Ranking Large Temporal Data

Jeffrey Jestes Jeff M. Phillips Feifei Li Mingwang Tang

THE UNIVERSITY OF UTAH®

[1]School of Computing
University of Utah

August 29, 2012

Introduction

- Temporal data is important in numerous domains:
  - financial market
  - scientific applications
  - biomedical field

Introduction

- Temporal data is important in numerous domains:
    - financial market
    - scientific applications
    - biomedical field
- Extensive efforts have been made towards efficiently storing, processing, and querying temporal data.

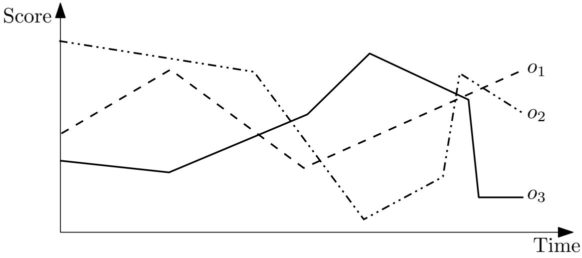
Introduction

- Temporal data is important in numerous domains:
    - financial market
    - scientific applications
    - biomedical field
- Extensive efforts have been made towards efficiently storing, processing, and querying temporal data.
- Ranking temporal data has only recently been studied. [LYL10]

[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Related Work

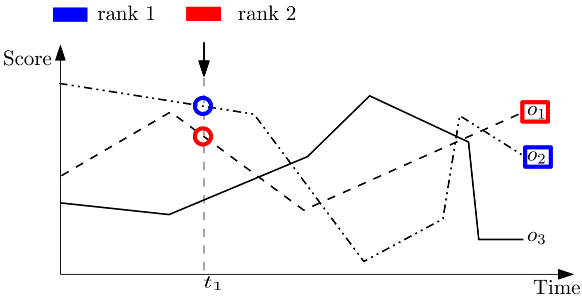
- The *instant* top-k query returns objects o_is with the k highest scores at query time t. [LYL10]



[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Related Work

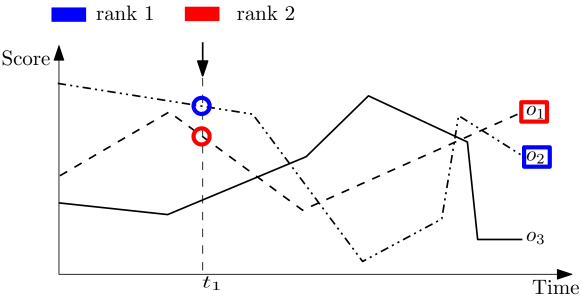
- The *instant* top-k query returns objects o_is with the k highest scores at query time t. [LYL10]



[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Related Work

- The *instant* top-k query returns objects o_is with the k highest scores at query time t. [LYL10]

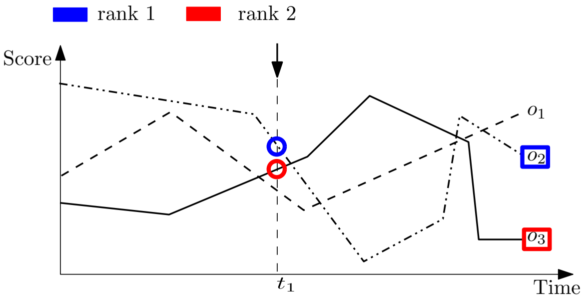


What is a good value for t?

[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Related Work

- The *instant* top-k query returns objects o_is with the k highest scores at query time t. [LYL10]

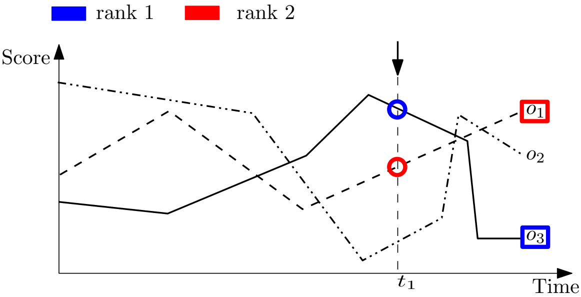


What is a good value for t?

[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Related Work

- The *instant* top-k query returns objects o_is with the k highest scores at query time t. [LYL10]

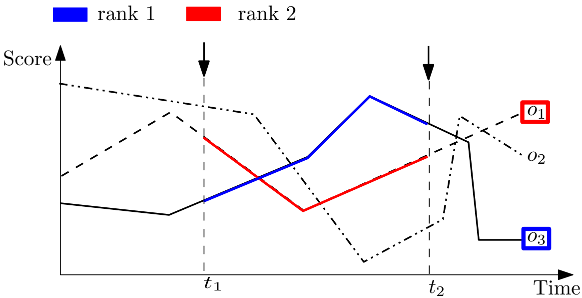


What is a good value for t?

[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Related Work

- The *instant* top-k query returns objects o_is with the k highest scores at query time t. [LYL10]

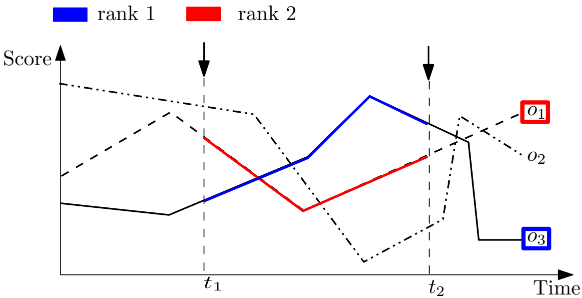


Use aggregation within a temporal interval instead!!!

[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Related Work

- The *instant* top-k query returns objects o_is with the k highest scores at query time t. [LYL10]

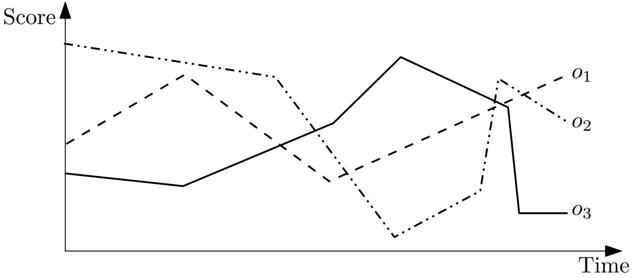


Use aggregation within a temporal interval instead!!!

- Example: Return top-10 weather stations with highest average temperature from 1 Aug to 27 Aug.

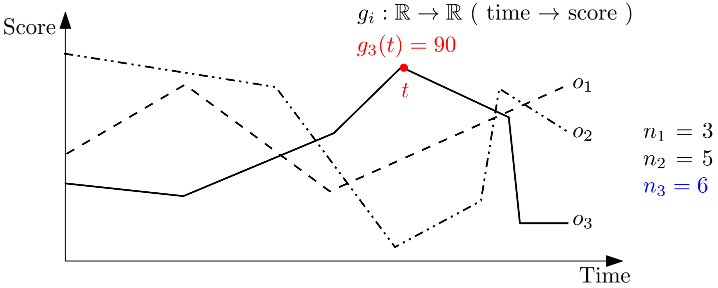
[LYL10] Li et al., Top-k queries on temporal data. In *VLDBJ*, 2010.

Problem Formulation



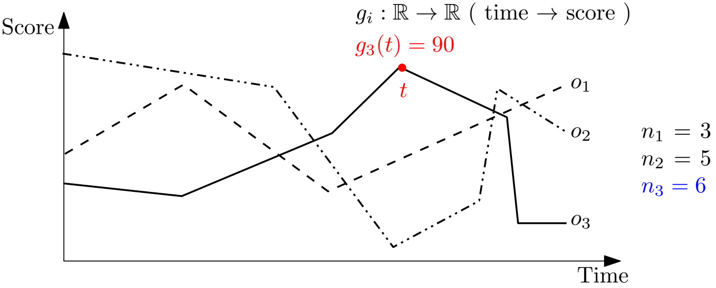
- Temporal database consists of m objects $o_1, o_2, \ldots, o_m$

Problem Formulation



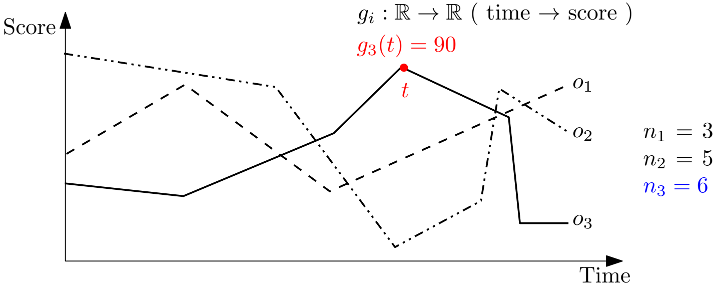
- Temporal database consists of m objects $o_1, o_2, \ldots, o_m$
- o_i is represented by piecewise linear function g_i with n_i segments.

Problem Formulation



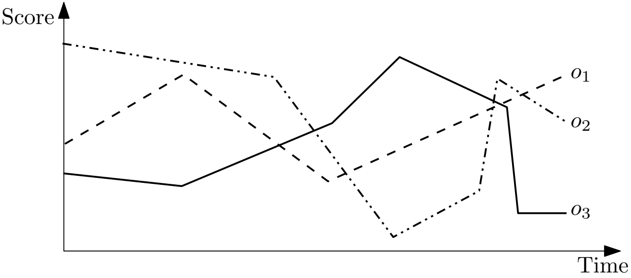
- Temporal database consists of m objects $o_1, o_2, \ldots, o_m$
- o_i is represented by piecewise linear function g_i with n_i segments.
- top-$k(t_1, t_2, \sigma)$ is an aggregate top-k query for aggregate function σ
  - $g_i(t_1, t_2)$ represent all possible values of g_i in $[t_1, t_2]$
  - $\sigma(g_i(t_1, t_2))$ $(= \sigma_i(t_1, t_2))$ is the aggregate score of o_i in $[t_1, t_2]$

Problem Formulation



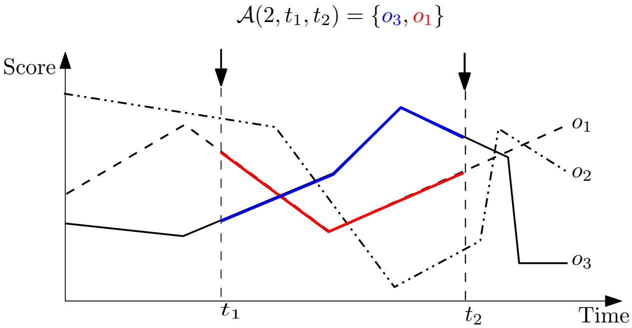
- Temporal database consists of m objects $o_1, o_2, \ldots, o_m$
- o_i is represented by piecewise linear function g_i with n_i segments.
- top-$k(t_1, t_2, \sigma)$ is an aggregate top-k query for aggregate function σ
  - $g_i(t_1, t_2)$ represent all possible values of g_i in $[t_1, t_2]$
  - $\sigma(g_i(t_1, t_2))$ $(= \sigma_i(t_1, t_2))$ is the aggregate score of o_i in $[t_1, t_2]$
  - For $\sigma =$ **sum**, $\sigma(g_i(t_1, t_2)) = \int_{t1}^{t2} g_i(t)dt$

Problem Formulation



- $\mathcal{A}(k, t_1, t_2)$: ordered top-k objects for top-$k(t_1, t_2, \sigma)$
- Let $\sigma = \textbf{sum} = \int_{t1}^{t2} g(t)dt$

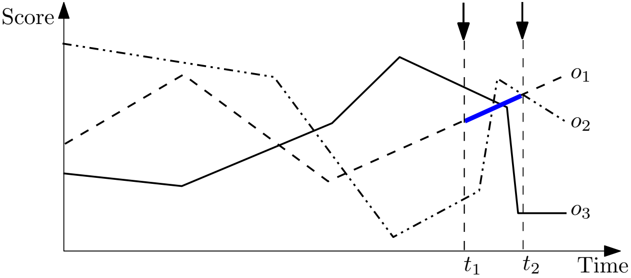
Problem Formulation



- $\mathcal{A}(k, t_1, t_2)$: ordered top-k objects for top-$k(t_1, t_2, \sigma)$
- Let $\sigma = \textbf{sum} = \int_{t1}^{t2} g(t)dt$

Problem Formulation

$\mathcal{A}(1, t_1, t_2) = \{o_1\}$



- $\mathcal{A}(k, t_1, t_2)$: ordered top-k objects for top-$k(t_1, t_2, \sigma)$
- Let $\sigma =$ **sum** $= \int_{t1}^{t2} g(t)dt$

Outline

Baseline Solution

- Compute $\sigma_i(t_1, t_2)$ for all objects by scanning each segment.

Baseline Solution

- Compute $\sigma_i(t_1, t_2)$ for all objects by scanning each segment.
- Simple improvement: use B-tree to avoid segments outside query interval.
- Query cost: $O(log_B N + \frac{\sum_{i=1}^{m} q_i}{B} + (m/B) log_B k)$
  - q_i = number of segments overlapping $[t_1, t_2]$
- We denote this query EXACT1.

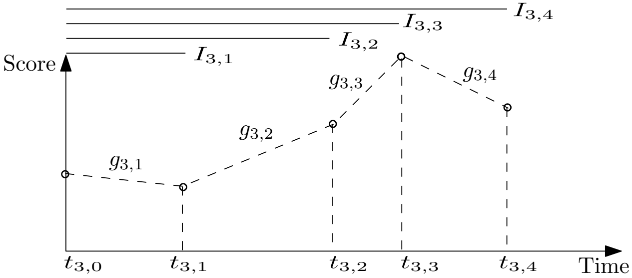
Improved Solution using Prefix Sums and B-tree Forest

- We can avoid scanning all overlapping segments with $[t_1, t_2]$ by using prefix sums:
  - Index segment and prefix sums for an object in a B-tree.
  - Compute $\sigma_i(t_1, t_2)$ by retrieving two segments from B-tree.

Improved Solution using Prefix Sums and B-tree Forest

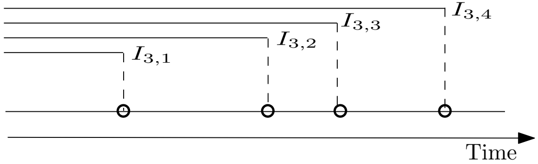
- We can avoid scanning all overlapping segments with $[t_1, t_2]$ by using prefix sums:
  - Index segment and prefix sums for an object in a B-tree.
  - Compute $\sigma_i(t_1, t_2)$ by retrieving two segments from B-tree.
- Query cost is $O(\sum_{i=1}^{m} log_B n_i + (m/B) log_B k)$
- This solution is denoted $\textsc{Exact2}$.

Improved Solution using Prefix Sums and Interval Tree



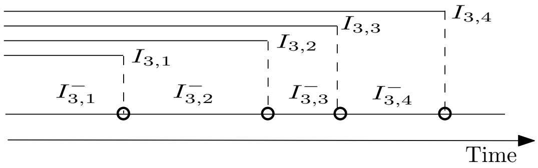
- Consider an object o_i with intervals $I_{i,1}, \ldots, I_{i,n_i}$
  - $g_{i,j} =$ jth segment of o_i is $((t_{i,j-1}, v_{i,j-1}), (t_{i,j}, v_{i,j}))$
  - $I_{i,\ell} = [t_{i,0}, t_{i,\ell}]$ for $\ell = 1, \ldots, n_i$

Improved Solution using Prefix Sums and Interval Tree



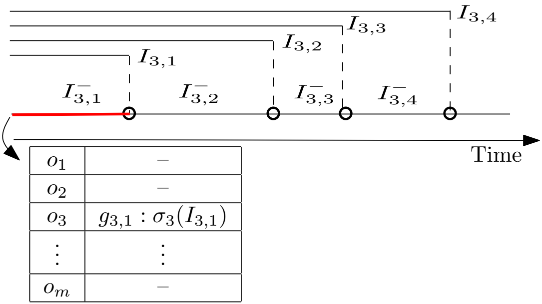
- We define $I^-_{i,1}, \ldots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



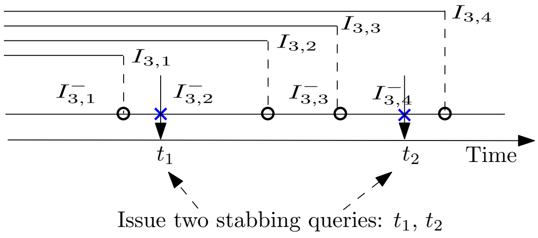
- We define $I^-_{i,1}, \ldots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



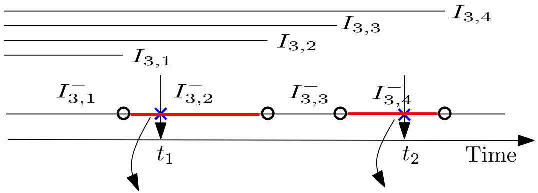
- We define $I_{i,1}^-, \ldots, I_{i,n_i}^-$ s.t. $I_{i,\ell}^- = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
    - key: $(I_{i,\ell}^-)$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



- We define $I^-_{i,1}, \ldots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



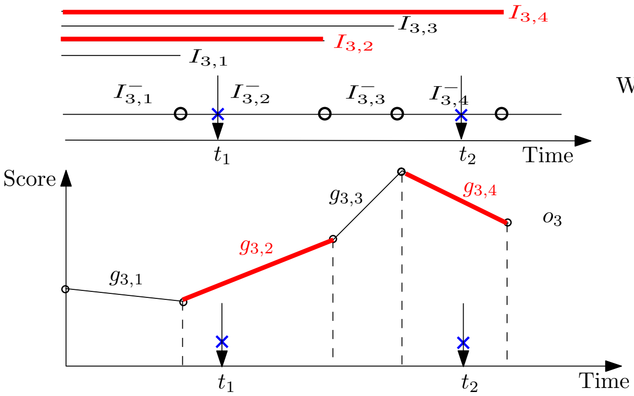
o_1	–
o_2	–
o_3	$g_{3,2} : \sigma_3(I_{3,2})$
$\vdots$	$\vdots$
o_m	–

o_1	–
o_2	–
o_3	$g_{3,4} : \sigma_3(I_{3,4})$
$\vdots$	$\vdots$
o_m	–

Retrieve associated $2m$ data entries

- We define $I^-_{i,1}, \ldots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

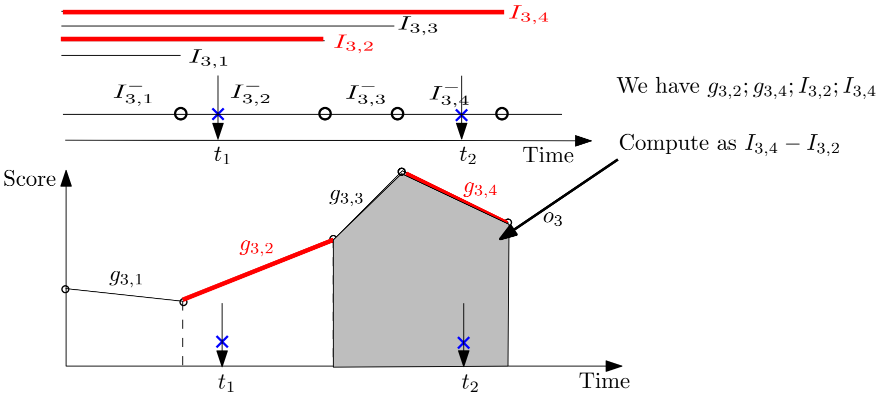
Improved Solution using Prefix Sums and Interval Tree



We have $g_{3,2}; g_{3,4}; I_{3,2}; I_{3,4}$

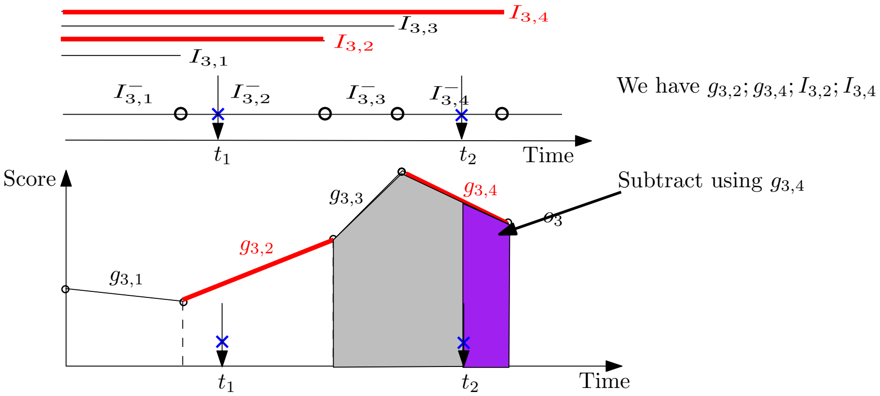
- We define $I^-_{i,1}, \dots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \dots, m$ and $\ell = 1, \dots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



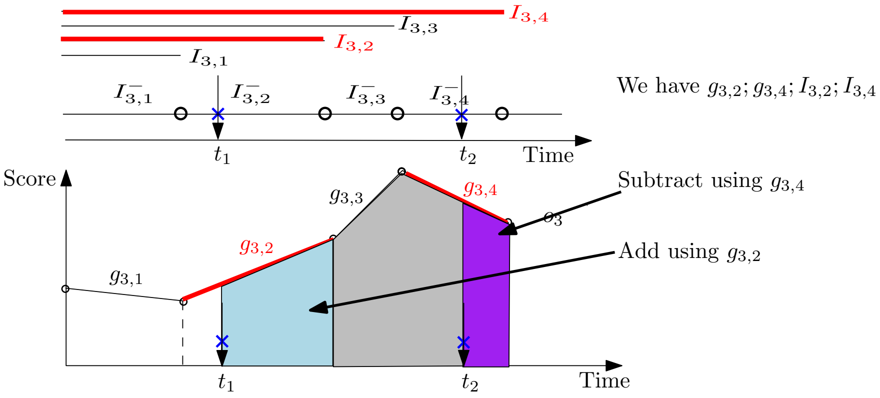
- We define $I^-_{i,1}, \ldots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



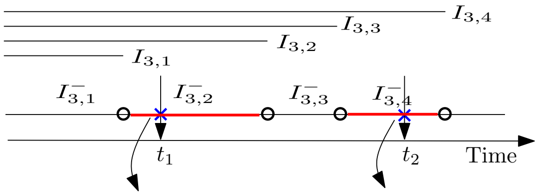
- We define $I^-_{i,1}, \ldots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



- We define $I^-_{i,1}, \ldots, I^-_{i,n_i}$ s.t. $I^-_{i,\ell} = [I_{i,\ell-1}, I_{i,\ell}]$
- The data entries for $i = 1, \ldots, m$ and $\ell = 1, \ldots, n_i$ are
  - key: $(I^-_{i,\ell})$ and value: $(g_{i,\ell}, \sigma_i(I_{i,\ell}))$

Improved Solution using Prefix Sums and Interval Tree



o_1	–
o_2	–
o_3	$g_{3,2} : \sigma_3(I_{3,2})$
⋮	⋮
o_m	–

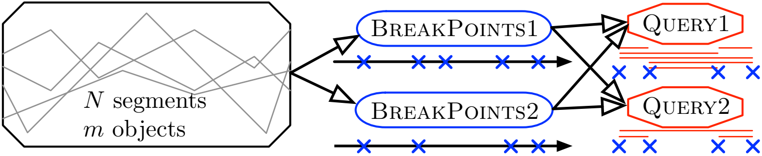
o_1	–
o_2	–
o_3	$g_{3,4} : \sigma_3(I_{3,4})$
⋮	⋮
o_m	–

Retrieve associated $2m$ data entries

- Total stabbing query cost is $O(log_B N + m/B)$.
  - Using priority queue to get top-k is $O(log_B N + (m/B) log_B k)$.
- We denote this query EXACT3.

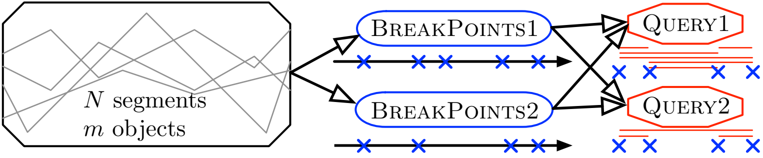
Outline

Approximate Solution Overview



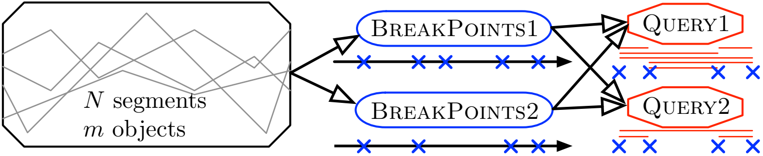
- Our most query-efficient technique costs $O(log_B N + m/B)$.
  - Must compute all m aggregates $\sigma_i(t_1, t_2)$.
  - Still too expensive for large datasets with large m.

Approximate Solution Overview



- Our most query-efficient technique costs $O(log_B N + m/B)$.
  - Must compute all m aggregates $\sigma_i(t_1, t_2)$.
  - Still too expensive for large datasets with large m.
- Our approximate methods construct breakpoints $\mathcal{B} = \{b_1, \ldots, b_r\}, b_i \in [0, T]$.

Approximate Solution Overview

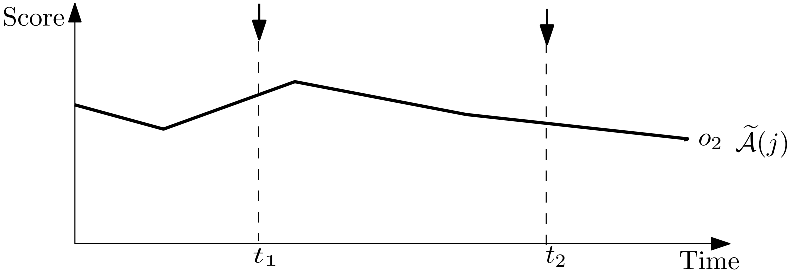


- Our most query-efficient technique costs $O(log_B N + m/B)$.
  - Must compute all m aggregates $\sigma_i(t_1, t_2)$.
  - Still too expensive for large datasets with large m.
- Our approximate methods construct breakpoints $\mathcal{B} = \{b_1, \ldots, b_r\}, b_i \in [0, T]$.
- Queries are snapped to align to breakpoints.
  - A query snapped to (b_i, b_j) uses $\sigma_i(b_i, b_j)$ as an object's score.

Approximate Solution Notations

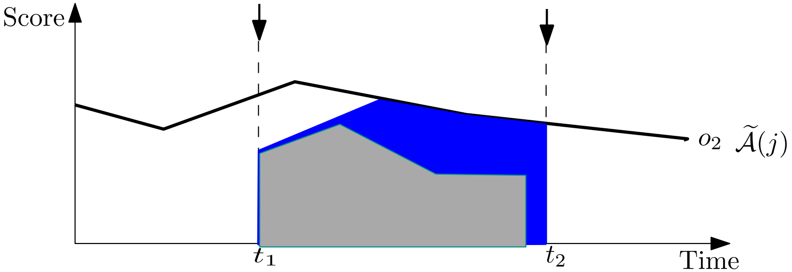
- G is an (ε, α)-approximation algorithm if:
  - G returns $\widetilde{\sigma}_i(t_1, t_2)$ s.t. $\sigma_i(t_1, t_2)/\alpha - \varepsilon M \leq \widetilde{\sigma}_i(t_1, t_2) \leq \sigma_i(t_1, t_2) + \varepsilon M$
  - $\alpha \geq 1$, $\varepsilon > 0$
  - $M = \sum_{i=1}^{m} \sigma_i(0, T)$
- Must hold for all objects and temporal intevals.

Approximate Solution Notations



- $\mathcal{A}(j)$ ($\widetilde{\mathcal{A}}(j)$) = the jth ranked object in $\mathcal{A}(k, t_1, t_2)$ ($\widetilde{\mathcal{A}}(k, t_1, t_2)$)
- R is an (ε, α)-approximation algorithm of top-$k(t_1, t_2, \sigma)$ if:
    - R returns $\widetilde{\mathcal{A}}(k, t_1, t_2)$ and $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ for $j \in [1, k]$, s.t.

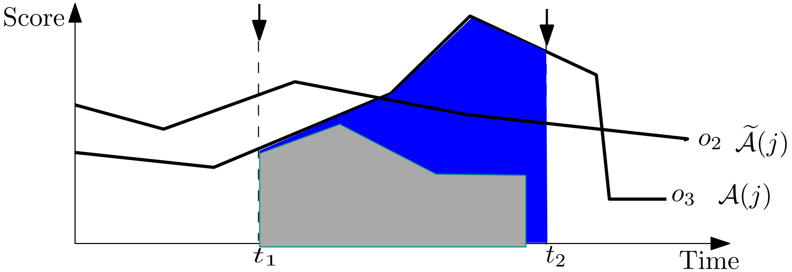
Approximate Solution Notations



$$\sigma_2(t_1, t_2)/\alpha - \varepsilon M \le \tilde{\sigma}_2(t_1, t_2) \le \sigma_2(t_1, t_2) + \varepsilon M$$

- $\mathcal{A}(j)$ ($\widetilde{\mathcal{A}}(j)$) = the jth ranked object in $\mathcal{A}(k, t_1, t_2)$ ($\widetilde{\mathcal{A}}(k, t_1, t_2)$)
- R is an (ε, α)-approximation algorithm of top-$k(t_1, t_2, \sigma)$ if:
  - R returns $\widetilde{\mathcal{A}}(k, t_1, t_2)$ and $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ for $j \in [1, k]$, s.t.
    1. $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ is an (ε, α)-approximation of $\sigma_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$

Approximate Solution Notations



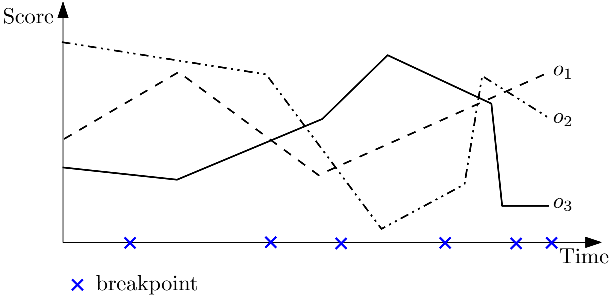
$$\sigma_3(t_1, t_2)/\alpha - \varepsilon M \leq \tilde{\sigma}_2(t_1, t_2) \leq \sigma_3(t_1, t_2) + \varepsilon M$$

- $\mathcal{A}(j)$ ($\widetilde{\mathcal{A}}(j)$) = the jth ranked object in $\mathcal{A}(k, t_1, t_2)$ ($\widetilde{\mathcal{A}}(k, t_1, t_2)$)
- R is an (ε, α)-approximation algorithm of top-$k(t_1, t_2, \sigma)$ if:
  - R returns $\widetilde{\mathcal{A}}(k, t_1, t_2)$ and $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ for $j \in [1, k]$, s.t.
    1. $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ is an (ε, α)-approximation of $\sigma_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$
    2. $\widetilde{\sigma}_{\widetilde{\mathcal{A}}(j)}(t_1, t_2)$ is an (ε, α)-approximation of $\sigma_{\mathcal{A}(j)}(t_1, t_2)$
- Must hold for all k and all temporal intervals.

Breakpoints

Starting from b_0 and moving forward we have:

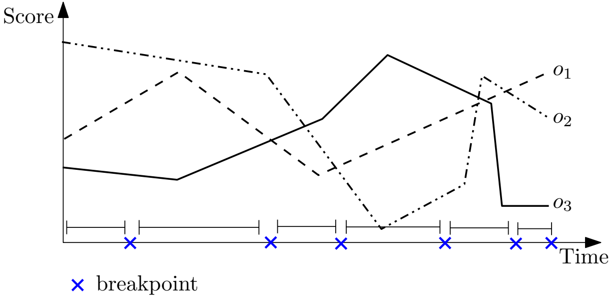
$$b_{j+1} \text{ so} \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in BreakPoints2}(\mathcal{B}_2) \end{cases}$$



Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints2}(\mathcal{B}_2) \end{cases}$$

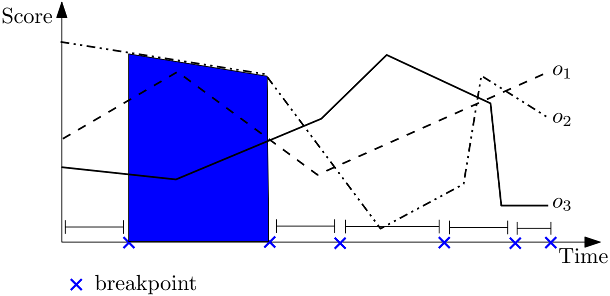


$$\sigma_1(b_j, b_{j+1}) + \sigma_2(b_j, b_{j+1}) + \sigma_3(b_j, b_{j+1}) = \varepsilon M$$

Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints2}(\mathcal{B}_2) \end{cases}$$

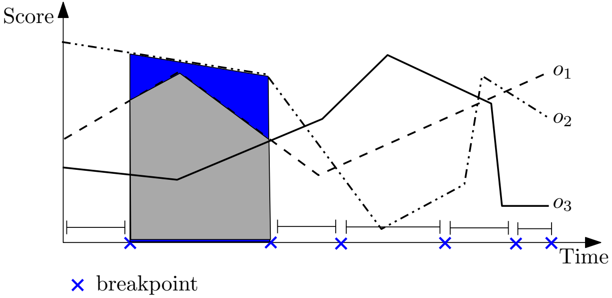


$$\sigma_1(b_j, b_{j+1}) + \underline{\sigma_2(b_j, b_{j+1})} + \sigma_3(b_j, b_{j+1}) = \varepsilon M$$

Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints2}(\mathcal{B}_2) \end{cases}$$

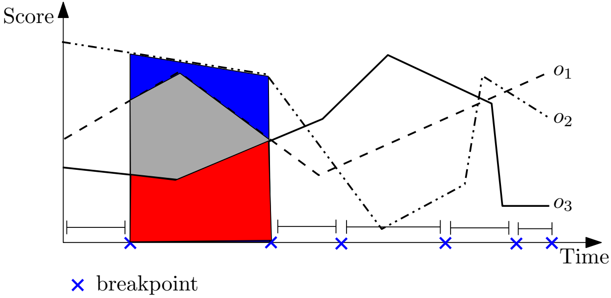


$$\sigma_1(b_j, b_{j+1}) + \sigma_2(b_j, b_{j+1}) + \sigma_3(b_j, b_{j+1}) = \varepsilon M$$

Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints2}(\mathcal{B}_2) \end{cases}$$

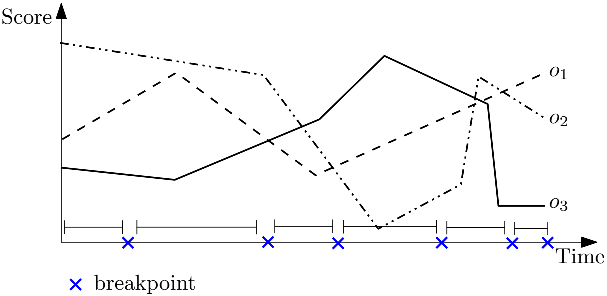


$$\sigma_1(b_j, b_{j+1}) + \sigma_2(b_j, b_{j+1}) + \sigma_3(b_j, b_{j+1}) = \varepsilon M$$

Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints2}(\mathcal{B}_2) \end{cases}$$

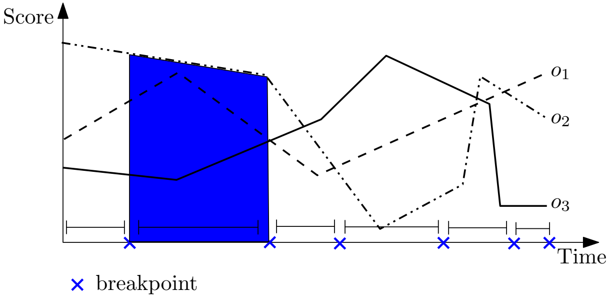


$$max\{\sigma_1(b_j, b_{j+1}), \sigma_2(b_j, b_{j+1}), \sigma_3(b_j, b_{j+1}\} = \varepsilon M$$

Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in BreakPoints2}(\mathcal{B}_2) \end{cases}$$



$$max\{\sigma_1(b_j, b_{j+1}), \underline{\sigma_2(b_j, b_{j+1})}, \sigma_3(b_j, b_{j+1}\} = \varepsilon M$$

Properties of Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints2}(\mathcal{B}_2) \end{cases}$$

- We show how to efficiently construct both types of breakpoints
  - A cost of $O((N/B) log_B N)$ IOs for both types.

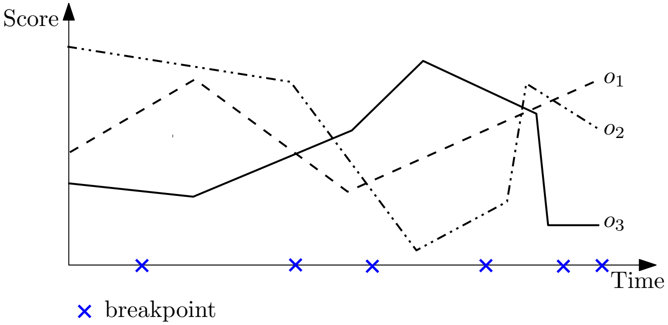
Properties of Breakpoints

Starting from b_0 and moving forward we have:

$$b_{j+1} \text{ so } \begin{cases} \sum_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints1}(\mathcal{B}_1) \\ \max_{i=1}^{m} \sigma_i(b_j, b_{j+1}) = \varepsilon M, & \text{in } \textsc{BreakPoints2}(\mathcal{B}_2) \end{cases}$$

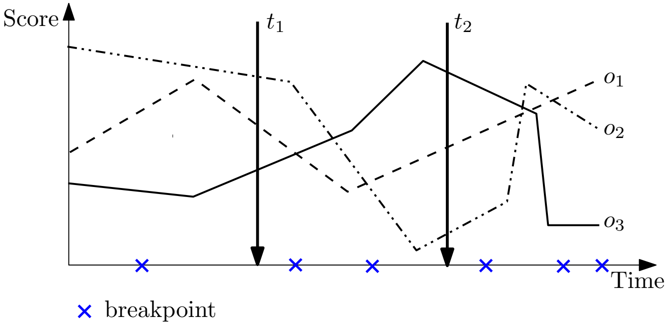
- We show how to efficiently construct both types of breakpoints
  - A cost of $O((N/B)log_B N)$ IOs for both types.
- The theoretical number of breakpoints is $O(1/\varepsilon)$ for both types.
  - BreakPoints2 has much fewer breakpoints than BreakPoints1 in practice.

Answering Queries with Breakpoints



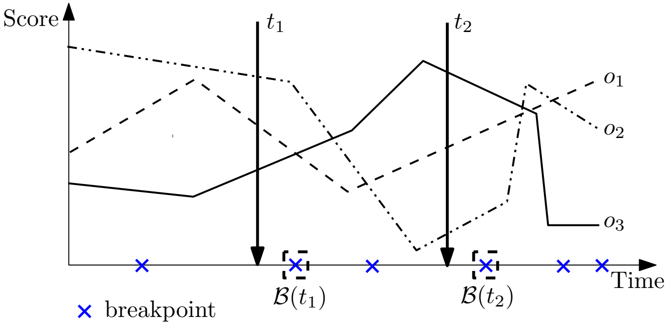
- We show how to answer queries using $\mathcal{B}_1$ or $\mathcal{B}_2$ approximately.
- $\forall(t_1, t_2)$, let $(\mathcal{B}(t_1), \mathcal{B}(t_2))$ be the *approximate interval*
  - $\mathcal{B}(t_1) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_1) \geq t_1$
  - $\mathcal{B}(t_2) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_2) \geq t_2$

Answering Queries with Breakpoints



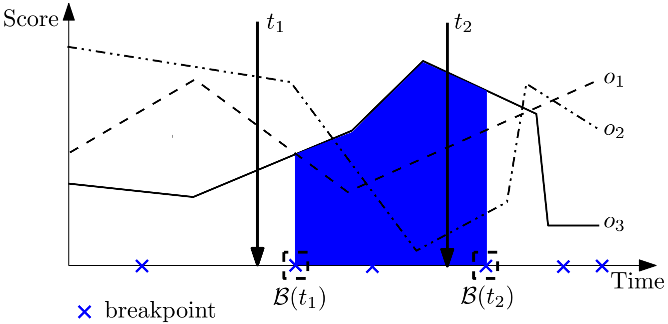
- We show how to answer queries using $\mathcal{B}_1$ or $\mathcal{B}_2$ approximately.
- $\forall(t_1, t_2)$, let $(\mathcal{B}(t_1), \mathcal{B}(t_2))$ be the *approximate interval*
  - $\mathcal{B}(t_1) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_1) \geq t_1$
  - $\mathcal{B}(t_2) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_2) \geq t_2$

Answering Queries with Breakpoints



- We show how to answer queries using $\mathcal{B}_1$ or $\mathcal{B}_2$ approximately.
- $\forall(t_1, t_2)$, let $(\mathcal{B}(t_1), \mathcal{B}(t_2))$ be the *approximate interval*
  - $\mathcal{B}(t_1) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_1) \geq t_1$
  - $\mathcal{B}(t_2) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_2) \geq t_2$

Answering Queries with Breakpoints

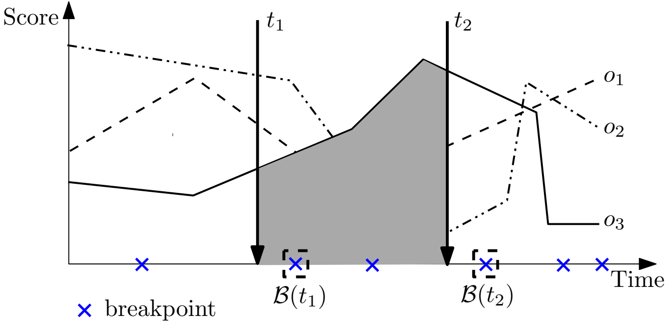


- We show how to answer queries using $\mathcal{B}_1$ or $\mathcal{B}_2$ approximately.
- $\forall(t_1, t_2)$, let $(\mathcal{B}(t_1), \mathcal{B}(t_2))$ be the *approximate interval*
  - $\mathcal{B}(t_1) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_1) \geq t_1$
  - $\mathcal{B}(t_2) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_2) \geq t_2$

Lemma

$\forall(t_1, t_2)$ and its approximate interval $(\mathcal{B}(t_1), \mathcal{B}(t_2))$: $\forall o_i$, $|\sigma_i(t_1, t_2) - \sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))| \leq \varepsilon M$.

Answering Queries with Breakpoints



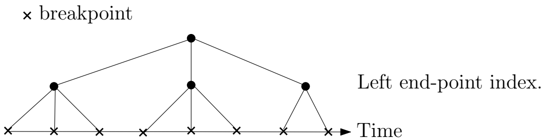
- We show how to answer queries using $\mathcal{B}_1$ or $\mathcal{B}_2$ approximately.
- $\forall(t_1, t_2)$, let $(\mathcal{B}(t_1), \mathcal{B}(t_2))$ be the *approximate interval*
  - $\mathcal{B}(t_1) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_1) \geq t_1$
  - $\mathcal{B}(t_2) = min_{b_i \in \mathcal{B}}$ s.t. $\mathcal{B}(t_2) \geq t_2$

Lemma

$\forall(t_1, t_2)$ and its approximate interval $(\mathcal{B}(t_1), \mathcal{B}(t_2))$: $\forall o_i$,
$|\sigma_i(t_1, t_2) - \sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))| \leq \varepsilon M$.

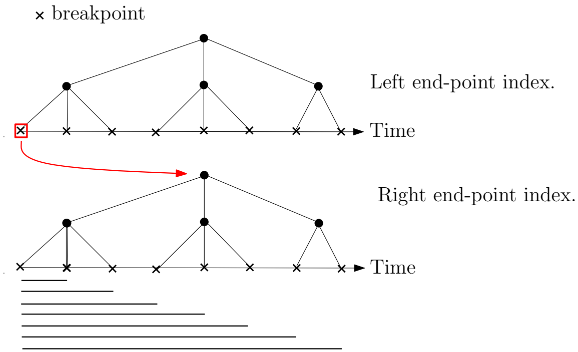
Outline

Querying Breakpoints with Nested B-trees



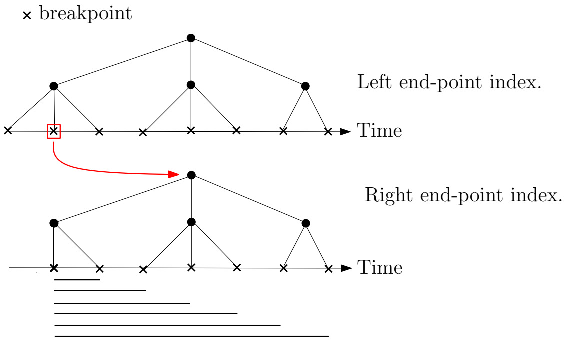
- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
    - For each interval $[b_j, b'_j]$, $\mathcal{A}(k_{max}, b_j, b'_j)$ is computed.

Querying Breakpoints with Nested B-trees



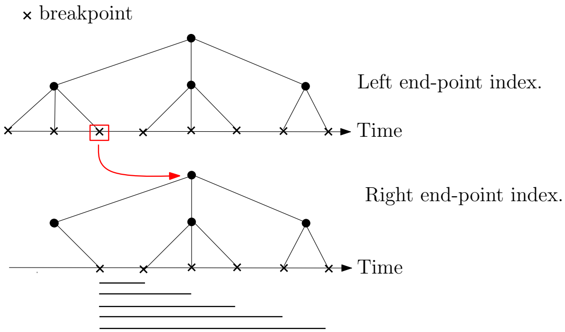
- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b_j']$, $\mathcal{A}(k_{max}, b_j, b_j')$ is computed.

Querying Breakpoints with Nested B-trees



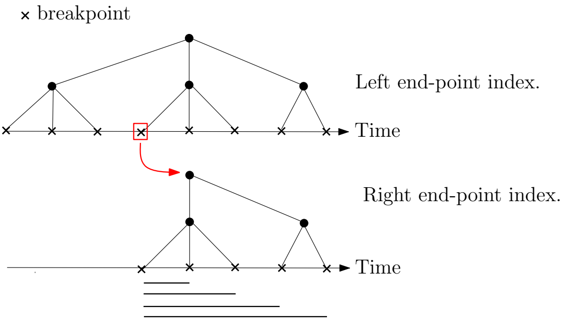
- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b'_j]$, $\mathcal{A}(k_{max}, b_j, b'_j)$ is computed.

Querying Breakpoints with Nested B-trees



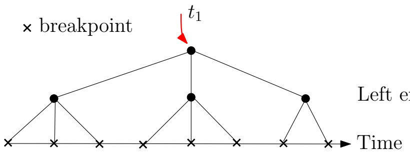
- Query1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b_j']$, $\mathcal{A}(k_{max}, b_j, b_j')$ is computed.

Querying Breakpoints with Nested B-trees



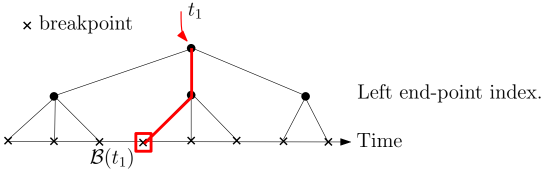
- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b_j']$, $\mathcal{A}(k_{max}, b_j, b_j')$ is computed.

Querying Breakpoints with Nested B-trees



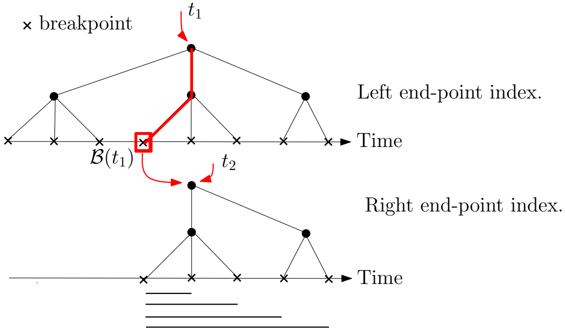
- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b_j']$, $\mathcal{A}(k_{max}, b_j, b_j')$ is computed.
- At query time we probe first-level B-tree with t_1 to get $\mathcal{B}(t_1)$.

Querying Breakpoints with Nested B-trees



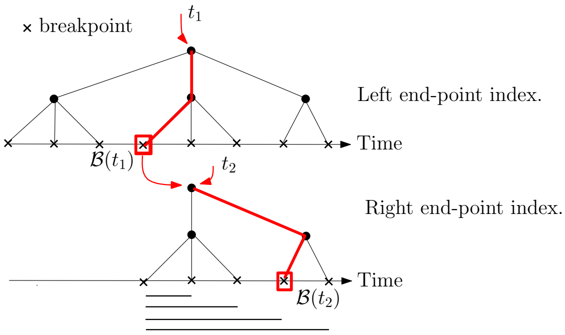
- Query1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b'_j]$, $\mathcal{A}(k_{max}, b_j, b'_j)$ is computed.
- At query time we probe first-level B-tree with t_1 to get $\mathcal{B}(t_1)$.

Querying Breakpoints with Nested B-trees



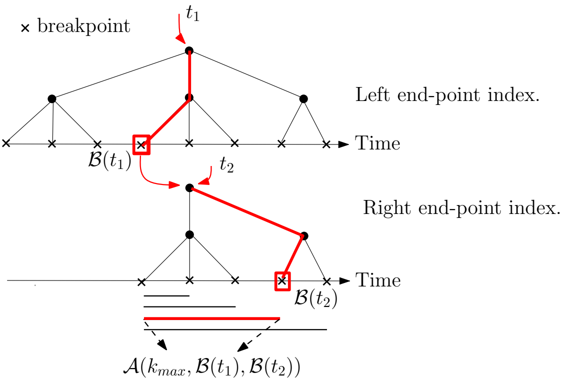
- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b_j']$, $\mathcal{A}(k_{max}, b_j, b_j')$ is computed.
- At query time we probe first-level B-tree with t_1 to get $\mathcal{B}(t_1)$.
- We probe $\mathcal{B}(t_1)$'s associated nested B-tree to get $\mathcal{B}(t_2)$.

Querying Breakpoints with Nested B-trees



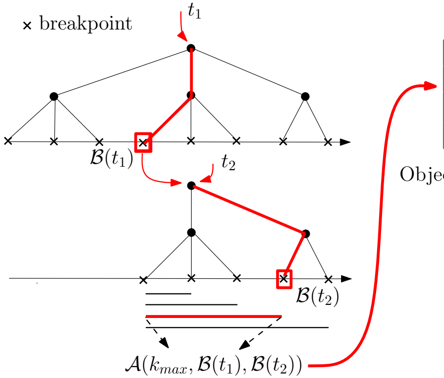
- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
    - For each interval $[b_j, b'_j]$, $\mathcal{A}(k_{max}, b_j, b'_j)$ is computed.
- At query time we probe first-level B-tree with t_1 to get $\mathcal{B}(t_1)$.
- We probe $\mathcal{B}(t_1)$'s associated nested B-tree to get $\mathcal{B}(t_2)$.

Querying Breakpoints with Nested B-trees



- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b_j']$, $\mathcal{A}(k_{max}, b_j, b_j')$ is computed.
- At query time we probe first-level B-tree with t_1 to get $\mathcal{B}(t_1)$.
- We probe $\mathcal{B}(t_1)$'s associated nested B-tree to get $\mathcal{B}(t_2)$.

Querying Breakpoints with Nested B-trees

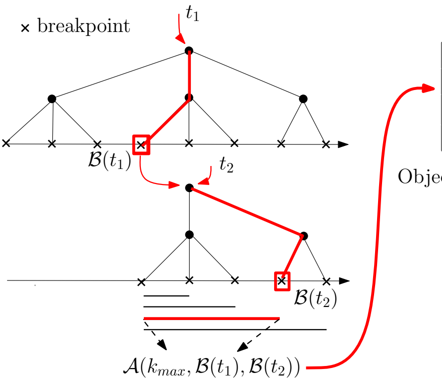


o_i	$\sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))$
o_{ℓ_1}	$\sigma_{\ell_1}(\mathcal{B}(t_1), \mathcal{B}(t_2))$
$\vdots$	$\vdots$
$o_{\ell_{kmax}}$	$\sigma_{\ell_{kmax}}(\mathcal{B}(t_1), \mathcal{B}(t_2))$

Objects ordered in descending order of $\sigma_i(.)$

- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b_j']$, $\mathcal{A}(k_{max}, b_j, b_j')$ is computed.
- At query time we probe first-level B-tree with t_1 to get $\mathcal{B}(t_1)$.
- We probe $\mathcal{B}(t_1)$'s associated nested B-tree to get $\mathcal{B}(t_2)$.

Querying Breakpoints with Nested B-trees



o_i	$\sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))$
o_{ℓ_1}	$\sigma_{\ell_1}(\mathcal{B}(t_1), \mathcal{B}(t_2))$
$\vdots$	$\vdots$
$o_{\ell_{k_{max}}}$	$\sigma_{\ell_{k_{max}}}(\mathcal{B}(t_1), \mathcal{B}(t_2))$

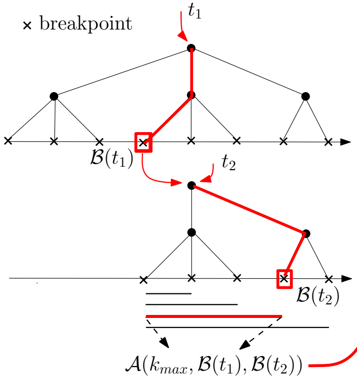
Objects ordered in descending order of $\sigma_i(.)$

Take top-k

$\widetilde{\mathcal{A}}(k, t_1, t_2)$

- QUERY1 indexes all $\binom{n}{2}$ intervals of breakpoints $\mathcal{B}$.
  - For each interval $[b_j, b'_j]$, $\mathcal{A}(k_{max}, b_j, b'_j)$ is computed.
- At query time we probe first-level B-tree with t_1 to get $\mathcal{B}(t_1)$.
- We probe $\mathcal{B}(t_1)$'s associated nested B-tree to get $\mathcal{B}(t_2)$.
- The approximate answer $\widetilde{\mathcal{A}}(k, t_1, t_2)$ is returned.

Querying Breakpoints with Nested B-trees

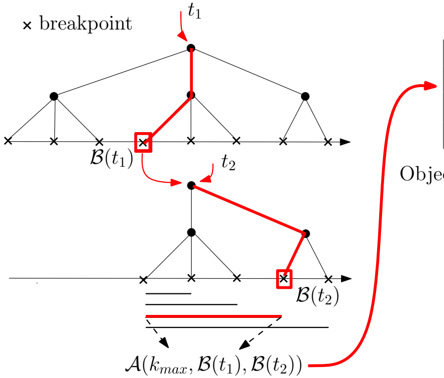


o_i	$\sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))$
o_{ℓ_1}	$\sigma_{\ell_1}(\mathcal{B}(t_1), \mathcal{B}(t_2))$
$\vdots$	$\vdots$
$o_{\ell_{k_{max}}}$	$\sigma_{\ell_{k_{max}}}(\mathcal{B}(t_1), \mathcal{B}(t_2))$

Objects ordered in descending order of $\sigma_i(.)$

- We prove QUERY1 has the following properties:
  - Index size $O((1/\varepsilon)^2 k_{max}/B)$.
  - Query cost $O(k/B + \log_B(1/\varepsilon))$.
  - $(\varepsilon, 1)$-approximation.

Querying Breakpoints with Nested B-trees

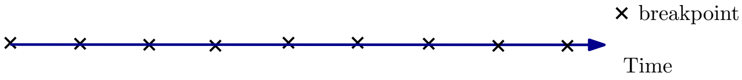


o_i	$\sigma_i(\mathcal{B}(t_1), \mathcal{B}(t_2))$
o_{ℓ_1}	$\sigma_{\ell_1}(\mathcal{B}(t_1), \mathcal{B}(t_2))$
$\vdots$	$\vdots$
$o_{\ell_{k_{max}}}$	$\sigma_{\ell_{k_{max}}}(\mathcal{B}(t_1), \mathcal{B}(t_2))$

Objects ordered in descending order of $\sigma_i(.)$

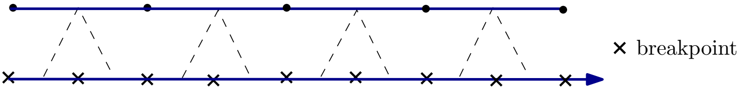
- We prove Query1 has the following properties:
  - Index size $O((1/\varepsilon)^2 k_{max}/B)$.
  - Query cost $O(k/B + \log_B(1/\varepsilon))$.
  - $(\varepsilon, 1)$-approximation.
- Query2 reduces space to $O((1/\varepsilon)k_{max}/B)$.
  - $(\varepsilon, 2log(1/\varepsilon))$-approximation.
  - Query cost $O(k \log(1/\varepsilon) \log_B k)$.

Querying Breakpoints with Dyadic Intervals



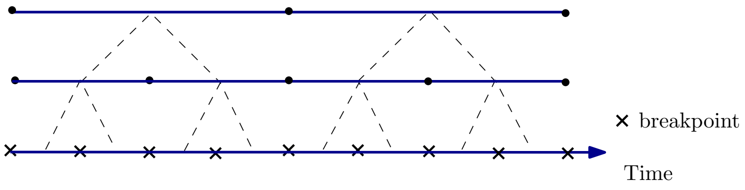
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.

Querying Breakpoints with Dyadic Intervals



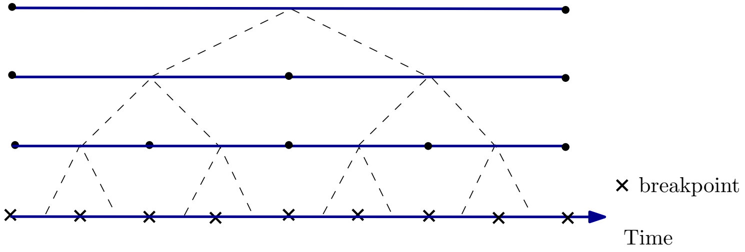
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.

Querying Breakpoints with Dyadic Intervals



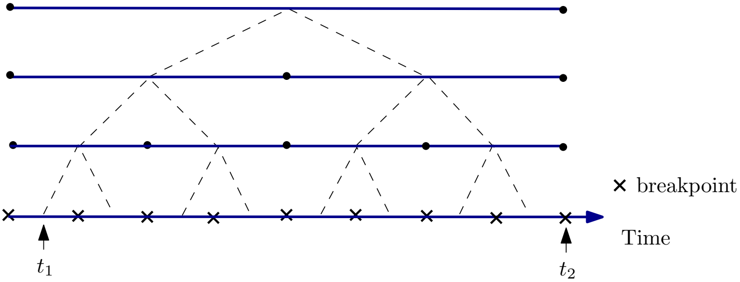
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.

Querying Breakpoints with Dyadic Intervals



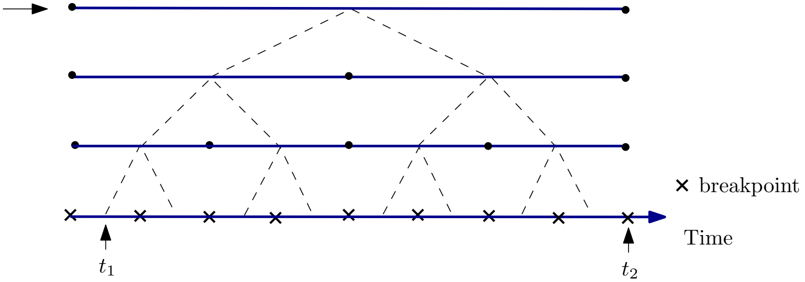
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.

Querying Breakpoints with Dyadic Intervals



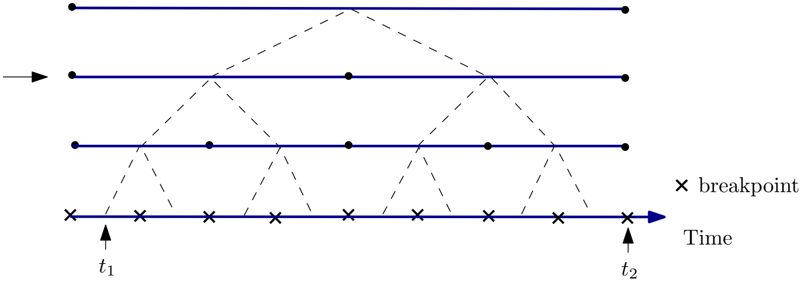
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.

Querying Breakpoints with Dyadic Intervals



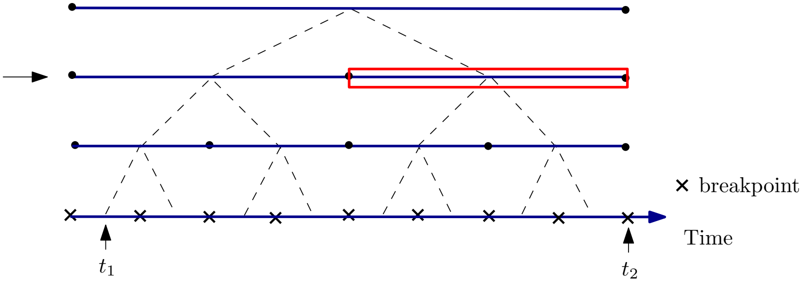
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.

Querying Breakpoints with Dyadic Intervals



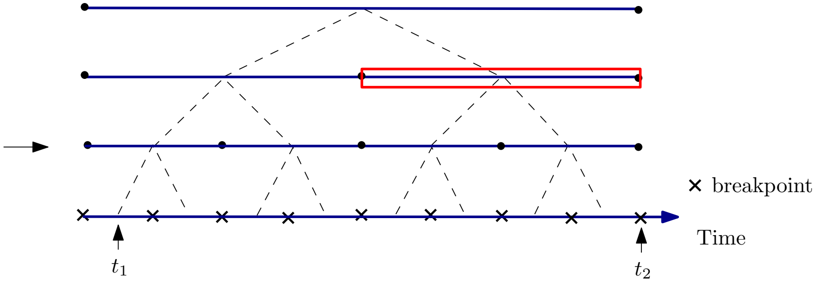
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.

Querying Breakpoints with Dyadic Intervals



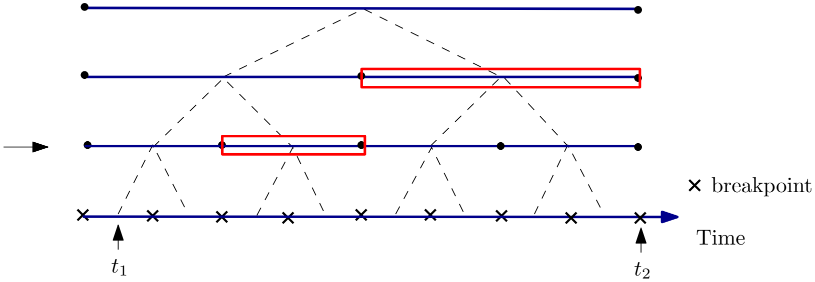
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.

Querying Breakpoints with Dyadic Intervals



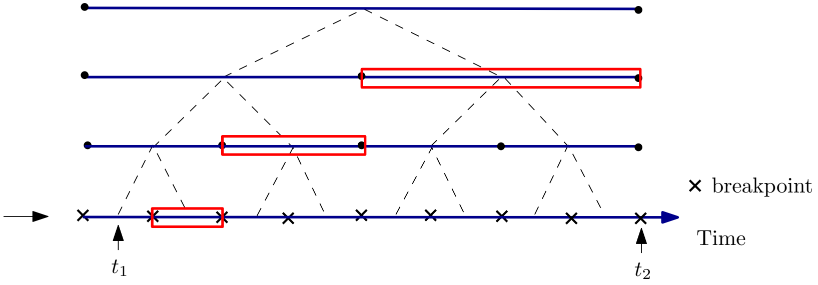
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.

Querying Breakpoints with Dyadic Intervals



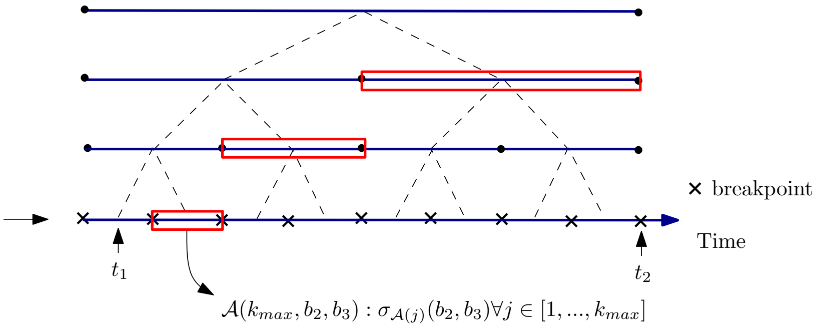
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.

Querying Breakpoints with Dyadic Intervals



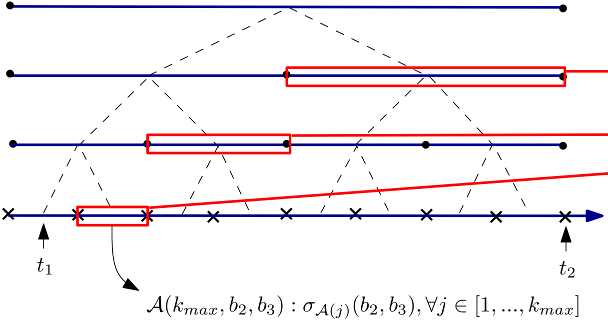
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.

Querying Breakpoints with Dyadic Intervals



- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.
- At each dyadic interval $[b_i, b_j]$ we store $\mathcal{A}(k_{max}, b_i, b_j)$.
  - There are at most $2log(1/\varepsilon)$ intervals and $2klog(1/\varepsilon)$ candidates.

Querying Breakpoints with Dyadic Intervals



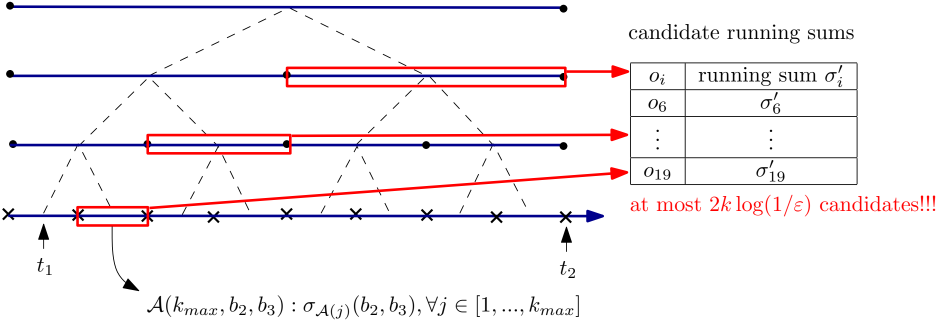
candidate running sums

o_i	running sum σ'_i
o_6	σ'_6
⋮	⋮
o_{19}	σ'_{19}

at most $2k \log(1/\varepsilon)$ candidates!!!

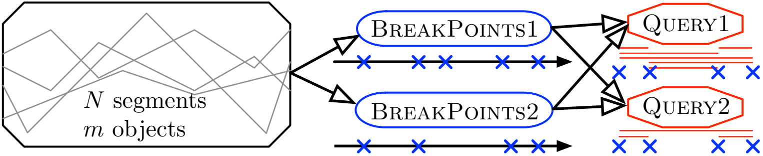
- QUERY2 indexes all dyadic intervals over the breakpoints $\mathcal{B}$
  - The intervals represent the span of nodes in a balanced binary tree.
- Consider a query over $[t_1, t_2]$.
- At each dyadic interval $[b_i, b_j]$ we store $\mathcal{A}(k_{max}, b_i, b_j)$.
  - There are at most $2log(1/\varepsilon)$ intervals and $2klog(1/\varepsilon)$ candidates.

Querying Breakpoints with Dyadic Intervals



- We prove QUERY2 has the following properties:
  - Index size $O((1/\varepsilon)k_{max}/B)$.
  - Query cost $O(k \log(1/\varepsilon) \log_B k)$.
  - $(\varepsilon, 2\log(1/\varepsilon))$-approximation.

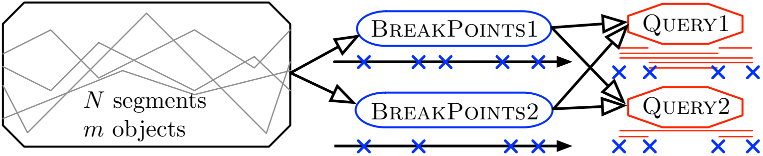
Combining Breakpoints with Queries



We consider the following algorithms:

- Appx1-B: (Query1, BreakPoints1)
- Appx2-B: (Query2, BreakPoints1)
- Appx1: (Query1, BreakPoints2)
- Appx2: (Query2, BreakPoints2)
- Appx2+: (Query2, BreakPoints2) and Discovers candidates' exact aggregate score using B-tree from Exact2 (B-tree forest).

Combining Breakpoints with Queries



We consider the following algorithms:

- Appx1-B: (Query1, BreakPoints1)
- Appx2-B: (Query2, BreakPoints1)
- Appx1: (Query1, BreakPoints2)
- Appx2: (Query2, BreakPoints2)
- Appx2+: (Query2, BreakPoints2) and Discovers candidates' exact aggregate score using B-tree from Exact2 (B-tree forest).

Experiments: Setup

- Our algorithms are designed to efficiently handle I/Os.
  - All algorithms are implemented in C++ using TPIE.

Experiments: Setup

- Our algorithms are designed to efficiently handle I/Os.
  - All algorithms are implemented in C++ using TPIE.
- All experiments performed on Linux machine with:
  - Intel Core i7-2600 3.4GHz CPU
  - 8GB of memory
  - 1TB hard drive

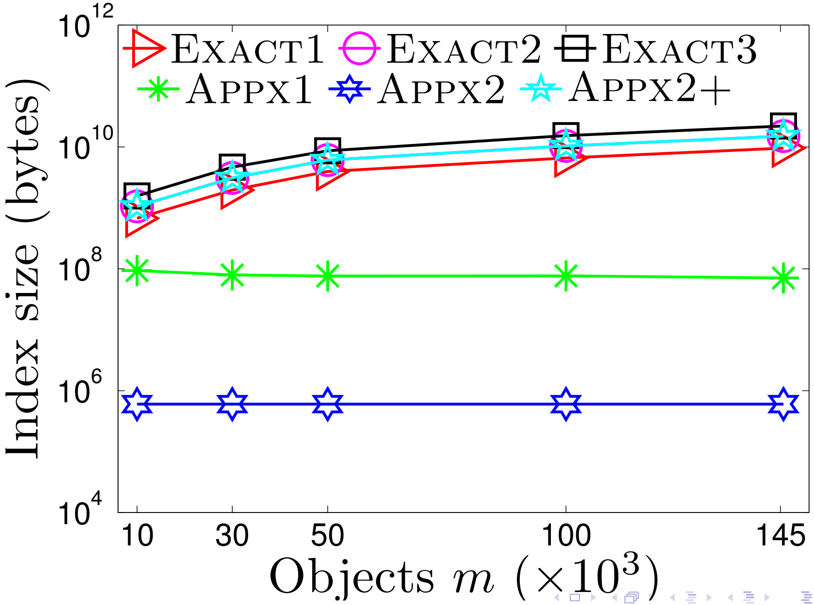
Experiments: Setup

- Our algorithms are designed to efficiently handle I/Os.
  - All algorithms are implemented in C++ using TPIE.
- All experiments performed on Linux machine with:
  - Intel Core i7-2600 3.4GHz CPU
  - 8GB of memory
  - 1TB hard drive
- We use two real large datasets:
  - *Temp* is a temperature dataset from the MesoWest Project.
    - contains measurements from Jan 1997 to Oct 2011.
    - there are $m = 145,628$ objects with average $n_{avg} = 17,833$.
  - *Meme* is obtained from the Memetracker Project.
    - tracks the frequency of popular quotes over time.
    - there are $m = 1.5$ million objects with $n_{avg} = 67$.

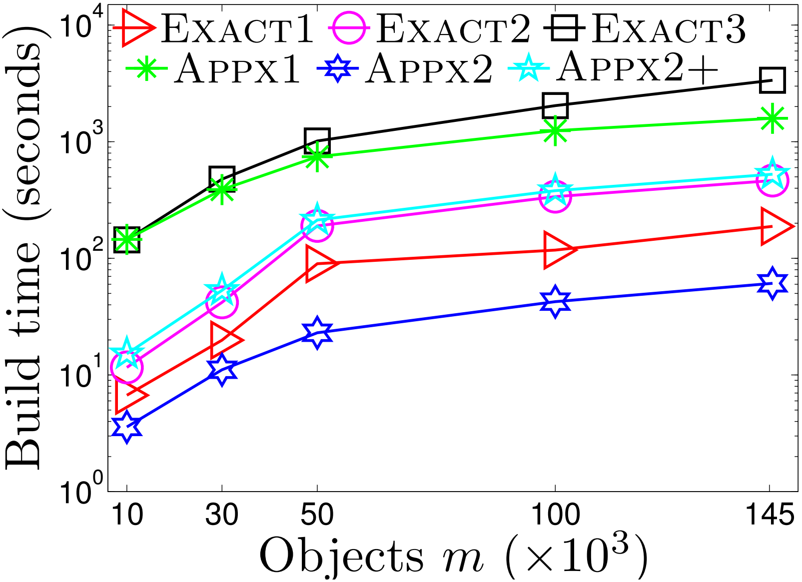
Experiments: Default Values

Parameter	Symbol	Default value
dataset		*Temp*
number of objects	m	50,000
average object line segments	n_{avg}	1,000
max top-k value	k_{max}	200
top-k value	k	50
number of breakpoints	$r = (1/\varepsilon)$	500
query interval size	$(t_2 - t_1)$	20% T
TPIE disk block size		4KB

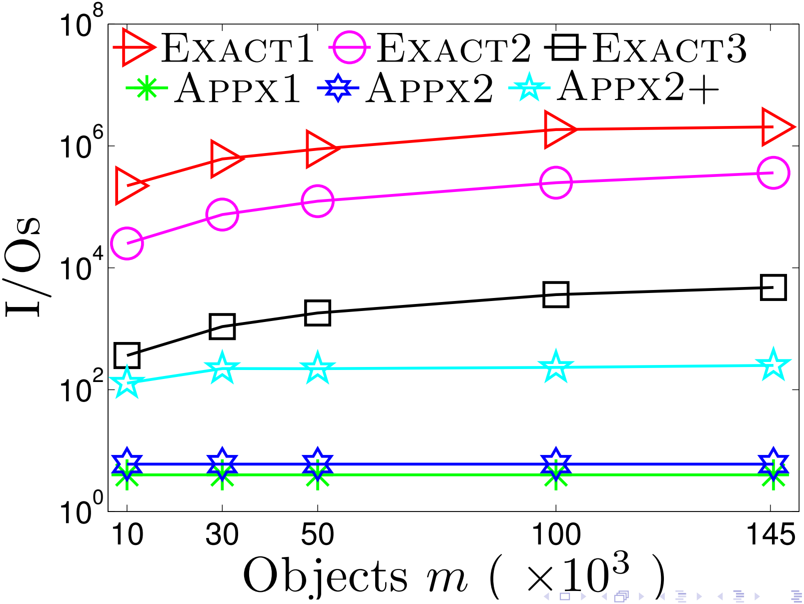
Experiment: Index size.



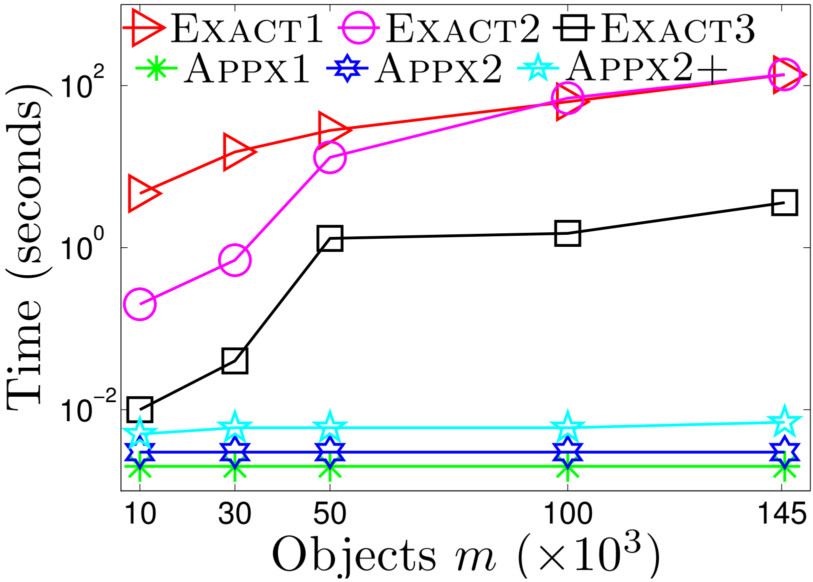
Experiment: Build time.



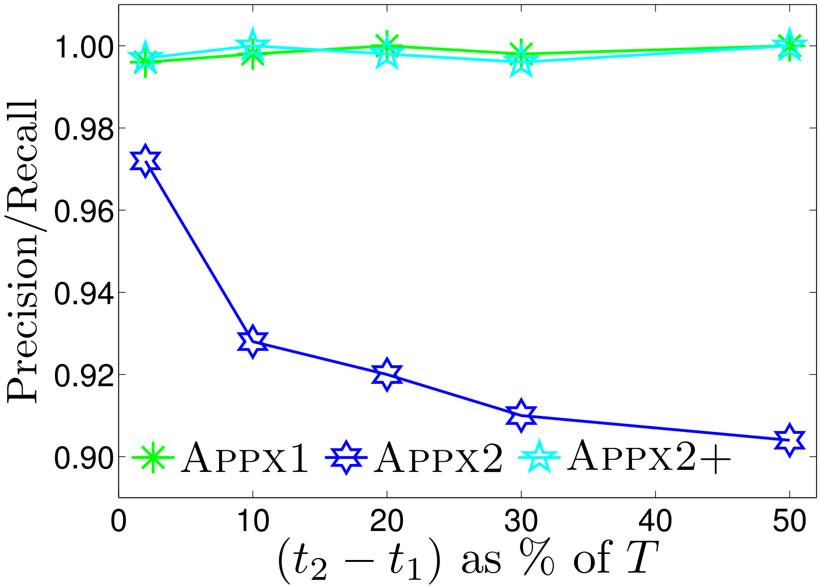
Experiment: Query I/Os.



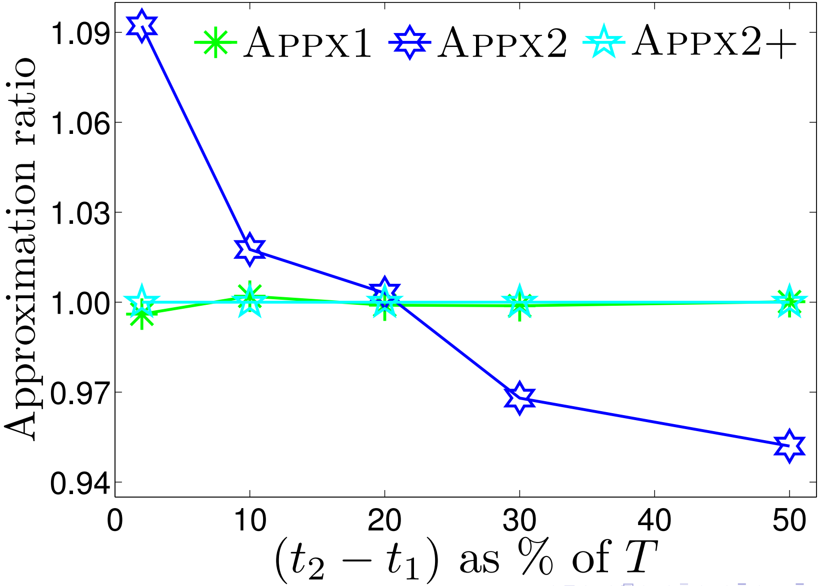
Experiment: Query time.



Experiment: Precision/Recall.



Experiment: Ratio.



Conclusions

- We studied ranking large temporal data using aggregate scores over a query interval.

Conclusions

- We studied ranking large temporal data using aggregate scores over a query interval.
- Our most efficient exact technique EXACT3 is more efficient than baseline solutions.
  - Approximations offer even more improvements.

Conclusions

- We studied ranking large temporal data using aggregate scores over a query interval.
- Our most efficient exact technique EXACT3 is more efficient than baseline solutions.
  - Approximations offer even more improvements.
- Future work includes ranking with holistic aggregations and extending to distributed settings.

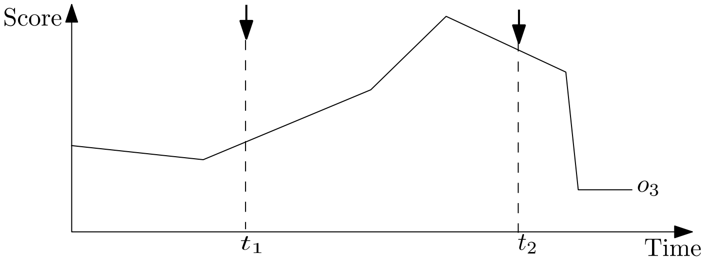
The End

Thank You

Q and A

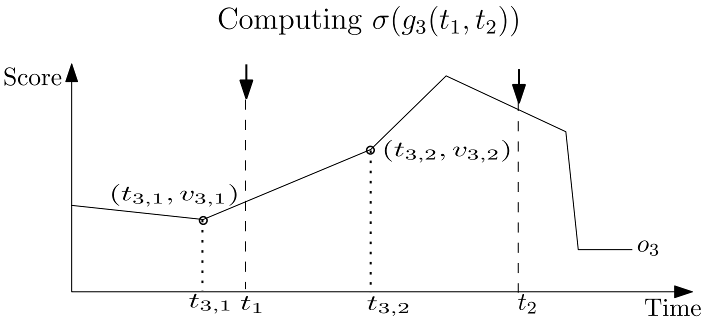
Baseline Solution

Computing $\sigma(g_3(t_1, t_2))$



1. Initialize sum $s_3 = 0$ for object o_3

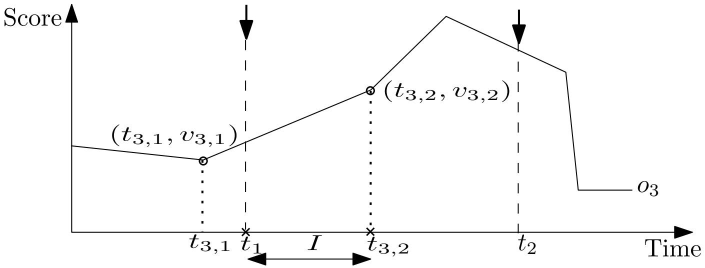
Baseline Solution



1. Initialize sum $s_3 = 0$ for object o_3
2. For each segment ℓ of g_3 defined by $(t_{3,j}, v_{3,j}), (t_{3,j+1}, v_{3,j+1})$

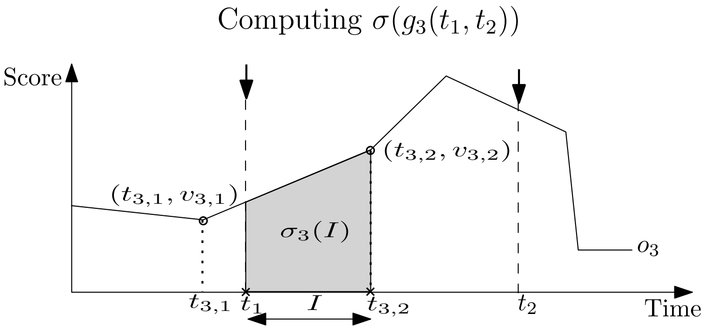
Baseline Solution

Computing $\sigma(g_3(t_1, t_2))$



1. Initialize sum $s_3 = 0$ for object o_3
2. For each segment ℓ of g_3 defined by $(t_{3,j}, v_{3,j}), (t_{3,j+1}, v_{3,j+1})$
   - Define $\mathcal{I} = [t_1, t_2] \cap [t_{3,j}, t_{3,j+1}]$

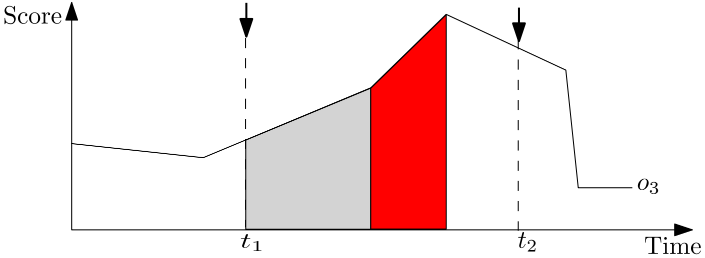
Baseline Solution



1. Initialize sum $s_3 = 0$ for object o_3
2. For each segment ℓ of g_3 defined by $(t_{3,j}, v_{3,j}), (t_{3,j+1}, v_{3,j+1})$
   - Define $\mathcal{I} = [t_1, t_2] \cap [t_{3,j}, t_{3,j+1}]$
   - Update $s_3 = s_3 + \sigma_3(\mathcal{I})$

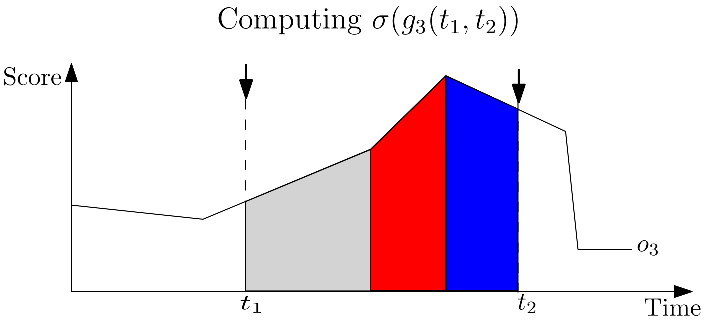
Baseline Solution

Computing $\sigma(g_3(t_1, t_2))$



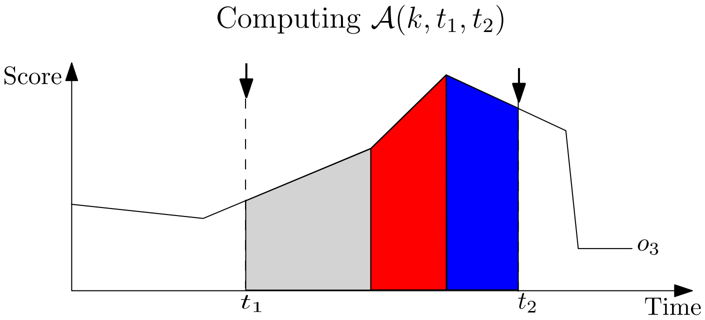
1. Initialize sum $s_3 = 0$ for object o_3
2. For each segment ℓ of g_3 defined by $(t_{3,j}, v_{3,j}), (t_{3,j+1}, v_{3,j+1})$
   - Define $\mathcal{I} = [t_1, t_2] \cap [t_{3,j}, t_{3,j+1}]$
   - Update $s_3 = s_3 + \sigma_3(\mathcal{I})$

Baseline Solution



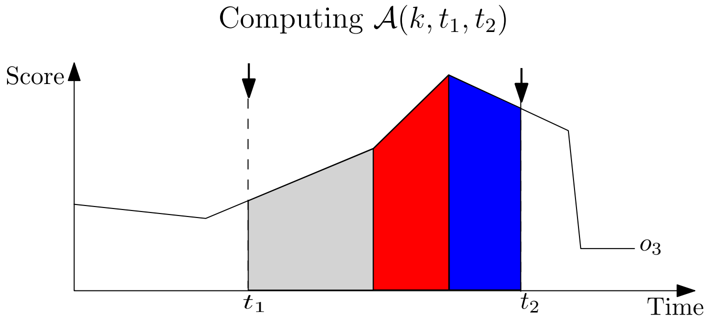
1. Initialize sum $s_3 = 0$ for object o_3
2. For each segment ℓ of g_3 defined by $(t_{3,j}, v_{3,j}), (t_{3,j+1}, v_{3,j+1})$
   - Define $\mathcal{I} = [t_1, t_2] \cap [t_{3,j}, t_{3,j+1}]$
   - Update $s_3 = s_3 + \sigma_3(\mathcal{I})$

Baseline Solution



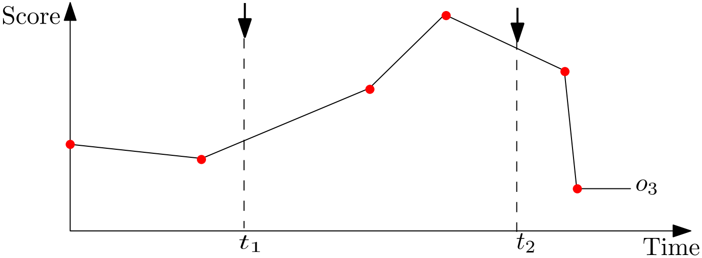
- Compute s_i for all objects $i \in [1, m]$.
  - Insert s_i's into priority queue of size k to get $\mathcal{A}(k, t_1, t_2)$.

Baseline Solution



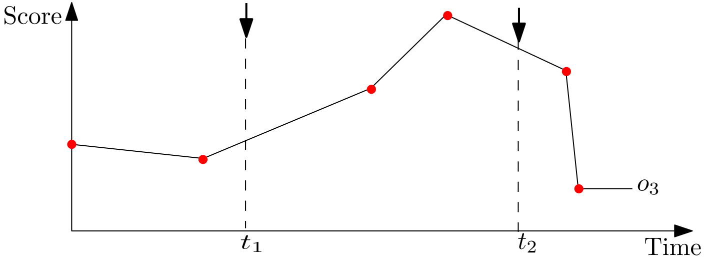
- Compute s_i for all objects $i \in [1, m]$.
  - Insert s_i's into priority queue of size k to get $\mathcal{A}(k, t_1, t_2)$.
- Naive cost: $O(N + mlogk)$

Improved Baseline Solution using B-tree



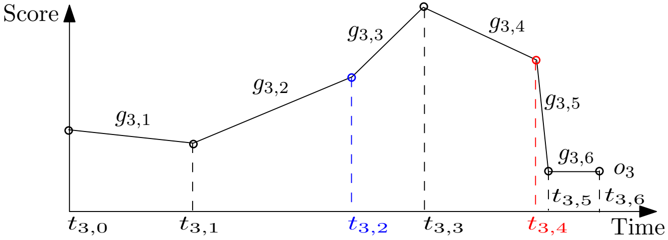
- For each line segment $\ell = \{(t_{i,j}, v_{i,j}), (t_{i,j+1}, v_{i,j+1})\}$
  - Index left end-point $t_{i,j}$ in B-tree.
  - The value associated with $t_{i,j}$ is ℓ.

Improved Baseline Solution using B-tree



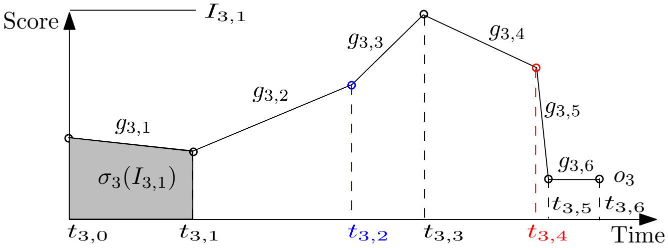
- For each line segment $\ell = \{(t_{i,j}, v_{i,j}), (t_{i,j+1}, v_{i,j+1})\}$
  - Index left end-point $t_{i,j}$ in B-tree.
  - The value associated with $t_{i,j}$ is ℓ.
- Query cost: $O(log_B N + \frac{\sum_{i=1}^{m} q_i}{B} + (m/B) log_B k)$
  - q_i = number of ℓ overlapping $[t_1, t_2]$
- We denote this query EXACT1.

Improved Solution using Prefix Sums and B-tree Forest



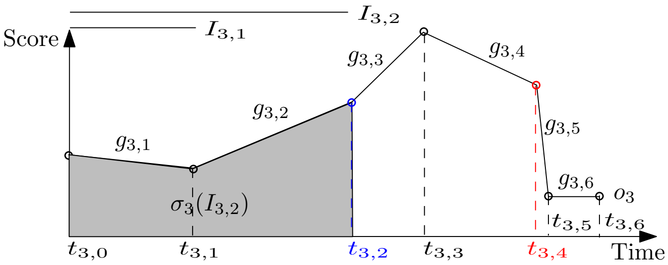
- $g_i = \cup g_{i,j}$
- $g_{i,j}$ is defined by $((t_{i,j-1}, v_{i,j-1}), (t_{i,j}, v_{i,j}))$ for $j \in \{1, \ldots, n_i\}$

Improved Solution using Prefix Sums and B-tree Forest



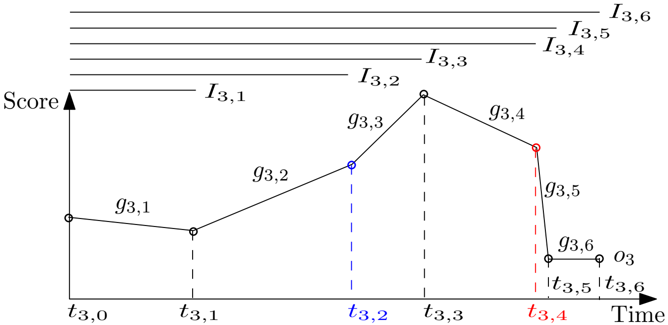
- $g_i = \cup g_{i,j}$
- $g_{i,j}$ is defined by $((t_{i,j-1}, v_{i,j-1}), (t_{i,j}, v_{i,j}))$ for $j \in \{1, \ldots, n_i\}$
- Let $I_{i,\ell} = [t_{i,0}, t_{i,\ell}]$ for $\ell = 1, \ldots, n_i$ and compute $\sigma_i(I_{i,\ell})$

Improved Solution using Prefix Sums and B-tree Forest



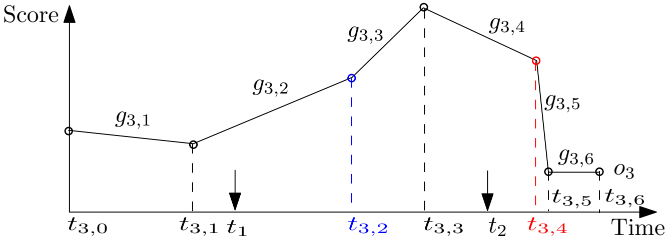
- $g_i = \cup g_{i,j}$
- $g_{i,j}$ is defined by $((t_{i,j-1}, v_{i,j-1}), (t_{i,j}, v_{i,j}))$ for $j \in \{1, \dots, n_i\}$
- Let $I_{i,\ell} = [t_{i,0}, t_{i,\ell}]$ for $\ell = 1, \dots, n_i$ and compute $\sigma_i(I_{i,\ell})$

Improved Solution using Prefix Sums and B-tree Forest



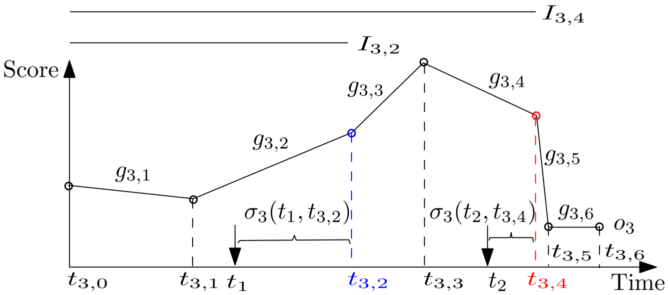
- $g_i = \cup g_{i,j}$
- $g_{i,j}$ is defined by $((t_{i,j-1}, v_{i,j-1}), (t_{i,j}, v_{i,j}))$ for $j \in \{1, \dots, n_i\}$
- Let $I_{i,\ell} = [t_{i,0}, t_{i,\ell}]$ for $\ell = 1, \dots, n_i$ and compute $\sigma_i(I_{i,\ell})$

Improved Solution using Prefix Sums and B-tree Forest



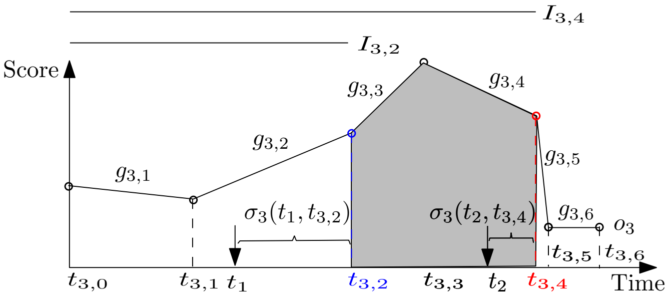
- Let $t_{i,L} = successor(t_{i,1})$ and $t_{i,R} = successor(t_{i,2})$

Improved Solution using Prefix Sums and B-tree Forest



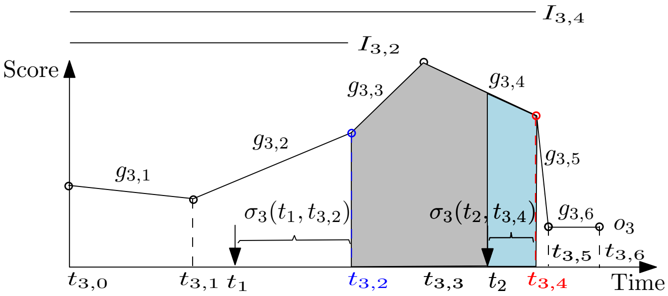
- Let $t_{i,L} = successor(t_{i,1})$ and $t_{i,R} = successor(t_{i,2})$
- $\sigma_i(t_1, t_2) = \sigma_i(I_{i,R}) - \sigma_i(I_{i,L}) - \sigma_i(t_2, t_{i,R}) + \sigma_i(t_1, t_{i,L})$

Improved Solution using Prefix Sums and B-tree Forest



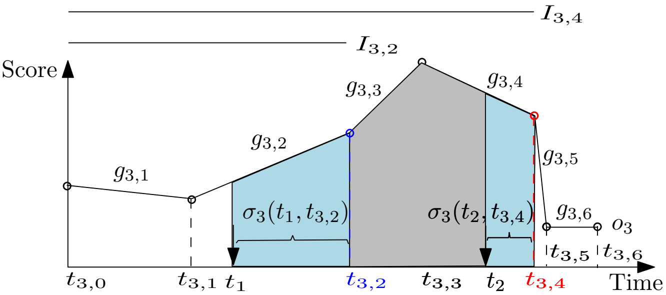
- Let $t_{i,L} = successor(t_{i,1})$ and $t_{i,R} = successor(t_{i,2})$
- $\sigma_i(t_1, t_2) = \sigma_i(I_{i,R}) - \sigma_i(I_{i,L}) - \sigma_i(t_2, t_{i,R}) + \sigma_i(t_1, t_{i,L})$

Improved Solution using Prefix Sums and B-tree Forest



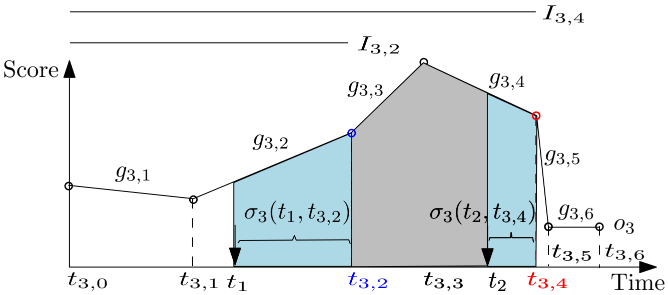
- Let $t_{i,L} = successor(t_{i,1})$ and $t_{i,R} = successor(t_{i,2})$
- $\sigma_i(t_1, t_2) = \sigma_i(I_{i,R}) - \sigma_i(I_{i,L}) - \sigma_i(t_2, t_{i,R}) + \sigma_i(t_1, t_{i,L})$

Improved Solution using Prefix Sums and B-tree Forest



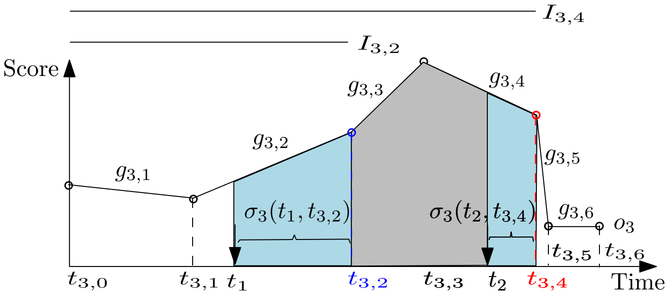
- Let $t_{i,L} = successor(t_{i,1})$ and $t_{i,R} = successor(t_{i,2})$
- $\sigma_i(t_1, t_2) = \sigma_i(I_{i,R}) - \sigma_i(I_{i,L}) - \sigma_i(t_2, t_{i,R}) + \sigma_i(t_1, t_{i,L})$

Improved Solution using Prefix Sums and B-tree Forest



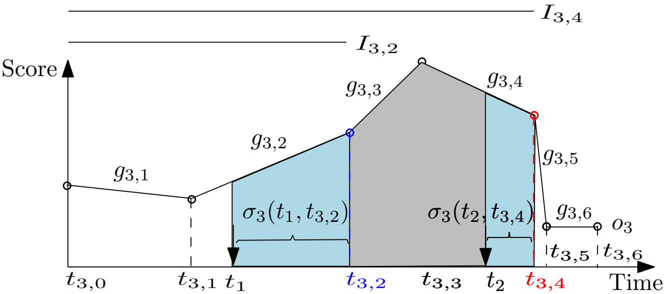
- Let $t_{i,L} = successor(t_{i,1})$ and $t_{i,R} = successor(t_{i,2})$
- $\sigma_i(t_1, t_2) = \sigma_i(I_{i,R}) - \sigma_i(I_{i,L}) - \sigma_i(t_2, t_{i,R}) + \sigma_i(t_1, t_{i,L})$
- Use a B-tree forest to index $(t_{3,\ell}, (g_{i,\ell}, \sigma_i(I_{i,\ell}))$
  - Each o_i indexed in a separate B-tree
  - Query cost is $O(\sum_{i=1}^{m} log_B n_i + (m/B) log_B k)$
- We denote this query EXACT2.

Improved Solution using Prefix Sums and B-tree Forest



- Our B-tree forest solution requires *m* B-trees.
  - Query time improves from baseline.
  - Opening/Closing *m* B-trees expensive for large *m*.

Improved Solution using Prefix Sums and B-tree Forest



- Our B-tree forest solution requires *m* B-trees.
  - Query time improves from baseline.
  - Opening/Closing *m* B-trees expensive for large *m*.
- We show how to solve a query using a single interval tree.