Optimal Splitters for Temporal and Multi-version Databases

Wangchao Le\textsuperscript{1}  Feifei Li\textsuperscript{1}  Yufei Tao\textsuperscript{2,3}  Robert Christensen\textsuperscript{1}

\textsuperscript{1}University of Utah  \textsuperscript{2}Chinese University of Hong Kong  \textsuperscript{3}Korea Adv. Inst. Sci & Tech

July 11, 2013
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing
Temporal and multi-version data are important in:

- financial market
- scientific application
- data warehousing

User applications:
- collect and query data in a long-running history of an object
- scale out by storing data in a distributed and parallel framework

Have to deal with data partitioning
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing

![Graph showing score over temporal data and time.](image)
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing

An object with 3 versions:
- update
- deletion
- insertion

User applications:
- collect and query data in a long-running history
- scale out by storing data in a distributed and parallel framework

Have to deal with data partitioning
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing

User applications:
- collect and query data in a long-running history
- an object \( o \rightarrow \) disjoint temporal intervals
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing

User applications:
- collect and query data in a long-running history
- an object \( o \rightarrow \) disjoint temporal intervals
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing

User applications:
- collect and query data in a long-running history
- an object $o \rightarrow$ disjoint temporal intervals
- scale out by storing data in a distributed and parallel framework
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing

User applications:
- collect and query data in a long-running history
- an object $o \rightarrow$ disjoint temporal intervals
- scale out by storing data in a distributed and parallel framework

Have to deal with data partitioning
Partition interval data into buckets based on time
- process queries w.r.t a given time with selected node(s)/core(s)
Problem Formulation

- Partition interval data into buckets based on time
  - process queries w.r.t a given time with selected node(s)/core(s)
- A size-\(k\) partition \(P\) over a set of intervals \(\mathcal{I}\), denoted as \(P(\mathcal{I}, k)\):
  - has \(k\) distinct vertical splitters and \(k + 1\) buckets

An example, \(k = 2\):

![Diagram showing three buckets and corresponding intervals for objects o1, o2, and o3.]
Partition interval data into buckets based on time
- process queries w.r.t a given time with selected node(s)/core(s)

A size-$k$ partition $P$ over a set of intervals $\mathcal{I}$, denoted as $P(\mathcal{I}, k)$:
- has $k$ distinct vertical splitters and $k + 1$ buckets

An example, $k = 2$

An interval $[s, e] \in b_i$ if it intersects $b_i$ ($b_i$ is a set of intervals)
Problem Formulation

- Partition interval data into buckets based on time
  - process queries w.r.t a given time with selected node(s)/core(s)
- A size-$k$ partition $P$ over a set of intervals $\mathcal{I}$, denoted as $P(\mathcal{I}, k)$:
  - has $k$ distinct vertical splitters and $k + 1$ buckets

An example, $k = 2$

- an interval $[s, e] \in b_i$ if it intersects $b_i$ ($b_i$ is a set of intervals)
Partition interval data into buckets based on time

- process queries w.r.t a given time with selected node(s)/core(s)

A size-\(k\) partition \(P\) over a set of intervals \(\mathcal{I}\), denoted as \(P(\mathcal{I}, k)\):

1. has \(k\) distinct vertical splitters and \(k + 1\) buckets

2. an interval \([s, e] \in b_i\) if it intersects \(b_i\) (\(b_i\) is a set of intervals)

3. **Cost** of a partition: \(c(P) = \max\{|b_1|, \ldots, |b_{k+1}|\}\)
Problem Formulation

- Partition interval data into buckets based on time
- process queries w.r.t a given time with selected node(s)/core(s)
- A size-$k$ partition $P$ over a set of intervals $I$, denoted as $P(I, k)$:
  - has $k$ distinct vertical splitters and $k + 1$ buckets
  - an interval $[s, e] \in b_i$ if it intersects $b_i$ ($b_i$ is a set of intervals)
  - **Cost** of a partition: $c(P) = \max\{|b_1|, \ldots, |b_{k+1}|\}$

An example, $k = 2$

$$C(P) = \max\{|b_1| = 3, |b_2| = 4, |b_3| = 5\} = 5$$
Load-balancing is important in a distributed setting.
Load-balancing is important in a distributed setting.
Objective: minimize the maximum load on a single node.

Definition

An **optimal partition** of size-\( k \) is a partition \( P^*(\mathcal{I}, k) \) with the smallest cost, i.e.

\[
P^*(\mathcal{I}, k) = \arg\min c(P)
\]
- **Load-balancing** is important in a distributed setting
- Objective: minimize the maximum load on a single node

**Definition**

An **optimal partition** of size-$k$ is a partition $P^*(\mathcal{I}, k)$ with the smallest cost, i.e.

$$P^*(\mathcal{I}, k) = \arg\min(c(P))$$

An example, $k = 2$

Optimal Splitters, $c(P) = 4$
Load-balancing is important in a distributed setting
Objective: minimize the maximum load on a single node

Definition

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

$$P^*(I, k) = \text{argmin}(c(P))$$

In this talk, our objective:

Find $P^*$ and $c(P^*)$ for $I$ and a fixed budget $k$
1 Motivation and Problem Formulation

2 A Baseline Method
   • Strategy to Place Splitters
   • Dynamic Programming Approach
   • Cost Analysis

3 Internal Memory Method
   • Cost-\(t\) Splitter Problem
   • Stabbing-count Array and \(t\)-jump method
   • Cost Analysis

4 External Memory Method
   • Concurrent \(t\)-jump method
   • Cost Analysis

5 Experiments

6 Conclusion
Outline

1 Motivation and Problem Formulation

2 A Baseline Method
   • Strategy to Place Splitters
   • Dynamic Programming Approach
   • Cost Analysis

3 Internal Memory Method
   • Cost-t Splitter Problem
   • Stabbing-count Array and t-jump method
   • Cost Analysis

4 External Memory Method
   • Concurrent t-jump method
   • Cost Analysis

5 Experiments

6 Conclusion
Strategy to Place Splitters

- Where to place splitters?
Where to place splitters?

let $I = \{[s_1, e_1]...[s_N, e_N]\}$, and let $S = \{s_1...s_N\}$ in ascending order.
Where to place splitters?

- let $I = \{[s_1, e_1],..., [s_N, e_N]\}$, and let $S = \{s_1, ..., s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$.
Where to place splitters?

- let $I = \{[s_1, e_1]...[s_N, e_N]\}$, and let $S = \{s_1...s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$
Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, and let $\mathbf{S} = \{s_1...s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$

Observation

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_k$. Let $\ell_i$ be the largest splitter that does not in $\mathbf{S}$. Define $P'$ from $P$ by replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.
Strategy to Place Splitters

Where to place splitters?

- Let $I = \{[s_1, e_1], \ldots, [s_N, e_N]\}$, and let $S = \{s_1, \ldots, s_N\}$ in ascending order.
- For any splitter $\ell$, let $\ell(1)$ be the smallest starting value $\ell$ such that $\ell(1) \geq \ell$.

Observation

For any partition $P$ with distinct splitters $\ell_1 < \ldots < \ell_k$. Let $\ell_i$ be the largest splitter that does not in $S$. Define $P'$ from $P$ by replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$. 

$c(P) = 5$
Where to place splitters?

- Let $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, and let $S = \{s_1...s_N\}$ in ascending order.
- For any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$.

**Observation**

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_k$. Let $\ell_i$ be the largest splitter that does not in $S$. Define $P'$ from $P$ by replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$. 

$$c(P) = 5$$
Where to place splitters?

- let $I = \{[s_1, e_1]...[s_N, e_N]\}$, and let $S = \{s_1...s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$

**Observation**

*For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_k$. Let $\ell_i$ be the largest splitter that does not in $S$. Define $P'$ from $P$ by replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.*
Where to place splitters?

- let \( \mathcal{I} = \{[s_1, e_1], \ldots, [s_N, e_N]\} \), and let \( S = \{s_1, \ldots, s_N\} \) in ascending order.
- for any splitter \( \ell \), let \( \ell(1) \) be the smallest starting value s.t. \( \ell(1) \geq \ell \).

Observation

For any partition \( P \) with distinct splitters \( \ell_1 < \ldots < \ell_k \). Let \( \ell_i \) be the largest splitter that does not in \( S \). Define \( P' \) from \( P \) by replacing \( \ell_i \) with \( \ell_i(1) \). Then, \( c(P') \leq c(P) \).
Where to place splitters?

- let \( \mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\} \), and let \( S = \{s_1...s_N\} \) in ascending order.
- for any splitter \( \ell \), let \( \ell(1) \) be the smallest starting value s.t. \( \ell(1) \geq \ell \)

Observation

For any partition \( P \) with distinct splitters \( \ell_1 < ... < \ell_k \). Let \( \ell_i \) be the largest splitter that does not in \( S \). Define \( P' \) from \( P \) by replacing \( \ell_i \) with \( \ell_i(1) \). Then, \( c(P') \leq c(P) \).

Should always try to split on \( S \)!
Outline

1 Motivation and Problem Formulation

2 A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis

3 Internal Memory Method
   - Cost-\(t\) Splitter Problem
   - Stabbing-count Array and \(t\)-jump method
   - Cost Analysis

4 External Memory Method
   - Concurrent \(t\)-jump method
   - Cost Analysis

5 Experiments

6 Conclusion
Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array.
Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array $\mathcal{I}^{-}(\ell) = \{[s_i, e_i] \in I | s_i < \ell\}$.
Dynamic Programming Approach

Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

- $\mathcal{I}^{-}(\ell) = \{[s_i, e_i] \in I | s_i < \ell \}$
- $\mathcal{I}^{+}(\ell) = \{[s_i, e_i] \in I | s_i > \ell \}$
Given a splitter $\ell$ and a set of intervals $I$ stored in an array:

- $I^{-}(\ell) = \{[s_i, e_i] \in I | s_i < \ell\}$
- $I^{+}(\ell) = \{[s_i, e_i] \in I | s_i > \ell\}$
- $I^{o}(\ell) = \{[s_i, e_i] \in I | s_i = \ell\}$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

\[
\ell
\]

\[
\mathcal{I}^{-}(\ell) = \{ [s_i, e_i] \in I | s_i < \ell \}
\]

\[
\mathcal{I}^{+}(\ell) = \{ [s_i, e_i] \in I | s_i > \ell \}
\]

\[
\mathcal{I}^{o}(\ell) = \{ [s_i, e_i] \in I | s_i = \ell \}
\]

\[
\mathcal{I}^{x}(\ell) = \{ [s_i, e_i] \in I | s_i < \ell < e_i \}
\]
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array
  - $\mathcal{I}^-(\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i < \ell \}$
  - $\mathcal{I}^+(\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i > \ell \}$
  - $\mathcal{I}^o(\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i = \ell \}$
  - $\mathcal{I}^x(\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i < \ell < e_i \}$

- Dynamic programming
Dynamic Programming Approach

Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

- $\mathcal{I}^- (\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i < \ell \}$
- $\mathcal{I}^+ (\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i > \ell \}$
- $\mathcal{I}^o (\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i = \ell \}$
- $\mathcal{I}^x (\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i < \ell < e_i \}$

Dynamic programming

$c(P^*(\mathcal{I}, k))$

- $\ell_k$

LastBucket = $|\mathcal{I}^o (\ell) + \mathcal{I}^x (\ell) + \mathcal{I}^+ (\ell)|$

A sub-problem: $c(P^*(\mathcal{I}^- (\ell_k), k - 1))$

- How many ways to place $\ell_k$? $\ell_k \in S(I)$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $I$ stored in an array

$$
I^-(\ell) = \{[s_i, e_i] \in I | s_i < \ell\}
$$
$$
I^+(\ell) = \{[s_i, e_i] \in I | s_i > \ell\}
$$
$$
I^o(\ell) = \{[s_i, e_i] \in I | s_i = \ell\}
$$
$$
I^x(\ell) = \{[s_i, e_i] \in I | s_i < \ell < e_i\}
$$

- Dynamic programming

$$
c(P^*(I, k)) = \max \{c(P^*(I^-(\ell_k), k - 1), \text{LastBucket})\}
$$

- How many ways to place $\ell_k$? $\ell_k \in S(I)$

| LastBucket | $|I^o(\ell) + I^x(\ell) + I^+(\ell)|$ |
|------------|----------------------------------|

A sub-problem: $c(P^*(I^-(\ell_k), k - 1))$
Dynamic Programming Approach

- Given a splitter \( \ell \) and a set of intervals \( \mathcal{I} \) stored in an array

\[
\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I | s_i < \ell \}
\]

\[
\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I | s_i > \ell \}
\]

\[
\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I | s_i = \ell \}
\]

\[
\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I | s_i < \ell < e_i \}
\]

- Dynamic programming

\[
c(P^*(\mathcal{I}, k)) = \max \{c(P^*(\mathcal{I}^-(\ell_k), k - 1), \text{LastBucket})\}
\]

- How many ways to place \( \ell_k \)? \( \ell_k \in S(I) \)
Dynamic Programming Approach

- Given a splitter \( \ell \) and a set of intervals \( \mathcal{I} \) stored in an array:

  \[
  \mathcal{I}^{(\ell)} = \{ [s_i, e_i] \in \mathcal{I} | s_i = \ell \}
  \]

  \[
  \mathcal{I}^+(\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i > \ell \}
  \]

  \[
  \mathcal{I}^-(\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i < \ell \}
  \]

  \[
  \mathcal{I}^x(\ell) = \{ [s_i, e_i] \in \mathcal{I} | s_i < \ell < e_i \}
  \]

- Dynamic programming:

  \[
  c(P^*(\mathcal{I}, k)) = \max\left\{ c(P^*(\mathcal{I}^-(\ell_k), k - 1), \text{LastBucket}) \right\}
  \]

  - How many ways to place \( \ell_k \)? \( \ell_k \in S(\mathcal{I}) \)

  \[
  \text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|
  \]

  A sub-problem: \( c(P^*(\mathcal{I}^-(\ell_k), k - 1)) \)
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

\[
\ell
\]

- Dynamic programming

\[
c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^{-}(\ell_k), k - 1), \text{LastBucket})\}
\]

- How many ways to place $\ell_k$? $\ell_k \in S(I)$

\[
\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|
\]

\[
\text{A sub-problem: } c(P^*(\mathcal{I}^{-}(\ell_k), k - 1))
\]
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $I$ stored in an array

  \[ I^-(\ell) = \{ [s_i, e_i] \in I | s_i < \ell \} \]
  \[ I^+(\ell) = \{ [s_i, e_i] \in I | s_i > \ell \} \]
  \[ I^0(\ell) = \{ [s_i, e_i] \in I | s_i = \ell \} \]
  \[ I^x(\ell) = \{ [s_i, e_i] \in I | s_i < \ell < e_i \} \]

- Dynamic programming

  \[ c(P^*(I, k)) = \max\{ c(P^*(I^-(\ell_k), k - 1), \text{LastBucket}) \} \]

  \[ \text{LastBucket} = |I^0(\ell) + I^x(\ell) + I^+(\ell)| \]

  \[ \text{A sub-problem: } c(P^*(I^-(\ell_k), k - 1)) \]

  - How many ways to place $\ell_k$? $\ell_k \in S(I)$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array:

  $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I | s_i < \ell\}$

  $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I | s_i > \ell\}$

  $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I | s_i = \ell\}$

  $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I | s_i < \ell < e_i\}$

- Dynamic programming:

  $$c(P^*(\mathcal{I}, k)) = \max\{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\}$$

- How many ways to place $\ell_k$? $\ell_k \in S(I)$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

- $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I | s_i < \ell \}$
- $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I | s_i > \ell \}$
- $\mathcal{I}^0(\ell) = \{[s_i, e_i] \in I | s_i = \ell \}$
- $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I | s_i < \ell < e_i \}$

- Dynamic programming

\[
c(P^*(\mathcal{I}, k)) = \max_{\ell_k} \{c(P^*(\mathcal{I}^-(\ell_k), k - 1), \text{LastBucket})\}
\]

- How many ways to place $\ell_k$? $\ell_k \in S(I)$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

  \[\mathcal{I}-(\ell) = \{[s_i, e_i] \in I | s_i < \ell\}\]
  \[\mathcal{I}+ (\ell) = \{[s_i, e_i] \in I | s_i > \ell\}\]
  \[\mathcal{I}^0 (\ell) = \{[s_i, e_i] \in I | s_i = \ell\}\]
  \[\mathcal{I}^x (\ell) = \{[s_i, e_i] \in I | s_i < \ell < e_i\}\]

- Dynamic programming

  \[c(P^*(\mathcal{I}, k)) = \max \{c(P^*(\mathcal{I}-(\ell_k), k-1), \text{LastBucket})\}\]

  LastBucket = $|\mathcal{I}^0 (\ell) + \mathcal{I}^x (\ell) + \mathcal{I}^+ (\ell)|$

  A sub-problem: $c(P^*(\mathcal{I}-(\ell_k), k-1))$

- How many ways to place $\ell_k$? $\ell_k \in S(I)$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $I$ stored in an array

$$\ell$$

- $I^-(\ell) = \{ [s_i, e_i] \in I | s_i < \ell \}$
- $I^+(\ell) = \{ [s_i, e_i] \in I | s_i > \ell \}$
- $I^o(\ell) = \{ [s_i, e_i] \in I | s_i = \ell \}$
- $I^x(\ell) = \{ [s_i, e_i] \in I | s_i < \ell < e_i \}$

- Dynamic programming

$$c(P^*(I, k)) = \max \{ c(P^*(I^-(\ell_k), k - 1), \text{LastBucket}) \}$$

- How many ways to place $\ell_k$? $\ell_k \in S(I)$

$\text{LastBucket} = |I^o(\ell) + I^x(\ell) + I^+(\ell)|$

A sub-problem: $c(P^*(I^-(\ell_k), k - 1))$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

$\mathcal{I}^-(\ell) = \{ [s_i, e_i] \in I | s_i < \ell \}$
$\mathcal{I}^+(\ell) = \{ [s_i, e_i] \in I | s_i > \ell \}$
$\mathcal{I}^o(\ell) = \{ [s_i, e_i] \in I | s_i = \ell \}$
$\mathcal{I}^x(\ell) = \{ [s_i, e_i] \in I | s_i < \ell < e_i \}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \max \{ c(P^*(\mathcal{I}^-(\ell_k), k - 1), \text{LastBucket}) \}$$

$\text{LastBucket} = |\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$

- How many ways to place $\ell_k$? $\ell_k \in S(I)$
Dynamic Programming Approach

- Given a splitter $\ell$ and a set of intervals $\mathcal{I}$ stored in an array

  $\mathcal{I}^-(\ell) = \{[s_i, e_i] \in I | s_i < \ell\}$
  $\mathcal{I}^+(\ell) = \{[s_i, e_i] \in I | s_i > \ell\}$
  $\mathcal{I}^o(\ell) = \{[s_i, e_i] \in I | s_i = \ell\}$
  $\mathcal{I}^x(\ell) = \{[s_i, e_i] \in I | s_i < \ell < e_i\}$

- Dynamic programming

$$c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in S(I)} \{\max \{c(P^*(\mathcal{I}^-(\ell_k), k - 1), \text{LastBucket})\}\}$$

  LastBucket = $|\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|$

  A sub-problem: $c(P^*(\mathcal{I}^-(\ell_k), k - 1))$

- How many ways to place $\ell_k$? $\ell_k \in S(I)$
Outline

1 Motivation and Problem Formulation

2 A Baseline Method
   • Strategy to Place Splitters
   • Dynamic Programming Approach
   • Cost Analysis

3 Internal Memory Method
   • Cost-\(t\) Splitter Problem
   • Stabbing-count Array and \(t\)-jump method
   • Cost Analysis

4 External Memory Method
   • Concurrent \(t\)-jump method
   • Cost Analysis

5 Experiments

6 Conclusion
Cost Analysis

- A common sub-problem may appear more than one time
A common sub-problem may appear more than one time

- Memoization

Cost of the DP approach

\[
c(P^*(I, k)) = \min_{\ell \in S} \{\max\{c(P^*(I - (\ell_k), k - 1)), \text{LastBucket}\}\}
\]

1. to fill in Cell\[i, j]\), need to check \(i - 1\) preceding rows
2. \(O(1)\) cost to obtain LastBucket (\(|I_o(\ell) + I_x(\ell) + I_+(\ell)|\)
3. \(O(kN^2)\) for DP

\(|S| = N, k\) splitters
A common sub-problem may appear more than one time

- **Memoization**

\[
\begin{array}{cccc}
[1,1] & \cdots & [1,k-1] & [1,k] \\
\vdots & \ddots & \vdots & \vdots \\
[N,1] & \cdots & [N,k-1] & [N,k] \\
\end{array}
\]

\[|S| = N, \text{ } k \text{ splitters}\]

Cost of the DP approach

\[
c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in S} \left\{ \max \{c(P^*(\mathcal{I}^-(\ell_k), k-1), \text{LastBucket})\} \right\}
\]
A common sub-problem may appear more than one time

- Memoization

Cost of the DP approach

\[ c(P^*(I, k)) = \min_{\ell_k \in S} \{ \max \{ c(P^*(I^-(\ell_k), k - 1), \text{LastBucket}) \} \} \]

- to fill in Cell\([i, j]\), need to check \(i - 1\) preceding rows
A common sub-problem may appear more than one time

- Memoization

Cost of the DP approach

\[ c(P^*(\mathcal{I}, k)) = \min_{\ell_k \in S} \{ \max \{ c(P^*(\mathcal{I}^-(\ell_k), k - 1), \text{LastBucket}) \} \} \]

1. to fill in Cell\([i, j]\), need to check \(i - 1\) preceding rows
2. \(O(1)\) cost to obtain \(\text{LastBucket} (|\mathcal{I}^o(\ell) + \mathcal{I}^x(\ell) + \mathcal{I}^+(\ell)|)\)
Cost Analysis

- A common sub-problem may appear more than one time
  - Memoization

\[
\begin{array}{cccc}
[1, 1] & \cdots & [1, k - 1] & [1, k] \\
[1, k] & \cdots & [N, k - 1] & [N, k] \\
[N, 1] & \cdots & [N, k - 1] & [N, k] \\
\end{array}
\]

\[|S| = N, k \text{ splitters}\]

- Cost of the DP approach

\[
c(P^*(I, k)) = \min_{\ell_k \in S} \left\{ \max \left\{ c(P^*(I^-(\ell_k), k - 1), \text{LastBucket}) \right\} \right\}
\]

1. to fill in Cell[i, j], need to check \(i - 1\) preceding rows
2. \(O(1)\) cost to obtain LastBucket (\(|I^o(\ell) + I^x(\ell) + I^+(\ell)|\))
3. \(O(kN^2)\) for DP
1 Motivation and Problem Formulation

2 A Baseline Method
   • Strategy to Place Splitters
   • Dynamic Programming Approach
   • Cost Analysis

3 Internal Memory Method
   • Cost-\( t \) Splitter Problem
   • Stabbing-count Array and \( t \)-jump method
   • Cost Analysis

4 External Memory Method
   • Concurrent \( t \)-jump method
   • Cost Analysis

5 Experiments

6 Conclusion
Outline

1. Motivation and Problem Formulation

2. A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis

3. Internal Memory Method
   - Cost-\(t\) Splitter Problem
   - Stabbing-count Array and \(t\)-jump method
   - Cost Analysis

4. External Memory Method
   - Concurrent \(t\)-jump method
   - Cost Analysis

5. Experiments

6. Conclusion
Cost-\(t\) Splitter Problem

A decision version of our problem:

**Definition** (*Cost-\(t\) splitters problem*)

Determine whether there is a size-\(k\) partition \(P\) with \(c(P) \leq t\)
A decision version of our problem:

**Definition (Cost-t splitters problem)**

Determine whether there is a size-\(k\) partition \(P\) with \(c(P) \leq t\)

- if such \(P\) exists, \(t\) is **feasible**
  - **Output:** \(\bar{t} \in [1, t]\) s.t. \(\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}\)
- otherwise, \(t\) is **infeasible**
  - **Output:** \(\bar{t} = 0\)
Cost-$t$ Splitter Problem

A decision version of our problem:

**Definition (Cost-$t$ splitters problem)**
Determine whether there is a size-$k$ partition $P$ with $c(P) \leq t$

1. if such $P$ exists, $t$ is **feasible**
   - Output: $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
2. otherwise, $t$ is **infeasible**
   - Output: $\bar{t} = 0$

**Lemma**

If $t$ is infeasible, then any $t' < t$ is also infeasible
Cost-\(t\) Splitter Problem

A decision version of our problem:

**Definition (Cost-\(t\) splitters problem)**

Determine whether there is a size-\(k\) partition \(P\) with \(c(P) \leq t\)

- if such \(P\) exists, \(t\) is **feasible**
  - **Output**: \(\bar{t} \in [1, t] \) s.t. \(\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}\)

- otherwise, \(t\) is **infeasible**
  - **Output**: \(\bar{t} = 0\)

**Lemma**

*If \(t\) is infeasible, then any \(t' < t\) is also infeasible*

**Sketch of the Algorithm:**
Cost-$t$ Splitter Problem

A decision version of our problem:

**Definition (Cost-$t$ splitters problem)**
Determine whether there is a size-$k$ partition $P$ with $c(P) \leq t$

- if such $P$ exists, $t$ is **feasible**
  - Output: $\bar{t} \in [1, t]$ s.t. $\exists P \in \mathcal{P}(I, k), c(P) = \bar{t}$
- otherwise, $t$ is **infeasible**
  - Output: $\bar{t} = 0$

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

**Sketch of the Algorithm:**
- The optimal cost $t^*$ is in the range of $R = [1, N]$
Cost-\( t \) Splitter Problem

A decision version of our problem:

**Definition (Cost-\( t \) splitters problem)**

Determine whether there is a size-\( k \) partition \( P \) with \( c(P) \leq t \)

1. if such \( P \) exists, \( t \) is feasible
   - **Output**: \( \bar{t} \in [1, t] \) s.t. \( \exists P \in \mathcal{P}(I, k), c(P) = \bar{t} \)
2. otherwise, \( t \) is infeasible
   - **Output**: \( \bar{t} = 0 \)

**Lemma**

If \( t \) is infeasible, then any \( t' < t \) is also infeasible

**Sketch of the Algorithm:**

1. The optimal cost \( t^* \) is in the range of \( R = [1, N] \)
2. Binary search on \( R \)
3. Solve \( O(\log N) \) instances of Cost-\( t \) splitters problem
The optimal cost $t^*$ is in the range of $R = [1, N]$

2. Binary search on $R$

3. Solve $O(\log N)$ instances of Cost-$t$ splitters problem

4. Report $t^*$, when $t^*$ is feasible but $t^* - 1$ is infeasible
Cost-\( t \) Splitter Problem

A decision version of our problem:

**Definition (Cost-\( t \) splitters problem)**

Determine whether there is a size-\( k \) partition \( P \) with \( c(P) \leq t \)

- if such \( P \) exists, \( t \) is **feasible**
  - Output: \( \bar{t} \in [1, t] \) s.t. \( \exists P \in \mathcal{P}(I, k), c(P) = \bar{t} \)
- otherwise, \( t \) is **infeasible**
  - Output: \( \bar{t} = 0 \)

**Lemma**

*If \( t \) is infeasible, then any \( t' < t \) is also infeasible*

**Sketch of the Algorithm:**

1. The optimal cost \( t^* \) is in the range of \( R = [1, N] \)
2. Binary search on \( R \)
3. Solve \( O(\log N) \) instances of Cost-\( t \) splitters problem
4. Report \( t^* \), when \( t^* \) is feasible but \( t^* - 1 \) is infeasible
Outline

1. Motivation and Problem Formulation
2. A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis
3. Internal Memory Method
   - Cost-\(t\) Splitter Problem
   - Stabbing-count Array and \(t\)-jump method
   - Cost Analysis
4. External Memory Method
   - Concurrent \(t\)-jump method
   - Cost Analysis
5. Experiments
6. Conclusion
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$
  - by non-descending order of $s_i$'s
  - break ties by non-descending order of $e_i$'s
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$
  - by non-descending order of $s_i$’s
  - break ties by non-descending order of $e_i$’s

- The stabbing-count array for $\mathcal{I}$
  - $\forall s_i \in \mathcal{I}$, maintain two counts $\sigma$, $\delta$
    - $\triangleright \sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
    - $\triangleright \delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1], \ldots, [s_N, e_N]\}$
  - by non-descending order of $s_i$’s
  - break ties by non-descending order of $e_i$’s
- The stabbing-count array for $\mathcal{I}$
  - $\forall s_i \in \mathcal{I}$, maintain two counts $\sigma$, $\delta$
    - $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
    - $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$

![Diagram showing stabbing-count array with intervals $s_1, s_2, s_3, s_4$ and corresponding stabbing-count values $\sigma[3] = 2$.]
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$
  - by non-descending order of $s_i$'s
  - break ties by non-descending order of $e_i$'s
- The stabbing-count array for $\mathcal{I}$
  - $\forall s_i \in \mathcal{I}$, maintain two counts $\sigma$, $\delta$
    - $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
    - $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$

  $\sigma[3] = 2$, $\delta[3] = 0$
Sort $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$
- by non-descending order of $s_i$'s
- break ties by non-descending order of $e_i$'s

The stabbing-count array for $\mathcal{I}$
- $\forall s_i \in \mathcal{I}$, maintain two counts $\sigma$, $\delta$
  - $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
  - $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$

$\sigma[4] = 2$,

\[
s_1 \quad s_2 \quad s_3 \quad s_4 \quad e_1 \quad e_2 \quad e_3 \quad e_4
\]
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$
  - by non-descending order of $s_i$'s
  - break ties by non-descending order of $e_i$'s

- The stabbing-count array for $\mathcal{I}$
  - $\forall s_i \in \mathcal{I}$, maintain two counts $\sigma$, $\delta$
    - $\sigma[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
    - $\delta[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$

\[
\sigma[4] = 2, \delta[4] = 1
\]
Stabbing-count Array

- Sort $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$ [\(O(N \log N)\) time]
  - by non-descending order of \(s_i\)'s
  - break ties by non-descending order of \(e_i\)'s
- The stabbing-count array for $\mathcal{I}$ [\(O(N)\) time]
  - \(\forall s_i \in \mathcal{I}\), maintain two counts \(\sigma, \delta\)
    - \(\sigma[i] = |\mathcal{I}^< (s_i)|\), \# intervals intersecting \(s_i\)
    - \(\delta[i] = |\mathcal{I}^< (s_i)|\), \# intervals in \(\mathcal{I}^< (s_i)\) with ids less than \(i\)

\[
\sigma[4] = 2, \; \delta[4] = 1
\]
Stabbing-count Array

- Sort $I = \{[s_1, e_1]...[s_N, e_N]\}$ \[O(N \log N) \text{ time}\]
  - by non-descending order of $s_i$'s
  - break ties by non-descending order of $e_i$'s
- The stabbing-count array for $I$ \[O(N) \text{ time}\]
  - $\forall s_i \in I$, maintain two counts $\sigma$, $\delta$
    - $\sigma[i] = |I^x(s_i)|$, \# intervals intersecting $s_i$
    - $\delta[i] = |I^o(s_i)|$, \# intervals in $I^o(s_i)$ with ids less than $i$

\[\sigma[4] = 2, \delta[4] = 1\]

Lemma

*The stabbing-count array can be built in $O(N \log N)$ time*
- **t-jump method**

  - t-jump method
t-jump method

1. solves an instance of the Cost-t splitters problem
2. if feasible, output the feasible $P$ and $c(P)$
t-jump method

1. solves an instance of the Cost-t splitters problem
2. if feasible, output the feasible $P$ and $c(P)$
3. a greedy algorithm
*t*-jump method

1. solves an instance of the Cost-*t* splitters problem
2. if feasible, output the feasible *P* and *c(P)*
3. a greedy algorithm

**Intuition**

- Place splitters in ascending order
- \( \ell_{i+1} \) is pushed as far as possible from \( \ell_i \), let each new \( b_i \) have size \( t \)
- If not achievable, move \( \ell_{i+1} \) backward just enough to form the new \( b_i \)
t-jump method

- t-jump method
  1. solves an instance of the Cost-$t$ splitters problem
  2. if **feasible**, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

Intuition

- place splitters in ascending order
### t-jump method

- **t-jump method**
  1. solves an instance of the Cost-\( t \) splitters problem
  2. if **feasible**, output the feasible \( P \) and \( c(P) \)
  3. a greedy algorithm

---

#### Intuition

- place splitters in ascending order
- \( \ell_{i+1} \) is pushed as far as possible from \( \ell_i \), let each new \( b_i \) have size \( t \)
- **t-jump method**
  1. solves an instance of the Cost-\( t \) splitters problem
  2. if **feasible**, output the feasible \( P \) and \( c(P) \)
  3. a greedy algorithm

**Intuition**

1. place splitters in ascending order
2. \( \ell_{i+1} \) is pushed as far as possible from \( \ell_i \), let each new \( b_i \) have size \( t \)
3. if not achievable, move \( \ell_{i+1} \) backward just enough to form the new \( b_i \)
t-jump method

- t-jump method
  1. solves an instance of the Cost-t splitters problem
  2. if feasible, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

### Intuition

1. place splitters in ascending order
2. $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
3. if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i$
**t-jump method**

- **t-jump method**
  1. solves an instance of the Cost-\(t\) splitters problem
  2. if feasible, output the feasible \(P\) and \(c(P)\)
  3. a greedy algorithm

**Intuition**

- place splitters in ascending order
- \(\ell_{i+1}\) is pushed as far as possible from \(\ell_i\), let each new \(b_i\) have size \(t\)
- if not achievable, move \(\ell_{i+1}\) backward just enough to form the new \(b_i\)
**t-jump method**

- **t-jump method**
  1. solves an instance of the Cost-$t$ splitters problem
  2. if feasible, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

\[ \sigma[4] = 1, \text{jump at most } t - \sigma[4] = 2 \text{ ids} \]

![Diagram of splitters](image)

\[
|b_1| = 3 \\
k = 2, \ t = 3
\]

**Intuition**

1. place splitters in ascending order
2. $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
3. if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i$
**t-jump method**

- **t-jump method**
  1. solves an instance of the Cost-$t$ splitters problem
  2. if **feasible**, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

![Diagram showing the t-jump method](image)

$k = 2, t = 3$

**Intuition**

1. place splitters in ascending order
2. $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
3. if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i$


- **t-jump method**
  1. solves an instance of the Cost-$t$ splitters problem
  2. if feasible, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

![Diagram showing placement of splitters]

**Intuition**

1. place splitters in ascending order
2. $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
3. if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i$
**t-jump method**

- **1.** Solves an instance of the Cost-\( t \) splitters problem
- **2.** If **feasible**, output the feasible \( P \) and \( c(P) \)
- **3.** A greedy algorithm

Intuition

- **1.** Place splitters in ascending order
- **2.** \( \ell_{i+1} \) is pushed as far as possible from \( \ell_i \), let each new \( b_i \) have size \( t \)
- **3.** If not achievable, move \( \ell_{i+1} \) backward just enough to form the new \( b_i \)
- \textit{t-jump method}
  - 1. solves an instance of the Cost-\(t\) splitters problem
  - 2. if \textbf{feasible}, output the feasible \(P\) and \(c(P)\)
  - 3. a greedy algorithm

\[k = 2, \ t = 2\]

\textbf{Intuition}

1. place splitters in ascending order
2. \(\ell_{i+1}\) is pushed as far as possible from \(\ell_i\), let each new \(b_i\) have size \(t\)
3. if not achievable, move \(\ell_{i+1}\) backward just enough to form the new \(b_i\)
- **t-jump method**
  1. solves an instance of the Cost-\( t \) splitters problem
  2. if **feasible**, output the feasible \( P \) and \( c(P) \)
  3. a greedy algorithm

\[
|b_1| = 2
\]

![Diagram](image)

\( k = 2, \ t = 2 \)

**Intuition**

1. place splitters in ascending order
2. \( \ell_{i+1} \) is pushed as far as possible from \( \ell_i \), let each new \( b_i \) have size \( t \)
3. if not achievable, move \( \ell_{i+1} \) backward just enough to form the new \( b_i \)
**t-jump method**

- **t-jump method**
  1. solves an instance of the Cost-$t$ splitters problem
  2. if **feasible**, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

**jump $t = 2$ ids**

\[ |b_1| = 2 \]

**Intuition**

1. place splitters in ascending order
2. $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
3. if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i$
$t$-jump method

- $t$-jump method
  1. solves an instance of the Cost-$t$ splitters problem
  2. if feasible, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

```
jump $t = 2$ ids, move back $\delta[5] = 1$
```

Intuition

- place splitters in ascending order
- $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
- if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i$
**t-jump method**

- **t-jump method**
  1. solves an instance of the Cost-\(t\) splitters problem
  2. if **feasible**, output the feasible \(P\) and \(c(P)\)
  3. a greedy algorithm

\[
\text{jump } t = 2 \text{ ids, move back } \delta[5] = 1 \]

\[
\begin{align*}
|b_1| &= 2 \\
|b_2| &= 1 \\
\ell_1 &\quad s_3 \\
\ell_2 &\quad s_4 \\
\ell_1 &\quad s_3 \\
\ell_2 &\quad s_4 \\
\end{align*}
\]

\(k = 2, \ t = 2\)

**Intuition**

1. place splitters in ascending order
2. \(\ell_{i+1}\) is pushed as far as possible from \(\ell_i\), let each new \(b_i\) have size \(t\)
3. if not achievable, move \(\ell_{i+1}\) backward just enough to form the new \(b_i\)
**t-jump method**

- **t-jump method**
  1. solves an instance of the Cost-$t$ splitters problem
  2. if **feasible**, output the feasible $P$ and $c(P)$
  3. a greedy algorithm

```
jump $t = 2$ ids, move back $\delta[5] = 1$
```

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

$k = 2, t = 2$

**Intuition**

1. place splitters in ascending order
2. $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
3. if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i$
### t-jump method

1. solves an instance of the Cost-$t$ splitters problem
2. if **feasible**, output the feasible $P$ and $c(P)$
3. a greedy algorithm

```
jump $t = 2$ ids, move back $\delta[5] = 1$
```

```
<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```
k = 2, t = 2
```

### Intuition

1. place splitters in ascending order
2. $\ell_{i+1}$ is pushed as far as possible from $\ell_i$, let each new $b_i$ have size $t$
3. if not achievable, move $\ell_{i+1}$ backward just enough to form the new $b_i`
$t$-jump method

- solves an instance of the Cost-$t$ splitters problem
- if feasible, output the feasible $P$ and $c(P)$
- a greedy algorithm

**jump $t = 2$ ids, move back $\delta[5] = 1$**

$|b_1| = 2$  $|b_2| = 1$  $|b_3| = 5$

$k = 2, t = 2$

$t = 2$ is infeasible

**Lemma (Correctness of $t$-jump)**

*If $t$-jump returns feasible, then the splitters output constitute a partition with cost $\bar{t} \leq t$. Otherwise, $t$ must be infeasible.*
# Outline

1. Motivation and Problem Formulation
2. A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis
3. Internal Memory Method
   - Cost-$t$ Splitter Problem
   - Stabbing-count Array and $t$-jump method
   - Cost Analysis
4. External Memory Method
   - Concurrent $t$-jump method
   - Cost Analysis
5. Experiments
6. Conclusion
Cost Analysis

1. The cost of \( O(N \log N) \) instances of Cost-t-splitters problems in a binary search can be answered in \( O(k) \) (where \( k \) is the number of splitters), \( O(k \log N) \) in total (where \( k \ll N \)).

2. Sorting intervals and constructing the stabbing-count array take \( O(N \log N) \) time.

Theorem: The problem of finding optimal splitters can be solved in \( O(N \log N) \) time in internal memory.
1. $O(\log N)$ instances of Cost-$t$ splitters problems in a binary search
2. Cost-$t$ splitters problem can be answered in $O(k)$ ($k$ is # splitters), $O(k \log N)$ in total ($k \ll N$)
\( O(\log N) \) instances of Cost-t splitters problems in a binary search

Cost-t splitters problem can be answered in \( O(k) \) (\( k \) is \# splitters), \( O(k \log N) \) in total (\( k \ll N \))

Sorting intervals and constructing the stabbing-count array take \( O(N \log N) \) time
1. $O(\log N)$ instances of Cost-$t$ splitters problems in a binary search
2. Cost-$t$ splitters problem can be answered in $O(k)$ ($k$ is \# splitters), $O(k \log N)$ in total ($k \ll N$)
3. Sorting intervals and constructing the stabbing-count array take $O(N \log N)$ time

**Theorem**

The problem of finding optimal splitters can be solved in $O(N \log N)$ time in internal memory.
Outline

1 Motivation and Problem Formulation

2 A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis

3 Internal Memory Method
   - Cost-$t$ Splitter Problem
   - Stabbing-count Array and $t$-jump method
   - Cost Analysis

4 External Memory Method
   - Concurrent $t$-jump method
   - Cost Analysis

5 Experiments

6 Conclusion
$I$ stored in a disk-resident array using $O(N/B)$ blocks
- $\mathcal{I}$ stored in a disk-resident array using $O(N/B)$ blocks
- Define the cost of external sorting as

$$SORT(N) = (N/B) \log_{M/B}(N/B)$$
I stored in a disk-resident array using $O(N/B)$ blocks

Define the cost of external sorting as

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

Theorem

*The problem of finding optimal splitters can be solved using $O(SORT(N))$ I/Os in external memory*
$I$ stored in a disk-resident array using $O(N/B)$ blocks

Define the cost of external sorting as

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

**Theorem**

*The problem of finding optimal splitters can be solved using $O(SORT(N))$ I/Os in external memory*

**Adapting the main-memory algorithm?**

1. sorting takes $SORT(N)$ I/Os
2. solving a cost-$t$ splitters problem takes $O(min(k, N/B))$ I/Os
3. $O(SORT(N) + min(k, N/B) \log N)$ I/Os in total
$\mathcal{I}$ stored in a disk-resident array using $O(N/B)$ blocks

Define the cost of external sorting as

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

**Theorem**

The problem of finding optimal splitters can be solved using $O(SORT(N))$ I/Os in external memory

**Adapting the main-memory algorithm?**

1. sorting takes $SORT(N)$ I/Os
2. solving a cost-$t$ splitters problem takes $O(min(k, N/B))$ I/Os
3. $O(SORT(N) + min(k, N/B) \log N)$ I/Os in total

**Problems**

- not a clean bound when $k \in [1, N]$
- may require excessive I/Os
Outline

1 Motivation and Problem Formulation

2 A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis

3 Internal Memory Method
   - Cost-$t$ Splitter Problem
   - Stabbing-count Array and $t$-jump method
   - Cost Analysis

4 **External Memory Method**
   - Concurrent $t$-jump method
   - Cost Analysis

5 Experiments

6 Conclusion
Concurrent \( t \)-jump method

**Definition (Cost-\( t \) testing)**

Determine whether there is a size-\( k \) partition \( P \) with \( c(P) \leq t \)

1. if such \( P \) exists, output \textbf{Yes}
2. otherwise, output \textbf{No}
Concurrent $t$-jump method

Definition (**Cost-**$t$ **testing**)

Determine whether there is a size-$k$ partition $P$ with $c(P) \leq t$

1. if such $P$ exists, output **Yes**
2. otherwise, output **No**

**Cost-**$t$ **Testing** vs. **Cost-**$t$ **Splitters Problem**

- avoid storing the feasible splitters ($O(k/B)$ space)
- lead to the concurrent extension of cost-$t$ testing
Concurrent \( t\)-jump method

Definition (\textbf{Cost-}\( t\) \textbf{testing})
Determine whether there is a size-\( k \) partition \( P \) with \( c(P) \leq t \)

1. if such \( P \) exists, output \textbf{Yes}
2. otherwise, output \textbf{No}

\textbf{Cost-}\( t \) \textbf{Testing} vs. \textbf{Cost-}\( t \) \textbf{Splitters Problem}
- avoid storing the feasible splitters (\( O(k/B) \) space)
- lead to the concurrent extension of cost-\( t \) testing

Intuition of concurrent \( t\)-jump

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intervals and Stabbing-Count Array on Disk

Wangchao Le  Feifei Li  Yufei Tao  Robert Christensen  Optimal Splitters for Temporal and Multi-version Databases
Concurrent \textit{t}-jump method

**Definition (Cost-\textit{t} testing)**
Determine whether there is a size-\(k\) partition \(P\) with \(c(P) \leq t\)

1. if such \(P\) exists, output \textbf{Yes}
2. otherwise, output \textbf{No}

**Cost-\textit{t} Testing vs. Cost-\textit{t} Splitters Problem**
- avoid storing the feasible splitters (\(O(k/B)\) space)
- lead to the concurrent extension of cost-\textit{t} testing

![Intuition of concurrent t-jump](image)

Intervals and Stabbing-Count Array on Disk

- \textit{t}-jump scans \textit{forwardly}, next block to be read is \textit{uniquely defined}
Concurrent $t$-jump method

**Definition (Cost-$t$ testing)**

Determine whether there is a size-$k$ partition $P$ with $c(P) \leq t$

1. if such $P$ exists, output **Yes**
2. otherwise, output **No**

Cost-$t$ Testing vs. Cost-$t$ Splitters Problem

▶ avoid storing the feasible splitters ($O(k/B)$ space)
▶ lead to the concurrent extension of cost-$t$ testing

**Intuition of concurrent $t$-jump**

$t$-jump scans *forwardly*, next block to be read is *uniquely defined*

- one execution requires $O(1)$ space

- Intervals and Stabbing-Count Array on Disk

```
<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Concurrent \( t \)-jump method

**Definition (Cost- \( t \) testing)**

Determine whether there is a size-\( k \) partition \( P \) with \( c(P) \leq t \)

1. If such \( P \) exists, output **Yes**
2. Otherwise, output **No**

**Cost- \( t \) Testing vs. Cost- \( t \) Splitters Problem**

▶ avoid storing the feasible splitters (\( O(k/B) \) space)
▶ lead to the concurrent extension of cost- \( t \) testing

**Intuition of concurrent \( t \)-jump**

\( t \)-jump scans **forwardly**, next block to be read is uniquely defined

- one execution requires \( O(1) \) space

---

System cache

\( \ell_i \)

\( \ell_{i+1} \)

Read-ahead buffer

block 1

block 2

block 3

Intervals and Stabbing-Count Array on Disk

- \( t \)-jump scans **forwardly**, next block to be read is *uniquely defined*
- one execution requires \( O(1) \) space
Concurrent $t$-jump method

- Initialize $h$ threads of cost-$t_i$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- At any time activate the thread with $\min(f(t_i))$

Intervals and Stabbing-Count Array

Permissible Range
Concurrent $t$-jump method

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- at any time activate the thread with $\min(f(t_i))$
Concurrent $t$-jump method

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- Initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- At any time activate the thread with $\min(f(t_i))$
Concurrent \( t \)-jump method

- Initialize \( h \) threads of cost-\( t \) testings, \( 1 \leq t_1 < t_2 < \ldots < t_h \leq N \)
- \( f(t_i) \) the frontier of cost-\( t_i \) testing
- At any time activate the thread with \( \min(f(t_i)) \)

Intervals and Stabbing-Count Array, \( h = 3 \) concurrent testings

Permissible Range

\[
\begin{array}{c|c|c|c}
& f(t_2) & f(t_3) & f(t_1) \\
\hline
\end{array}
\]
Concurrent \( t \)-jump method

- Initialize \( h \) threads of cost-\( t \) testings, \( 1 \leq t_1 < t_2 < \ldots < t_h \leq N \)
- \( f(t_i) \) the frontier of cost-\( t_i \) testing
- At any time activate the thread with \( \min(f(t_i)) \)

Intervals and Stabbing-Count Array, \( h = 3 \) concurrent testings

Permissible Range
Concurrent $t$-jump method

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- at any time activate the thread with $\min(f(t_i))$
Concurrent $t$-jump method

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- at any time activate the thread with $\min(f(t_i))$
**Concurrent $t$-jump method**

<table>
<thead>
<tr>
<th>$f(t_2)$</th>
<th>$f(t_1)$</th>
<th>$f(t_3)$</th>
</tr>
</thead>
</table>

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- at any time activate the thread with $\min(f(t_i))$

![Permissible Range Diagram](image)

**Permissible Range**

1 $t_1$ $t_2$ $\ldots$ $t_h$ $N$
Concurrent $t$-jump method

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- Initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- At any time activate the thread with $\min(f(t_i))$
Concurrent $t$-jump method

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- at any time activate the thread with $\min(f(t_i))$
Concurrent \( t \)-jump method

- \( f(t_i) \) the frontier of cost-\( t_i \) testing
- initialize \( h \) threads of cost-\( t \) testings, \( 1 \leq t_1 < t_2 < \ldots < t_h \leq N \)
- \( f(t_i) \) the frontier of cost-\( t_i \) testing
- at any time activate the thread with \( \min(f(t_i)) \)

Intervals and Stabbing-Count Array, \( h = 3 \) concurrent testings
Concurrent $t$-jump method

> initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
> $f(t_i)$ the frontier of cost-$t_i$ testing
> at any time activate the thread with $\min(f(t_i))$

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings
Concurrent \( t \)-jump method

- \( \times \) cost-\( t_1 \) infeasible
- \( \checkmark \) cost-\( t_2 \) feasible
- \( \checkmark \) cost-\( t_3 \) feasible

Intervals and Stabbing-Count Array, \( h = 3 \) concurrent testings

- initialize \( h \) threads of cost-\( t \) testings, \( 1 \leq t_1 < t_2 < \ldots < t_h \leq N \)
- \( f(t_i) \) the frontier of cost-\( t_i \) testing
- at any time activate the thread with \( \min(f(t_i)) \)

Permissible Range

\[
\begin{array}{cccccc}
& & & & f(t_2) & \quad f(t_3) & \quad f(t_1) \\
\hline
&&&&&&
\end{array}
\]
Concurrent \( t \)-jump method

\[
\begin{array}{c}
\times \text{ cost-} t_1 \text{ infeasible } & \checkmark \text{ cost-} t_2 \text{ feasible } & \checkmark \text{ cost-} t_3 \text{ feasible }
\end{array}
\]

Intervals and Stabbing-Count Array, \( h = 3 \) concurrent testings

- initialize \( h \) threads of cost-\( t \) testings, \( 1 \leq t_1 < t_2 < \ldots < t_h \leq N \)
- \( f(t_i) \) the frontier of cost-\( t_i \) testing
- at any time activate the thread with \( \min(f(t_i)) \)

Permissible Range
Concurrent $t$-jump method

- $\times$ cost-$t_1$ infeasible  ✔ cost-$t_2$ feasible  ✔ cost-$t_3$ feasible

Intervals and Stabbing-Count Array, $h = 3$ concurrent testings

- initialize $h$ threads of cost-$t$ testings, $1 \leq t_1 < t_2 < \ldots < t_h \leq N$
- $f(t_i)$ the frontier of cost-$t_i$ testing
- at any time activate the thread with $\min(f(t_i))$
- when $t^*$ is found, one more scan to locate the splitters
Outline

1 Motivation and Problem Formulation

2 A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis

3 Internal Memory Method
   - Cost-$t$ Splitter Problem
   - Stabbing-count Array and $t$-jump method
   - Cost Analysis

4 External Memory Method
   - Concurrent $t$-jump method
   - Cost Analysis

5 Experiments

6 Conclusion
Cost Analysis

- Construct the stabbing-count array: $O(SORT(N))$ I/Os
- Construct the stabbing-count array: $O(SORT(N))$ I/Os
- One round of Concurrent Cost-$t$ testings: $O(N/B)$ I/Os at most
Cost Analysis

- Construct the stabbing-count array: \( O(SORT(N)) \) I/Os
- One round of Concurrent Cost-\( t \) testings: \( O(N/B) \) I/Os at most
- \# rounds of Concurrent Cost-\( t \) testings: \( O(\log_M N) \leq O(\log_{M/B} N/B) \)
Construct the stabbing-count array: $O(SORT(N))$ I/Os

One round of Concurrent Cost-$t$ testings: $O(N/B)$ I/Os at most

# rounds of Concurrent Cost-$t$ testings: $O(\log_M N) \leq O(\log_{M/B} N/B)$

Cost to find $t^*$: $SORT(N)$ at most
- Construct the stabbing-count array: $O(SORT(N))$ I/Os
- One round of Concurrent Cost-\(t\) testings: $O(N/B)$ I/Os at most
- \# rounds of Concurrent Cost-\(t\) testings: $O(\log_M N) \leq O(\log_{M/B} N/B)$
- Cost to find $t^*$: $SORT(N)$ at most
- Retrieve the optimal splitters: $O(\min(k, N/B))$ I/Os
Construct the stabbing-count array: $O(SORT(N))$ I/Os

One round of Concurrent Cost-$t$ testings: $O(N/B)$ I/Os at most

# rounds of Concurrent Cost-$t$ testings: $O(\log_M N) \leq O(\log_{M/B} N/B)$

Cost to find $t^*$: $SORT(N)$ at most

Retrieve the optimal splitters: $O(\min(k, N/B))$ I/Os

Concurrent $t$-jump method is as efficient as external sorting!
Outline

1 Motivation and Problem Formulation
2 A Baseline Method
   - Strategy to Place Splitters
   - Dynamic Programming Approach
   - Cost Analysis
3 Internal Memory Method
   - Cost-\(t\) Splitter Problem
   - Stabbing-count Array and \(t\)-jump method
   - Cost Analysis
4 External Memory Method
   - Concurrent \(t\)-jump method
   - Cost Analysis
5 Experiments
6 Conclusion
Experiments: Setup

- Internal: DP, $t$-jump

Implementation in C++

I/O efficient methods are implemented with TPIE

Experiments on a Linux machine with 4GB of Mem

Two large real datasets:

- Temp is a temperature dataset from the MesoWest contains measurements from Jan 1997 to Oct 2011
- Meme is obtained from the Memetracker Project tracks the frequency of popular quotes over time

Internal External

Dataset a subset of Meme a subset of Temp

Size $\sim 21$ MB $\sim 5$ GB

$N \sim 1$ million $\sim 200$ million

$k$ 40 5000
h not applicable 5

Wangchao Le Feifei Li Yufei Tao Robert Christensen

Optimal Splitters for Temporal and Multi-version Databases
Experiments: Setup

- Internal: DP, $t$-jump
- External: $t$-jump, $ct$-jump, $sc$-tree (use Segment $B$-tree)
Experiments: Setup

- Internal: DP, $t$-jump
- External: $t$-jump, $ct$-jump, $sc$-tree (use Segment B-tree)
- Implementation in C++
  - I/O efficient methods are implemented with TPIE
Experiments: Setup

- Internal: DP, $t$-jump
- External: $t$-jump, $ct$-jump, $sc$-tree (use *Segment B-tree*)
- Implementation in C++
  - I/O efficient methods are implemented with TPIE
- Experiments on a Linux machine with 4GB of Mem

Two large real datasets:
- Temp is a temperature dataset from the MesoWest contains measurements from Jan 1997 to Oct 2011
- Meme is obtained from the Memetracker Project tracks the frequency of popular quotes over time

Internal External
Dataset
Size
$\sim 21 \text{ MB}$
$\sim 5 \text{ GB}$
$N$ not applicable
$\sim 1 \text{ million}$
$\sim 200 \text{ million}$
$k$
$\sim 40 \sim 5000$
$h$
$\not\text{applicable}$
$5$
Experiments: Setup

- **Internal**: DP, \( t \)-jump
- **External**: \( t \)-jump, \( ct \)-jump, **sc-tree** (use *Segment B-tree*)
- **Implementation in C++**
  - I/O efficient methods are implemented with TPIE
- **Experiments on a Linux machine with 4GB of Mem**
- **Two large real datasets**:
  - **Temp** is a temperature dataset from the *MesoWest*
    - contains measurements from Jan 1997 to Oct 2011
  - **Meme** is obtained from the *Memetracker* Project
    - tracks the frequency of popular quotes over time
Experiments: Setup

- Internal: DP, $t$-jump
- External: $t$-jump, $ct$-jump, $sc$-tree (use $Segment$ $B$-tree)
- Implementation in C++
  - I/O efficient methods are implemented with TPIE
- Experiments on a Linux machine with 4GB of Mem
- Two large real datasets:
  - $Temp$ is a temperature dataset from the $MesoWest$
    - contains measurements from Jan 1997 to Oct 2011
  - $Meme$ is obtained from the $Memetracker$ Project
    - tracks the frequency of popular quotes over time

<table>
<thead>
<tr>
<th></th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>a subset of $Meme$</td>
<td>a subset of $Temp$</td>
</tr>
<tr>
<td>Size</td>
<td>$\sim 21$ MB</td>
<td>$\sim 5$ GB</td>
</tr>
<tr>
<td>$N$</td>
<td>$\sim 1$ million</td>
<td>$\sim 200$ million</td>
</tr>
<tr>
<td>$k$</td>
<td>40</td>
<td>5000</td>
</tr>
<tr>
<td>$h$</td>
<td>not applicable</td>
<td>5</td>
</tr>
</tbody>
</table>
Experiments: Vary k Internal Memory Methods

Time (second)

<table>
<thead>
<tr>
<th>k</th>
<th>DP</th>
<th>t-jump</th>
<th>sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experiments: Vary $h$ External Memory Methods

Time (seconds)

$k=2000$  $k=5000$  $k=10000$

$h$

Wangchao Le  Feifei Li  Yufei Tao  Robert Christensen
Optimal Splitters for Temporal and Multi-version Databases
Experiments: Vary $k$ External Memory Methods

Left: Number of I/Os ($\times 10^6$) vs. $k$
- ct-jump
- t-jump
- sc-tree
- sort

Right: Time (seconds) vs. $k$
- ct-jump
- t-jump
- sc-tree
- sort
Motivation and Problem Formulation

A Baseline Method
- Strategy to Place Splitters
- Dynamic Programming Approach
- Cost Analysis

Internal Memory Method
- Cost-\(t\) Splitter Problem
- Stabbing-count Array and \(t\)-jump method
- Cost Analysis

External Memory Method
- Concurrent \(t\)-jump method
- Cost Analysis

Experiments

Conclusion
We studied the optimal splitters problem for large interval data, which is essential in a distributed and parallel setting.

Future work includes extending our studies to higher dimensions.
We studied the optimal splitters problem for large interval data, which is essential in a distributed and parallel setting. Our best solutions \( t \)-jump and \( ct \)-jump are more efficient than the baseline solutions. Both are as efficient as sorting algorithms.

Future work includes extending our studies to higher dimensions.
We studied the optimal splitters problem for large interval data, which is essential in a distributed and parallel setting.

Our best solutions $t$-jump and $ct$-jump are more efficient than the baseline solutions.

- both are as efficient as sorting algorithms

Future work includes extending our studies to higher dimensions.
Thank You

Q and A
Strategy to Place Splitters

- Where to place splitters?
Where to place splitters?

- let \( \mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\} \), and let \( S = \{s_1...s_N\} \) in ascending order.
Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, and let $\mathbf{S} = \{s_1...s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$
Strategy to Place Splitters

- Where to place splitters?
  - let $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, and let $\mathbf{S} = \{s_1...s_N\}$ in ascending order.
  - for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$
Where to place splitters?

- let \( I = \{[s_1, e_1], \ldots, [s_N, e_N]\} \), and let \( S = \{s_1, \ldots, s_N\} \) in ascending order.
- for any splitter \( \ell \), let \( \ell(1) \) be the smallest starting value s.t. \( \ell(1) \geq \ell \)

**Lemma**

*For any partition \( P \) with distinct splitters \( \ell_1 < \ldots < \ell_m \) and \( \ell_m(1) \) is undefined. Let \( P' \) be a partition with splitters \( \ell_1, \ldots, \ell_{m-1} \), then \( c(P') = c(P) \).*
Strategy to Place Splitters

Where to place splitters?
- Let \( \mathcal{I} = \{[s_1, e_1], ..., [s_N, e_N]\} \), and let \( S = \{s_1, ..., s_N\} \) in ascending order.
- For any splitter \( \ell \), let \( \ell(1) \) be the smallest starting value s.t. \( \ell(1) \geq \ell \)

Lemma

For any partition \( P \) with distinct splitters \( \ell_1 < ... < \ell_m \) and \( \ell_m(1) \) is undefined. Let \( P' \) be a partition with splitters \( \ell_1, ..., \ell_{m-1} \), then \( c(P') = c(P) \).

\[
\begin{align*}
\ell_{m-1} & \quad \ell_m & \quad \ell_{m-1} \\
\bigcirc & \quad \square & \quad \bigcirc \\
b_m & \quad b_{m+1} & \quad b'_m \\
\end{align*}
\]

- \( b_{m+1} \subseteq b_m = b'_m \)
- \( c(P') = c(P) \geq |b_m| = |b'_m| \geq |b_{m+1}| \)
Where to place splitters?

- Let $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, and let $\mathbf{S} = \{s_1...s_N\}$ in ascending order.
- For any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$

**Lemma**

*For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is undefined. Let $P'$ be a partition with splitters $\ell_1, ..., \ell_{m-1}$, then $c(P') = c(P)$.*

**Lemma**

*For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is defined. Let $\ell_i$ be the largest splitter that does not in $\mathbf{S}$. Define $P'$ from $P$ by (i) deleting $\ell_i$, if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.*
Strategy to Place Splitters

- Where to place splitters?
  - let $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, and let $\mathbf{S} = \{s_1...s_N\}$ in ascending order.
  - for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$

**Lemma**

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is undefined. Let $P'$ be a partition with splitters $\ell_1,...,\ell_{m-1}$, then $c(P') = c(P)$.

**Lemma**

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is defined. Let $\ell_i$ be the largest splitter that does not in $\mathbf{S}$. Define $P'$ from $P$ by (i) deleting $\ell_i$, if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.

\[ c(P) = 5 \]
Where to place splitters?

- let $I = \{[s_1, e_1]...[s_N, e_N]\}$, and let $S = \{s_1...s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the **smallest starting value** s.t. $\ell(1) \geq \ell$

**Lemma**

*For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is undefined. Let $P'$ be a partition with splitters $\ell_1, ..., \ell_{m-1}$, then $c(P') = c(P)$.*

**Lemma**

*For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is defined. Let $\ell_i$ be the largest splitter that does not in $S$. Define $P'$ from $P$ by (i) deleting $\ell_i$, if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.*

\[c(P) = 5\]
Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, and let $\mathbf{S} = \{s_1...s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$

Lemma

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is undefined.
Let $P'$ be a partition with splitters $\ell_1, ..., \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is defined.
Let $\ell_i$ be the largest splitter that does not in $\mathbf{S}$. Define $P'$ from $P$ by (i) deleting $\ell_i$, if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.
Strategy to Place Splitters

Where to place splitters?

- let $\mathcal{I} = \{[s_1, e_1], \ldots, [s_N, e_N]\}$, and let $\mathbf{S} = \{s_1, \ldots, s_N\}$ in ascending order.
- for any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$

Lemma

For any partition $P$ with distinct splitters $\ell_1 < \ldots < \ell_m$ and $\ell_m(1)$ is undefined. Let $P'$ be a partition with splitters $\ell_1, \ldots, \ell_{m-1}$, then $c(P') = c(P)$.

Lemma

For any partition $P$ with distinct splitters $\ell_1 < \ldots < \ell_m$ and $\ell_m(1)$ is defined. Let $\ell_i$ be the largest splitter that does not in $\mathbf{S}$. Define $P'$ from $P$ by (i) deleting $\ell_i$, if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.

Wangchao Le Feifei Li Yufei Tao Robert Christensen

Optimal Splitters for Temporal and Multi-version Databases
Where to place splitters?

- Let $I = \{[s_1, e_1], ..., [s_N, e_N]\}$, and let $S = \{s_1, ..., s_N\}$ in ascending order.
- For any splitter $\ell$, let $\ell(1)$ be the smallest starting value s.t. $\ell(1) \geq \ell$

**Lemma**

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is undefined. Let $P'$ be a partition with splitters $\ell_1, ..., \ell_{m-1}$, then $c(P') = c(P)$.

**Lemma**

For any partition $P$ with distinct splitters $\ell_1 < ... < \ell_m$ and $\ell_m(1)$ is defined. Let $\ell_i$ be the largest splitter that does not in $S$. Define $P'$ from $P$ by (i) deleting $\ell_i$, if $\ell_i(1) = \ell_{i+1}$, otherwise (ii) replacing $\ell_i$ with $\ell_i(1)$. Then, $c(P') \leq c(P)$.

Should always try to split on $S$!