Optimal Splitters for Temporal and Multi-version Databases

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

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

June 30, 2013
Temporal and multi-version data are important in:
- financial market
- scientific application
- data warehousing

![Graph showing temporal data over time](image)

Score

Temporal data

Time

Optimal Splitters for Temporal and Multi-version Databases
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
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

- 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
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 $P(I,k)$:

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: $c(P) = \max \{|b_1|, \ldots, |b_{k+1}|\}$
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, $k = 2$

Cost of a partition: $c(P) = \max \{|b_1|, \ldots, |b_{k+1}|\}$
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 $P(\mathcal{I}, k)$:
- has $k$ distinct vertical splitters and $k + 1$ buckets

1. 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 $P(\mathcal{I}, k)$:

- 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: $c(P) = \max\{|b_1|,...,|b_{k+1}|\}$
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

\[
\begin{align*}
C(P) &= \max\{|b_1| = 3, |b_2| = 4, |b_3| = 5\} \\
&= 5
\end{align*}
\]

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

Cost of a partition: \(c(P) = \max\{|b_1|\ldots|b_{k+1}|\}\)
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 \( P^*(I, k) \) with the smallest cost, i.e.

\[
P^*(I, k) = \arg\min_c(c(P))
\]

An example, \( k = 2 \)

![Diagram showing object time with three objects and time intervals](image-url)
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.

\[
P^*(I, k) = \arg\min (c(P))
\]

An example, \( k = 2 \)

Optimal Splitters, \( c(P) = 4 \)
Problem Formulation

- **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.

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

- In this talk, our objective:

  Find \( P^* \) and \( c(P^*) \) for \( \mathcal{I} \) and a fixed budget \( k \)
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
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?

\[
I = \{s_1, e_1\} \ldots \{s_N, e_N\}, \quad \text{and let} \quad S = \{s_1 \ldots s_N\} \text{in ascending order.}
\]
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.
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. \( \ell(1) \geq \ell \)

\[
\begin{array}{c}
\ell \\
\end{array}
\]
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. $\ell(1) \geq \ell$.
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. $\ell(1) \geq \ell$.

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, $c(P') \leq c(P)$. 

$c(P) = 5$
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. $\ell(1) \geq \ell$

![Diagram of splitters and their placements]

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, $c(P') \leq c(P)$.

$c(P) = 5$

![Graphical representation of partition and observation]

Wangchao Le  Feifei Li  Yufei Tao  Robert Christensen
Optimal Splitters for Temporal and Multi-version Databases
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, $c(P') \leq c(P)$.
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. $\ell(1) \geq \ell$.

\[ \ell \quad \ell(1) \]

- 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, $c(P') \leq c(P)$.

\[ \ell_1 \quad \ell_2(1) \]

- No effect on $b_2$
- Shrink $b_3$

\[ c(P') = 4 \leq c(P) = 5 \]
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. $\ell(1) \geq \ell$.

**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 $\mathcal{I} = \{[s_1, e_1]...[s_N, e_N]\}$, 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 $S$ !**
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
Given a splitter \( \ell \) and a set of intervals \( \mathcal{I} \) stored in an array.
Dynamic Programming Approach

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

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

$\mathcal{I}^+(\ell) = \{(s_i, e_i) \in I \mid s_i > \ell\}$
Dynamic Programming Approach

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

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

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

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

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

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

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

  $c(P^*(I, k))$

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

  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

\[
\begin{align*}
\mathcal{I}^- (\ell) &= \{ [s_i, e_i] \in I | s_i < \ell \} \\
\mathcal{I}^+ (\ell) &= \{ [s_i, e_i] \in I | s_i > \ell \} \\
\mathcal{I}^o (\ell) &= \{ [s_i, e_i] \in I | s_i = \ell \} \\
\mathcal{I}^x (\ell) &= \{ [s_i, e_i] \in I | s_i < \ell < e_i \}
\end{align*}
\]

- 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

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

  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 $\mathcal{I}$ stored in an array

\[
\begin{align*}
\mathcal{I}^- (\ell) &= \{ [s_i, e_i] \in I | s_i < \ell \} \\
\mathcal{I}^+ (\ell) &= \{ [s_i, e_i] \in I | s_i > \ell \} \\
\mathcal{I}^o (\ell) &= \{ [s_i, e_i] \in I | s_i = \ell \} \\
\mathcal{I}^x (\ell) &= \{ [s_i, e_i] \in I | s_i < \ell < e_i \}
\end{align*}
\]

- Dynamic programming

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

- 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 $\ell_k$? $\ell_k \in S(I)$
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 $\ell_k$? $\ell_k \in S(I)$

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

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

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

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

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

- 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

$$
\ell
$$

- Dynamic programming

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

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

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

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

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

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

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

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

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

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

  

  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

$$\ell$$

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

- Dynamic programming

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

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

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

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

- Dynamic programming

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

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

  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

  $\ell$

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

  **Dynamic programming**

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

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

  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

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

- Dynamic programming

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

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

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

A sub-problem: $c(P^*(\mathcal{I}^- (\ell_k), k - 1))$
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
A common sub-problem may appear more than one time
A common sub-problem may appear more than one time

- Memoization

| \( |S| = N, k \) splitters |
|--------------------------|
| \( [1, 1] \)          | \( \cdots \) | \( [1, k - 1] \) | \( [1, k] \) |
| \( [N, 1] \)          | \( \cdots \) | \( [N, k - 1] \) | \( [N, k] \) |
A common sub-problem may appear more than one time

- Memoization

Cost of the DP approach

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

- Memoization

\[ \begin{array}{c|ccccc}
 & \cdots & [1,k-1] & [1,k] \\
\hline
[1,1] & \cdots & \cdots & \cdots \\
[N,1] & \cdots & \cdots & \cdots \\
\hline
\end{array} \]

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

- Cost of the DP approach

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

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

*Memoization*

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

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

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 \(\text{LastBucket} (|\mathcal{I}\circ(\ell) + \mathcal{I}\times(\ell) + \mathcal{I}\plus(\ell)|)\)
A common sub-problem may appear more than one time

- Memoization

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

|\(S| = N, k \) splitters

Cost of the DP approach

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

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
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 \(c(P) \leq t\)
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. $\exists P \in \mathcal{P}(l, k), c(P) = \bar{t}$
- 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 \(c(P) \leq t\)

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

**Lemma**

*If \(t\) is infeasible, then any \(t' < t\) is also infeasible*
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:
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 \( c(P) \leq t \)

\[ \begin{align*}
\text{if such } P \text{ exists, } t \text{ is feasible} \\
& \quad \text{Output: } \bar{t} \in [1, t] \text{ s.t. } \exists P \in \mathcal{P}(l, k), c(P) = \bar{t} \\
\text{otherwise, } t \text{ is infeasible} \\
& \quad \text{Output: } \bar{t} = 0
\end{align*} \]

**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 \( R = [1, N] \)
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
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
4. Report \(t^*\), when \(t^*\) is feasible but \(t^* - 1\) is infeasible
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
4. Report \(t^*\), when \(t^*\) is feasible but \(t^* - 1\) is infeasible
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
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
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 $\text{interset}$, $\text{tie}$
  - $\text{interset}[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
  - $\text{tie}[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$
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 $\text{interset}, \text{tie}$
    - $\text{interset}[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
    - $\text{tie}[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$

\[
\text{intersect}[3] = 2,
\]
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 `interset`, `tie`
    - $\triangleright$ `interset[i] = |\mathcal{I}^x(s_i)|$, # intervals intersecting $s_i$
    - $\triangleright$ `tie[i] = |\mathcal{I}^o(s_i)|$, # intervals in $\mathcal{I}^o(s_i)$ with ids less than $i$

```
```

![Diagram of stabbing count array](https://via.placeholder.com/150)
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 $\text{interset}, \text{tie}$
  $\triangleright \text{interset}[i] = |\mathcal{I}^\times(s_i)|$, # intervals intersecting $s_i$
  $\triangleright \text{tie}[i] = |\mathcal{I}^\circ(s_i)|$, # intervals in $\mathcal{I}^\circ(s_i)$ with ids less than $i$

intersect[4] = 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 $\text{interset}, \text{tie}$
    - $\text{interset}[i] = |\mathcal{I}^\times(s_i)|$, # intervals intersecting $s_i$
    - $\text{tie}[i] = |\mathcal{I}^\circ(s_i)|$, # intervals in $\mathcal{I}^\circ(s_i)$ with ids less than $i$

$\text{interset}[4] = 2$, $\text{tie}[4] = 1$

![Diagram showing intervals $s_1$ to $s_4$ and their corresponding end points $e_1$ to $e_4$.]
Stabbing-count Array

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

- For $s_4 \in \mathcal{I}$, $\text{intersect}[4] = 2, \text{tie}[4] = 1$

```
s_1 e_1
ds_2 e_2
---
s_3 e_3
ds_4 e_4
```
Stabbing-count Array

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

```
```

Lemma

\textit{The stabbing-count array can be built in } O(N \log N) \textit{ 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
- if not achievable, move $\ell_i + 1$ backward just enough to form the new $b_i$
*t*-jump method

1. *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**

4. 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**

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 $t$
The $t$-jump method solves an instance of the Cost-$t$ splitters problem. If feasible, it outputs the feasible $P$ and $c(P)$. It uses 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 $b_i$. 

---

The diagram shows the placement of splitters $s_1, s_2, s_3, s_4, s_5, s_6, s_7$ in ascending order.
- **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

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

```
intersect[4] = 1, jump at most 2 ids
```

```
|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

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

\[
|b_1| = 3 \quad \ell_1 \quad |b_2| = 3 \quad \ell_2
\]

\[
s_1 \quad s_2 \quad s_3 \quad s_4 \quad s_5 \quad s_6 \quad s_7
\]

$k = 2, t = 3$

**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 $b_i$
\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

\begin{align*}
|b_1| &= 3 & \ell_1 & |b_2| = 3 & \ell_2 & |b_3| = 3 \\
\ell_1 & & \ell_2 & & t = 3 \text{ is feasible} \\

k &= 2, \ t = 3
\end{align*}

\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**

- **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

\[
|b_1| = 3 \quad \ell_1 \quad |b_2| = 3 \quad \ell_2 \quad |b_3| = 3
\]

\[
s_1 \quad s_3 \quad s_2 \quad s_4 \quad s_5 \quad s_6 \quad s_7\]

\[
k = 2, \quad t = 3
\]

**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

**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

\[ |b_1| = 2 \quad \ell_1 \quad s_1 \quad s_2 \quad s_3 \quad s_4 \quad s_5 \quad s_6 \quad s_7 \quad k = 2, \quad t = 2 \]

**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
```

```
| \( b_1 \) | = 2
---|---
\( s_1 \) | \( s_2 \) | \( s_3 \) | \( \ell_1 \) | \( s_4 \) | \( s_5 \) | \( s_6 \) | \( s_7 \)
```

**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

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

  $|b_1| = 2$

  $k = 2, t = 2$

**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

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

  \[|b_1| = 2, |b_2| = 1, k = 2, t = 2\]

**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 \textbf{feasible}, output the feasible \(P\) and \(c(P)\)
  3. a greedy algorithm

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

\[
|b_1| = 2 \quad |b_2| = 1 \quad |b_3| = 5
\]

\[
l_1 \quad l_2 \quad l_3
\]

\[
s_1 \quad s_2 \quad s_3 \quad s_4 \quad s_5 \quad s_6 \quad s_7
\]

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

**Intuition**

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

\begin{align*}
    \text{jump } t = 2 \text{ ids, move back tie}[5] = 1
\end{align*}

\begin{align*}
    |b_1| = 2 & \quad |b_2| = 1 & \quad |b_3| = 5
\end{align*}

\begin{align*}
    k = 2, t = 2
    \quad t = 2 \text{ is infeasible}
\end{align*}

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 tie[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. \( \log N \) 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.

Wangchao Le Feifei Li Yufei Tao Robert Christensen

Optimal Splitters for Temporal and Multi-version Databases
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 ($k \ll N$)
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
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.*
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.
External Memory Method

- $\mathcal{I}$ 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

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

**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
\( I \) 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.
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
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**
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 (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}

**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**

\(\ell_i\)

Block 1, Block 2, Block 3

Intervals and Stabbing-Count Array on Disk
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**

Intervals and Stabbing-Count Array on Disk

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

- block 1
- block 2
- block 3

Read-ahead buffer
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 \( \min(f(t_i)) \)

Permissible Range

Intervals and Stabbing-Count Array
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

- \( f(t_1) \) 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 \( \min(f(t_i)) \)

Permissible Range

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

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

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
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

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

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

- 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

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

- cost-$t_1$ infeasible  ✓ cost-$t_2$ feasible ✓ cost-$t_3$ feasible

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>$f(t_2)$</th>
<th>$f(t_3)$</th>
<th>$f(t_1)$</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 $\min(f(t_i))$
Concurrent \( t \)-jump method

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

\[
\begin{array}{cccc}
\text{feasible} & f(t_2) & f(t_3) & f(t_1) \\
\text{infeasible} & & & \\
\end{array}
\]

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 \( \min(f(t_i)) \)
- 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
Construct the stabbing-count array: $O(\text{SORT}(N))$ 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
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 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: \( O(\log MN) \leq O(\log_{M/B} N/B) \)
- 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
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: \( O(\log_M N) \leq O(\log_{M/B} N/B) \)
- 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
Experiments: Setup

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

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

Experiment on a Linux machine with 4GB of Mem

Two large real datasets:

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

- Internal Dataset: a subset of Meme
- External Dataset: a subset of Temp

- Size: $\sim 21$ MB, $\sim 5$ GB
- $N$: $\sim 1$ million, $\sim 200$ million
- $k$: 40, 50, 500

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, \textit{sc-tree} (use \textit{Segment B-tree})
- Implementation in C++
  - I/O efficient methods are implemented with TPIE
- Experiments on a Linux machine with 4GB of Mem
Experiments: Setup

- Internal: DP, \( t \)-jump
- External: \( t \)-jump, \( ct \)-jump, \textit{sc-tree} (use \textit{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:
  - \textit{Temp} is a temperature dataset from the \textit{MesoWest}
    - contains measurements from Jan 1997 to Oct 2011
  - \textit{Meme} is obtained from the \textit{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>Dataset</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>( \sim 21 \text{ MB} )</td>
<td>( \sim 5 \text{ GB} )</td>
</tr>
<tr>
<td>( N )</td>
<td>( \sim 1 \text{ million} )</td>
<td>( \sim 200 \text{ 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

[Graph showing time (second) vs. $k$ for different methods: DP, t-jump, sort]
Experiments: Vary $h$ External Memory Methods

![Bar chart showing time (seconds) vs. $h$ for different $k$ values: k=2000, k=5000, k=10000.](chart.png)
Experiments: Vary $k$ External Memory Methods

![Graph 1](image1.png)

- Number of I/O ($\times 10^6$)
- k: 2000, 4000, 6000, 8000, 10000

![Graph 2](image2.png)

- Time (second)
- k: 2000, 4000, 6000, 8000, 10000

- ct-jump
- t-jump
- sc-tree
- sort

Wangchao Le  Feifei Li  Yufei Tao  Robert Christensen

Optimal Splitters for Temporal and Multi-version Databases
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
We studied the optimal splitters problem for large interval data, which is essential in a distributed and parallel setting.
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