be read from I (A) is uniquely determined by its front-line value $p(i)$. In particular, if $p(i)$ is larger, then the block lies further down in the array I (A).

3. As one execution of the algorithm requires only $c' = O(1)$ words of memory, by setting
$$c = \frac{1}{c'}\left(1 - \frac{2B}{M}\right) \tag{7}$$
we ensure that h concurrent threads of the algorithm demand at most $cM \cdot c' = M - 2B$ words of memory. This will always leave us with two available memory blocks, which we deploy as the input buffers for reading I and A, respectively. Note that since $M \geq 3B$, we have $c \geq 1/(3c')$, indicating that $cM = \Omega(M)$.

In memory, the algorithm uses a min-heap H to manage the front-line values of the h threads of cost-t testing. At each step, it de-heaps the smallest value p from H. Suppose without loss of generality that p comes from the thread of cost-t_j testing, for some $j \in [1, h]$. We execute this thread until having obtained its new front-line value, which is then en-heaped in H. This continues until all threads have terminated, at which point we determine the output τ as explained before. A trivial improvement is to stop testing t_j's for $t_j > t_i$ if the testing on t_i returns that t_i is feasible. The fact that it performs only $O(N/B)$ I/Os follows directly from the preceding observations about each thread of cost-t testing.

Solving the static interval splitters problem. Our I/O efficient algorithm for the static interval splitters problem has three steps:

1. Construct the stabbing count array A.
2. Obtain the minimum t^* such that cost-t^* testing returns yes.

3. Solve the cost-t^* splitters problem to retrieve the splitters of an optimal partition P^*.

We refer to this algorithm as the *concurrent t-jump* method, or in short *ct-jump*. The correctness of the algorithm is obvious, noticing that Step 2 guarantees t^* to be the cost of an optimal partition.

We explained previously how to do Step 1 in $O(SORT(N))$ I/Os and Step 3 in $O(N/B)$ I/Os. Next, we will show that Step 2 requires only $O((N/B)\log_M N)$ I/Os. This will establish Theorem 2 because, for $N \geq M$, it holds that[1]

$$\frac{\log N}{\log M} \leq \frac{\log N - \log B}{\log M - \log B}$$

Note that the left hand side is $\log_M N$ whereas the right hand side is $\log_{M/B}(N/B)$.

The rest of the section will concentrate on Step 2. It is easy to see that t^* falls in the range $[1, N]$. We will gradually shrink this *permissible range* until eventually it contains only a single value, i.e., t^*. We achieve the purpose by launching multiple rounds of concurrent testing such that, after each round, the permissible range will be shrunk to $O(1/M)$ of the original length. An example for performing concurrent testings to shrink the permissible range is shown in Figure 5.

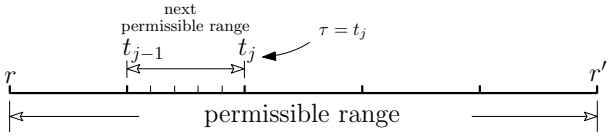


Figure 5: Concurrent testing on permissible ranges.

Specifically, suppose that the permissible range is currently $[r, r']$ such that $r' - r \geq cM$, where c is the constant given in (7). We choose $h = cM$ integers $t_1, ..., t_h$ to divide $[r, r']$ as evenly as possible, namely: $t_j = r + \lceil j(r' - r)/(h + 1) \rceil$, for $j = 1, ..., h$. Then, we carry out concurrent testing with $t_1, ..., t_h$. Let τ be the output of the concurrent testing. Then, the permissible range can be shortened to:

- $[t_h + 1, r']$ if τ does not exist (*i.e.*, the concurrent testing returned nothing).
- $[r, t_1]$ if $\tau = t_1$.
- $[t_{j-1} + 1, t_j]$ if $\tau = t_j$ for some $j > 1$.

It is easy to verify that the length of the new permissible range is at most $2/(cM) = O(1/M)$ that of the old one.

Finally, when the permissible range $[r, r']$ has length at most cM, we acquire the final t^* by one more concurrent testing with $h = r' - r + 1$ values $t_1, ..., t_h$, each of which is set to a distinct integer in $[r, r']$. The value of t^* equals the output τ of this concurrent testing.

It is clear that we perform $O(\log_M N)$ rounds of concurrent testing in total. As each round takes linear I/Os, we thus have obtained an algorithm that implements Step 2 in $O((N/B)\log_M N)$ I/Os. In other words, the algorithm *concurrent t-jump* has IO cost $O(SORT(N))$. This completes the proof of Theorem 2.

6. QUERYABLE INTERVAL SPLITTERS AND UPDATES

In this section, we tackle the challenges in solving the queryable interval splitters problem (*i.e.*, Problem 2).

Queryable interval splitters. Our solutions for the static interval splitters problem also lead to efficient solutions to the queryable interval splitters problem.

In internal memory, we can use the t-jump algorithm for answering any queries with different k values. In a preprocessing step, we build the stabbing count array A (which occupies $O(N)$ space) in $O(N \log N)$ time. For subsequent queries, each query with a different k value takes $O(k \log N)$ time to answer, by solving $\log N$ cost-t splitters problems and each taking $O(k)$ time.

In external memory, the preprocessing cost of the *t-jump method* is $O(SORT(N))$ I/Os to build and maintain the disk-based stabbing count array A. The size of the index (which is just A) is $O(N/B)$, and the query cost is $O(\min\{k, N/B\} \log N)$ I/Os.

The preprocessing step of the *concurrent t-jump method* is the same. Hence, it also takes $O(SORT(N))$ I/Os, and its index also uses $O(N/B)$ space. Each query takes $O((N/B)\log_M N)$ I/Os.

Updates. In the queryable problem, another interesting challenge is to handle dynamic updates in the interval set I. Unfortunately, in this case, update costs in both the *t-jump* and *concurrent t-jump* methods are expensive. The indexing structure in both methods is the stabbing-count array A. To handle arbitrary updates, in the worst case, all elements in A need to be updated. Hence, the update cost is $O(N)$ time in internal memory, and $O(N/B)$ I/Os in external memory. This is too expensive for large data sets.

This limitation motivates us to explore update-efficient indexing structures and query methods for the queryable interval splitters problem. We again leverage the idea of solving $O(\log N)$ instances of the cost-t splitters problems (the decision version of our problem), in order to answer an optimal splitter query with any k value. The challenge boils down to designing an update-friendly indexing structure for answering a cost-t splitters query efficiently.

Observe that the key step in our algorithm for solving a cost-t splitters query in Section 4 is to figure out which starting value to use for placing the next splitter, which is given by Lines 8 and 12 in Algorithm 1. The critical part is to find out:

1) the number of intervals in I that strongly cover a starting value s, which is where a splitter has been placed at, i.e., $|I^{\times}(s)|$.
2) the number of intervals in I that share the same starting values as an interval $[s_i, e_i]$ $(1 \leq i \leq N)$, but with smaller ids less than i, i.e., $|I^o_<(s_i)|$.

Hence, to solve a cost-t splitters problem instance, we just need to use an update-friendly index that answers any stabbing count query efficiently, i.e., an index that finds $|I^{\times}(s)|$ and $|I^o_<(s)|$ for any point s efficiently. This can be done efficiently using a *segment B-tree* [23]. In internal memory, this structure occupies $O(N)$ space, can be built in $O(N \log N)$ time, answers a stabbing count query in $O(\log N)$ time, and supports an insertion/deletion in $O(\log N)$ time. In external memory, the space, construction, query, and update costs are $O(N/B)$, $O((N/B)\log_{M/B}(N/B))$, $O(\log_B N)$ and $O(\log_B N)$, respectively.

Finally, answering a cost-t splitters query requires answering k different stabbing count queries; and answering a queryable interval splitters problem then takes $O(\log N)$ cost-t splitters problem instances. Hence, the overall query cost of this approach is $O(k \log^2 N)$ time in internal memory, and $O(k \log_B N \cdot \log N)$ I/Os in external memory. We denote this method as the *stabbing-count-tree* method, or just *sc-tree*.

7. EXPERIMENTS

We implemented all methods in C++. The external memory methods were implemented using the TPIE-library [4]. All exper-

[1]Let x, y, z be positive values such that $x \geq y > z$, then $\frac{x}{y} \leq \frac{x-z}{y-z}$.

iments were performed on a Linux machine with an Intel Core i7-2600 3.4GHz CPU, a 4GB memory and a 1TB hard drive.

Datasets. We used several large real datasets. The first one *Temp* is from the MesoWest project. It contains temperature measurements from Jan 1997 to Oct 2011 from 26,383 distinct stations across the United States. There are almost 2.6 billion total readings from all stations with an average of 98,425 readings per station. For our experiments, we view each year of readings from a distinct station as *a distinct object*. Each new reading in an object is viewed as an update and creates *a new version of that object*. This leads to a multi-version database and every version of an object defines an interval in the database. *Temp* has 145,628 objects with an average of 17,833 versions per object. So *Temp* has approximately 2.6 billion intervals in total.

The second real dataset, *Meme*, was obtained from the Memetracker project. It tracks popular quotes and phrases which appear from various sources on the Internet. Each record has the URL of the website containing the memes, the time Memetracker observed the memes, and a list of the observed memes. We view each website as an object. Each record reports a new list of memes from a website, and it is treated as an update to the object representing that website, which creates a new version that is alive until the next update (a record containing the same website). *Meme* has almost 1.5 million distinct websites and an average of 67 records per website. Each version of an object in *Meme* also defines an interval, and we have approximately 100 million intervals in total.

We have also obtained the popular time series datasets [12] from the UCR Time Series website. In particular, we used three of the largest datasets from this collection; but they are all much smaller than *Temp* and *Meme* as described above. All real datasets are summarized in Table 1, with their number of intervals.

Temp	Meme	CMU	Mallat	NHL
2.6×10^9	1.0×10^8	1.57×10^5	2.39×10^6	1.41×10^6

Table 1: Number of intervals in real datasets tested.

Setup. We primarily used *Meme* in internal memory and *Temp* in external memory. In order to test the scalability, we randomly selected subsets from *Meme* and *Temp* to produce data with different number of intervals. Unless otherwise specified, the default values for important parameters in our experiments are summarized in Table 2. The page size is set to 4096 bytes by default. The default fill factor in the *sc-tree* is 0.7. In each experiment, we varied the value of one parameter of interest, while setting other parameters in their default values. Since the UCR time series datasets are all relatively small in size compared to *Meme* and *Temp*, we only used them for evaluating the internal memory methods.

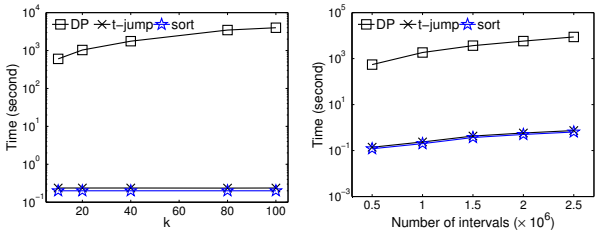
	Internal	External
Dataset	a subset of *Meme*	a subset of *Temp*
Size	$\sim$ 21 MB	$\sim$ 4.1 GB
N	$\sim$ 1 million	$\sim$ 200 million
k	40	5000
h	not applicable	5

Table 2: Default datasets and default values of key parameters.

7.1 Results from internal memory methods

In this case, we focus on the results from the *static interval splitters* problem.

Figure 6 studies the effect of k and N. Clearly, the *DP* method is linear to k as shown in Figure 6(a) and quadratic to N as shown in Figure 6(b). DP is 4-5 orders of magnitude more expensive than our t-jump method. On the other hand, even though the second step in t-jump is to solve $O(\log N)$ instances of cost-t splitters problem and each instance takes $O(k)$ time, the dominant cost for



(a) Vary k. (b) Vary N.

Figure 6: Running time of internal memory methods.

t-jump is the sorting operation in its step, which is to construct the stabbing count array. Hence, its overall cost is $O(N \log N)$. As a result, its running time is almost not affected by k as seen in 6(a), but (roughly) linearly affected by N as seen in 6(b). In conclusion, t-jump is extremely efficient for finding optimal splitters, and it is highly scalable (as cheap as main memory sorting methods). It takes only about 1 second to find optimal splitters for size-40 partitions over 2.5 million intervals, when they are not sorted. Note that t-jump will be even more efficient and scalable if data is already sorted; in that case, its cost reduces to only $O(k \log N)$.

In Figure 7, we report the experimental results on the datasets from the UCR time series collection. The trend is similar to our observations from the *Meme* dataset. Our best solution t-jump consistently performs 3-4 orders of magnitudes faster than the DP approach in all three datasets. The dominant cost for t-jump is from sorting the input data, which is clearly shown in Figure 7.

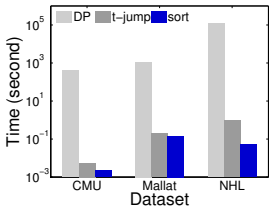
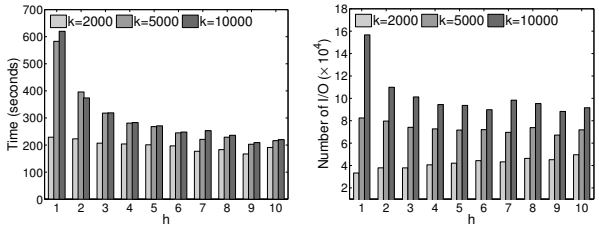


Figure 7: Results from the UCR datasets.

7.2 Results from external memory methods

We first present the results for the *static interval splitters* problem, then analyze the results for the *queryable interval splitters* problem. In both problems, we need to study the effect of h from the second step in our concurrent t-jump method. Hence, we first evaluate the impact of h as shown in Figure 8.



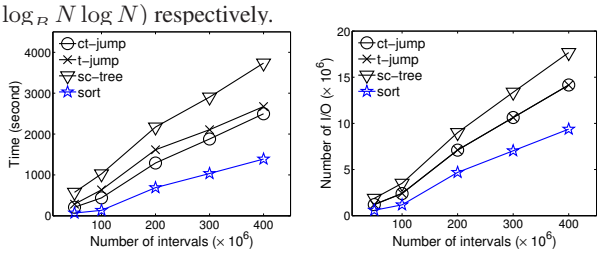
(a) Time. (b) IO.

Figure 8: Effect of h in the second step of ct-jump.

To isolate the impact of h, we show only running time from the second step of the ct-jump method, i.e., we assume that the stabbing count array has already been constructed and do not include its construction cost in Figure 8. We vary h between 1 to 10, and repeat the same experiment for $k = 2,000$, $k = 5,000$ and $k = 10,000$. What is interesting to observe from Figure 8(a) is that the running time initially deceases sharply and then slightly increases (very slowly), when we increase h. The same trend (albeit being less obvious and consistent) can also be observed for the number of IOs in Figure 8(b). This is because that the initial increment in h helps quickly reduce the permissible range, but subsequent increases in h lead to little gain (in shrinking the permissible range), but require more unnecessary testings. These results show that a

small h value is sufficient for ct-jump to produce consistently good performance for a wide-range of k values. Hence, we set $h = 5$ as the default value for the rest of the experiments.

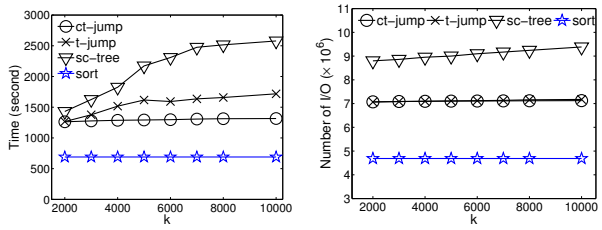
Results for static interval splitters. Recall that in this case, the overall cost includes the cost for building either the stabbing count array or the stabbing count tree, and the cost for finding the optimal splitters with the help of such a data structure. Also recall that the cost in this case, in terms of IOs, for ct-jump, t-jump, and sc-tree is $O(SORT(N))$, $O(SORT(N)+k\log N)$, and $O(SORT(N)+k\log_B N\log N)$ respectively.



(a) Running time. (b) Total IO.

Figure 9: Static splitters, vary N.

Figure 9 studies the scalability of different methods by varying N from 50 million to 400 million. Not surprisingly, all methods have an almost linear dependence to N in terms of both running time and IOs. But obviously ct-jump achieves the best overall running time in Figure 9(a), and both ct-jump and t-jump have clearly outperformed the sc-tree method. The ct-jump method also has better IOs than t-jump, but the difference is not clearly visible in Figure 9(b), because that the dominant IO cost for both methods is the external sort. Both methods have fewer IOs than sc-tree, especially for larger data. Overall, ct-jump is the best method that is almost as efficient and scalable as external sorting. With $N = 400$ million intervals, ct-jump achieves a 30% speedup over the *sc-tree* method and a 10% speedup over the t-jump method.



(a) Running time. (b) Total IO.

Figure 10: Static splitters, vary k.

We also investigated the impact of k by varying k from $2,000$ to $10,000$. As k becomes larger, both the running time and the number of IOs increase in all methods, especially for the sc-tree method as shown in Figure 10. A larger k value increases the cost of solving a cost-t splitters problem, which is the essential step for all methods. However, the benefit of concurrent testing in ct-jump is clearly reflected in Figure 10(a) for larger k values. Its cost is still very much just the sorting cost, which is consistent with our theoretical analysis. In contrast, the t-jump method displays a clear increase in running time over larger k values. The ct-jump method is more than 2 times faster than sc-tree, and about 20-30% faster than the t-jump method, as we increase k.

In conclusion, ct-jump is the best method for the static interval splitters problem in external memory.

Results for queryable interval splitters. We first study the preprocessing cost to construct the indices for different external memory methods, which is to construct either the stabbing count array in ct-jump and t-jump, or the stabbing count tree in sc-tree. Figure 11 compares the size of the indices in different methods when we increase N from 50 million to 400 million. Both stabbing count array and stabbing count tree are linear to N, however, a stabbing count tree does require much more space, almost by a factor 2 when N becomes 400 million. The stabbing count array in both ct-jump and t-jump is the same in size as the size of all intervals. This means that it will be much smaller than the size of the database, which contains other values than just an interval for each record.

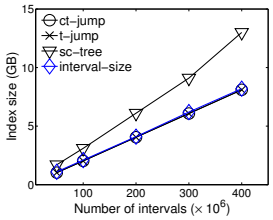
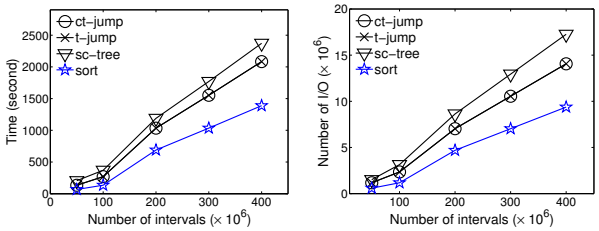


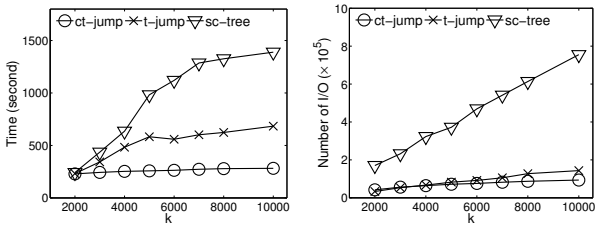
Figure 11: Index size.



(a) Preprocessing time. (b) Preprocessing IO.

Figure 12: Preprocessing cost.

In terms of the construction cost of indices in these methods, Figure 12 shows that building a stabbing count array is slightly cheaper than building a stabbing count tree. But both are dominated by the external sorting cost, and obviously the ct-jump and the t-jump methods share the same preprocessing cost. All three methods have an almost linear construction cost to N.



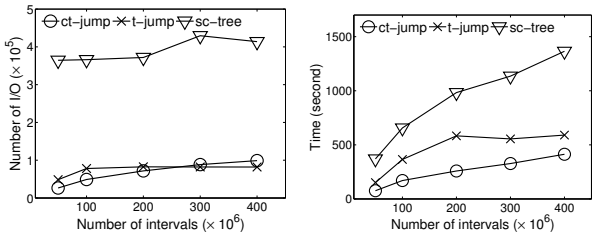
(a) Query time. (b) Query IO.

Figure 13: Queryable splitters, vary k.

We next study the query cost. Figure 13 investigates the impact of k when it changes from $2,000$ to $10,000$. The query time for all three methods increase, as shown in Figure 13(a), but the cost of ct-jump increases at the slowest rate. Specifically, we observe that ct-jump answers a query 3-4 times faster than sc-tree, and about 1-2 times faster than t-jump. A similar trend can be observed in the IO cost, as shown in Figure 13(b). However, in this case, the IO difference between ct-jump and t-jump appears to be much smaller, compared to their difference in query time.

So one may wonder why the query time of ct-jump is much better than t-jump, when the difference in terms of the number of IOs is not so significant. This is explained by the (much) better caching behavior in ct-jump. Recall that ct-jump performs concurrent testing of several cost-t testing instances. As a result, it makes very small leaps while scanning through the stabbing count array, compared to the big leaps made by t-jump over the stabbing count array. These small leaps result in much better hit rates in buffer and cache, leading to much better running time.

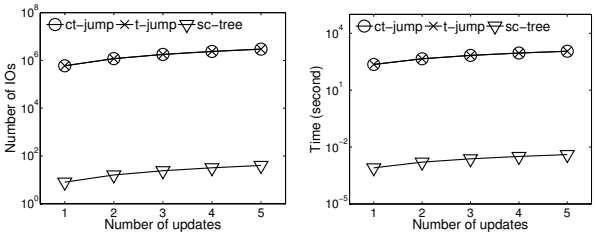
Next, we examine the scalability of our query methods when we increase the size of the data from 50 million to 400 million. Figure 14(a) reports the query cost in terms of number of IOs. Not surprisingly, larger N values only increase the query IOs by a small amount. In contrast to the linear dependence on k, the query cost of ct-jump and t-jump only depends on N by a $O(\log N)$ factor, and



(a) Query IO. (b) Query time.

Figure 14: Queryable splitters, vary N.

the query cost of sc-tree only depends on N by a $O(\log_B N \log N)$ factor. On the other hand, the increases in running time for all three methods are more significant when N increases, as shown in Figure 14(b). This is because that as N becomes larger, for a fixed k value, the distance between two consecutive splitters also becomes larger. Since all three methods essentially make k jumps over the data to solve a cost-t splitters problem instance, a larger jump in distance leads to poorer caching performance, which explains the more notable increase in query time than that in query IO when N increases. Nevertheless, ct-jump outperforms sc-tree by about 60% and t-jump by about 30% in time on average.



(a) Update IO. (b) Update time.

Figure 15: The update cost for queryable splitters.

Lastly, updates to the data may happen for the queryable splitters problem. When dealing with updates is important, as we have analyzed earlier in Section 6, sc-tree becomes a much more attractive method than ct-jump and t-jump. The fundamental reason is that a stabbing count tree is a dynamic indexing structure with an update cost of $O(\log_B N)$ IOs, while a stabbing count array is not with an update cost of $O(N/B)$ IOs. Of course, the update-friendly property in a stabbing count tree comes with the price of having more expensive query cost, as we have already shown.

That said, we generate a number of updates by randomly issuing a few insertions/deletions on the initial set of intervals. We report the average cost in IOs and time from 10 updates, as shown in Figure 15(a) and Figure 15(b). Clearly, the *sc-tree* method is much more efficient in handling updates than *t-jump* and *ct-jump*.

In summary, for the queryable splitter problem in external memory, when data is static (which is often the case for big data), the *ct-jump* method is the most efficient method. When dealing with dynamic updates is important, *sc-tree* works the best. Nevertheless, all three methods have excellent efficiency and scalability on big data, and the key idea behind all three methods is our observation on solving $O(\log N)$ cost-t problem (or testing) instances.

7.3 Optimal Point Splitters

A special case of our problem is when *all intervals degenerate to just points*, where *each interval* starts and ends at one same time instance.

Our solution can gracefully handle such special case, i.e., to find the optimal splitters for a point set. The state-of-the-art method for finding the optimal splitters in a point set runs in $O(k \log^2 N)$ time [21], assuming the set of input points has already been sorted. We dub this approach the p-split method. The details of this study will be surveyed in Section 8. In contrast, t-jump runs in only $O(k \log N)$ time, saving a $O(\log N)$ factor compared to the state-of-the-art on a point dataset. Note that the investigation from [21] focuses only on the RAM model, i.e., p-split is an internal memory method. Therefore, in this experiment, we compare p-split against our internal memory method t-jump by using all the internal memory datasets. In particular, we represent each interval only by its starting value and report the running time for finding the optimal point splitters. The results are shown in Figure 16. Not surprisingly, t-jump performs 3-4 times faster than the state-of-the-art method p-split across all datasets.

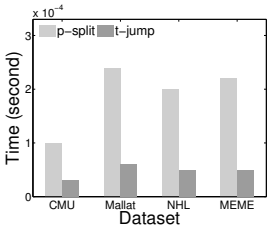
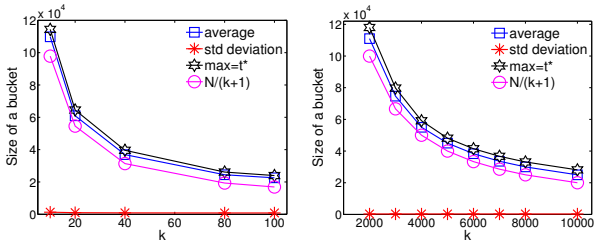


Figure 16: Comparison with p-split [21] to find optimal point splitters.

7.4 Final Remark

It is also interesting to point out our algorithms, t-jump, ct-jump, and sc-tree also produce very balanced buckets while minimizing the size of the maximum bucket. This is due to the fact that they all use our algorithm for solving the cost-t splitters (or cost-t testing) problem as a basic building block. And the way we solve the cost-t splitters (or cost-t testing) problem is to attempt to produce each bucket with a size t. When this is not possible, our algorithm finds a bucket with a size that is as close as possible to t, before producing the next bucket. In other words, in the optimal partition P^* that our algorithms find with the maximum bucket size being equal to t^*, the size of each bucket in P^* is in fact very close to t^*.



(a) Meme Data. (b) Temp Data.

Figure 17: Balanced partitions produced by our algorithms.

Figure 17 verifies this claim over both real data sets, when we vary k and show the average and the standard deviation for the sizes of all buckets in the optimal partition P^* produced by our algorithms, along with $t^* = c(P^*)$, the size of the maximum bucket in P^*. In all cases, the average bucket size is very close to t^*, and the standard deviation is very small (hundred to a thousand, compared to tens of or hundred thousand for the average bucket size). Furthermore, note that the average size of a bucket is not necessarily $\frac{N}{k+1}$ in our problem, since an interval may span over multiple buckets. In fact, it is always $\geq \frac{N}{k+1}$ and varies for different partitions depending on the data set I, the budget k, and the partitioning algorithm used. Nevertheless, all three of our methods, t-jump, ct-jump, and sc-tree, produce the same P^* for a given k and I.

8. RELATED WORK

To the best of our knowledge, this is the first work that investigates the problem of finding optimal splitters for large interval data.

Ross and Cieslewicz studied the problem of finding optimal splitters for a set of one dimensional points [21]. In their problem, a partition consists a set of disjoint buckets over the point dataset. The buckets are produced by k splitters. However, a set of k splitters will produce $(2k + 1)$ buckets instead of just $(k + 1)$ buckets:

they count all points that have values equal to a splitter as a separate bucket, so k splitters form k distinct buckets by themselves (which we refer to as the *splitter buckets*), plus the other $(k+1)$ buckets formed by any two neighboring splitters (which we refer to as the *non-splitter buckets*). Their goal, however, is to minimize the size of the maximum bucket from only the $(k+1)$ non-splitter buckets. Their motivation was to disregard buckets that contain only a single (but many duplicates) value. This finds applications in incremental sorting, distributed and parallel processing (of some applications), etc., as they argued in [21]. They have proposed a $O(k \log^2 N)$ method for memory-resident point sets. Clearly, their problem is fundamentally different from the interval splitters problem. Interestingly, it is possible to extend our stabbing-count array based methods, namely, *t-jump* and *concurrent t-jump*, to solve this problem in internal and external memory respectively. In that case, our solution leads to a $O(k \log N)$ method, which improves the results in [21] by a $O(\log N)$ factor for the point splitters problem.

On the other hand, the array partitioning problem is as follows. The input is an one-dimension array E of N non-negative numbers, and a *weight function w* that maps a contiguous segment of A to a non-negative integer. The k-partition of E is a division of E into $(k+1)$ contiguous segments, that is, setting dividers $\ell_0 = 0 < \ell_1 < \cdots < \ell_k < \ell_{k+1} = N$. Here, the ith segment is $E[\ell_{i-1}+1 \cdots \ell_i]$. The MAX norm of a partition over E is $\max_{i=1}^{k+1} w(E[\ell_{i-1}+1 \cdots \ell_i]$. The goal is to find a partition of size k that minimizes the MAX norm of any size-k partitions over E. Typical weight function includes addition and Hamming weight function [14, 18] among others. This problem can also be extended to 2-dimensional arrays, where a partition consists of a number of disjoint but complete-covering 2-dimensional blocks over the 2-dimensional array E. Khanna et al. studied this problem and gave an $O(N \log N)$ algorithm for memory resident arrays in 1d for arbitrary k. This problem is NP-hard in 2d and they gave approximation methods instead. More efficient and effective approximations were then given in [18]. This problem is related but certainly very different from our work. An interesting open problem is the interval array partitioning problem, which is defined similarly as the array partitioning problem, except that each element in the array is an interval of values (e.g., those from an uncertain object).

Our study may find interesting applications in parallel interval scheduling problems [9], where each job has a specified interval within which it needs to be executed. Each machine has a *busy interval* which contains all the intervals corresponding to the jobs it processes. Given the parallelism parameter $g \geq 1$, which is the maximal number of jobs that can be processed simultaneously by a single machine. The goal is to assign the jobs to machines such that the total busy time of the machines is minimized. This problem is known to be NP-hard for $g \geq 2$. Nevertheless, it is an intriguing future work to explore if our techniques can help design efficient approximate solutions for such problems.

Lastly, the DP method follows the general intuition of bucketization that finds application in optimal histogram constructions, e.g., the DP method for constructing a V-optimal histogram [11,20]. The construction of the stabbing count array is somewhat related to the prefix sum array that finds applications in data warehouses and histogram constructions, e.g., [10, 15]. Our study is also generally related with the management of interval and temporal data [1,5,8,16,16,22].

9. CONCLUSION

Temporal and multi-version databases often generate massive amounts of data. Therefore, it becomes increasingly important to store and process these data in a distributed and parallel fashion. This paper makes an important contribution in solving the optimal splitters problem, which is essential in enabling efficient distributed and parallel processing of such data. An interesting open problem is to extend our study to higher dimensions.

10. ACKNOWLEDGMENT

Wangchao Le, Feifei Li, and Robert Christensen were supported in part by NSF Grants IIS-0916488, IIS-1053979 and an REU supplement. Yufei Tao was supported in part by (i) projects GRF 4166/10, 4165/11, and 4164/12 from HKRGC, and (ii) the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007).

11. REFERENCES

[1] P. K. Agarwal, L. Arge, and K. Yi. An optimal dynamic interval stabbing-max data structure. In *SODA*, pages 803–812, 2005.

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

[3] A. Anagnostopoulos, M. Vlachos, M. Hadjieleftheriou, E. Keogh, and P. S. Yu. Global distance-based segmentation of trajectories. In *KDD*, 2006.

[4] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *ESA*, 2002.

[5] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[6] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDBJ*, 5(4):264–275, 1996.

[7] Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu. Indexable PLA for efficient similarity search. In *VLDB*, 2007.

[8] J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL databases with temporal logic. *ACM TODS*, 26(2):145–178, 2001.

[9] M. Flammini, G. Monaco, L. Moscardelli, H. Shachnai, M. Shalom, T. Tamir, and S. Zaks. Minimizing total busy time in parallel scheduling with application to optical networks. In *IPDPS*, 2009.

[10] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, pages 73–88, 1997.

[11] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, 1998.

[12] E. Keogh and et. al. The UCR time series classification/clustering homepage. www.cs.ucr.edu/∼eamonn/time_series_data.

[13] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *ICDM*, 2001.

[14] S. Khanna, S. Muthukrishnan, and S. Skiena. Efficient array partitioning. In *ICALP*, pages 616–626, 1997.

[15] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In *PODS*, pages 196–204, 2000.

[16] H.-P. Kriegel, M. Pötke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, 2000.

[17] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction time indexing with version compression. *PVLDB*, 1(1):870–881, 2008.

[18] S. Muthukrishnan and T. Suel. Approximation algorithms for array partitioning problems. *J. Algorithms*, 54(1):85–104, 2005.

[19] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, 2004.

[20] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, 1996.

[21] K. A. Ross and J. Cieslewicz. Optimal splitters for database partitioning with size bounds. In *ICDT*, pages 98–110, 2009.

[22] Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, 2001.

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