Outline

1. Motivation: XML Fragmentation
2. Problem Definitions
3. Previous Work
4. Streaming Algorithms for Partitioning Integer Sequences
5. Streaming Algorithms for Partitioning Trees
6. Outlook
Motivation
Querying massive XML Databases

[Diagram showing the interaction between a client and an XML DB server for querying massive XML databases.]
Querying massive XML Databases
Querying massive XML Databases

Distributed Query Processing
How to fragment XML Documents?

- Structured (taking XML schema into account)
- Ad-hoc

Survey: [Braganholo, Mattoso, SIGMOD 2014]

Important: Fragments are of similar sizes for good load balancing

Algorithmic Perspective

Challanging if XML documents are massive

Objective of this Work

- Develop space efficient streaming algorithms for fragmenting XML documents
- Focus on load balancing aspect
Problem Definitions
Partitioning Trees: Remove $p - 1$ edges from a node-weighted tree s.t. maximum weight of the resulting subtrees is minimized

![Diagram of a tree with weighted nodes and edges]
Partitioning Trees: Remove $p - 1$ edges from a node-weighted tree s.t. maximum weight of the resulting subtrees is minimized

- $n$: number of nodes of input tree ($n = 9$)
- $p$: number of partitions to be created ($p = 3$)
- $B$: Bottleneck value, weight of heaviest subtree ($B = 7$)
- $B^*$: Bottleneck value of optimal partitioning ($B^* = 7$)
Partitioning Integer Sequences: Split sequence \( X = X[1] \ldots X[n] \) into \( p \) blocks such that maximum weight of a block is minimized.

\[
X = \begin{array}{cccc}
5 & 6 & 11 & 2 & 9 \\
\sum = 33 \\
\end{array} \quad | \quad \begin{array}{cccc}
14 & 3 & 8 & 1 \\
\sum = 26 \\
\end{array} \quad | \quad \begin{array}{cccc}
11 & 22 \\
\sum = 33 \\
\end{array}
\]

- \( n \): length of sequence (\( n = 11 \))
- \( p \): number of partitions to be created (\( p = 3 \))
- \( B \): Bottleneck value, weight of heaviest partition (\( B = 33 \))
- \( B^* \): Bottleneck value of optimal partitioning (\( B^* = 33 ? \))
Streaming

- **Objective:** compute some function $f(x_1, \ldots, x_n)$ given only sequential access

  How much RAM is required for the computation of $f$?

- **Motivation:** massive data sets (too large for storage in RAM)

- **Streaming Complexity**
  - Number of passes $p$, usually $\in O(1)$ *this talk:* $p = 1, 2$
  - Memory space $s \in o(n)$
  - Update-time $t$, usually $\in O(1)$ (or $O(\log n)$)
Partitioning Sequences in the Streaming Model:

- Input Stream: sequence $X = X_1 X_2 \ldots X_n$
- Output: positions of partition separators
Partitioning Sequences in the Streaming Model:
- Input Stream: sequence $X = X_1 X_2 \ldots X_n$
- Output: positions of partition separators

Partitioning Trees in the Streaming Model:
- Input Stream: depth-first-traversal of input tree

241223314122113312
- Output: IDs of root nodes of partitions (1, 3, 6)
XML Document is a Depth-First-Traversal

```xml
<?xml version="1.0"?>
<university name="Reykjavik University">
  <lab name="DATALAB"></lab>
  <lab name="CADIA"></lab>
  <lab name="ICLT"></lab>
  <lab name="ICE-TCS">
    <members>
      <professor>Luca Aceto</professor>
      <professor>Magnus Halldorsson</professor>
      <postdoc>Ignacio Fabregas</postdoc>
      <phd-student>Christian Bean</phd-student>
    </members>
    <location>3rd floor, Mars</location>
  </lab>
  <lab name="ICE-ROSE"></lab>
</university>
```

Depth-first traversal:
- Opening tag $x$: down-step
- Closing tag $\overline{x}$: up-step
XML Document is a Depth-First-Traversal

```xml
<?xml version="1.0"?>
<university name="Reykjavik University">
  <lab name="DATALAB"></lab>
  <lab name="CADIA"></lab>
  <lab name="ICLT"></lab>
  <lab name="ICE-TCS">
    <members>
      <professor>Luca Aceto</professor>
      <professor>Magnus Halldorsson</professor>
      <postdoc>Ignacio Fabregas</postdoc>
      <phd-student>Christian Bean</phd-student>
    </members>
    <location>3rd floor, Mars</location>
  </lab>
</university>
```

Depth-first traversal:
- Opening tag $\times$: down-step
- Closing tag $\bar{\times}$: up-step
Previous Work
Previous Work: Partitioning Integer Sequences

Dynamic Programming:

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bokhari</td>
<td>1988</td>
<td>$O(n^3 p)$</td>
</tr>
<tr>
<td>Anily &amp; Federgruen</td>
<td>1991</td>
<td>$O(n^2 p)$</td>
</tr>
<tr>
<td>Hansen &amp; Liu</td>
<td>1992</td>
<td>$O(n^2 p)$</td>
</tr>
</tbody>
</table>
Previous Work: Partitioning Integer Sequences

**Dynamic Programming:**

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bokhari</td>
<td>1988</td>
<td>$O(n^3p)$</td>
</tr>
<tr>
<td>Anily &amp; Federgruen</td>
<td>1991</td>
<td>$O(n^2p)$</td>
</tr>
<tr>
<td>Hansen &amp; Liu</td>
<td>1992</td>
<td>$O(n^2p)$</td>
</tr>
</tbody>
</table>

**Iterative Improvement:**

<table>
<thead>
<tr>
<th>Researcher</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manne &amp; Sorevik</td>
<td>1995</td>
<td>$O(np \log p)$</td>
</tr>
<tr>
<td>Olstadt &amp; Manne</td>
<td>1995</td>
<td>$O(np)$</td>
</tr>
</tbody>
</table>
Previous Work: Partitioning Integer Sequences

Dynamic Programming:

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bokhari</td>
<td>1988</td>
<td>$O(n^3p)$</td>
</tr>
<tr>
<td>Anily &amp; Federgruen</td>
<td>1991</td>
<td>$O(n^2p)$</td>
</tr>
<tr>
<td>Hansen &amp; Liu</td>
<td>1992</td>
<td>$O(n^2p)$</td>
</tr>
</tbody>
</table>

Iterative Improvement:

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manne &amp; Sorevik</td>
<td>1995</td>
<td>$O(np \log p)$</td>
</tr>
<tr>
<td>Olstadt &amp; Manne</td>
<td>1995</td>
<td>$O(np)$</td>
</tr>
</tbody>
</table>

Other Results:

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nicol</td>
<td>1991</td>
<td>$O(n + p^2 \log^2 n)$</td>
</tr>
<tr>
<td>Charikar, Chekuri &amp; Motwani</td>
<td>1996</td>
<td>$O(n + p^2 \log^2 n)$</td>
</tr>
<tr>
<td>Han, Narahari &amp; Choi</td>
<td>1992</td>
<td>$O(n + p^{1+\epsilon})$, for any $\epsilon &gt; 0$</td>
</tr>
</tbody>
</table>
Previous Work: Partitioning Integer Sequences

**Dynamic Programming:**

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bokhari</td>
<td>1988</td>
<td>$O(n^3p)$</td>
</tr>
<tr>
<td>Anily &amp; Federgruen</td>
<td>1991</td>
<td>$O(n^2p)$</td>
</tr>
<tr>
<td>Hansen &amp; Liu</td>
<td>1992</td>
<td>$O(n^2p)$</td>
</tr>
</tbody>
</table>

**Iterative Improvement:**

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manne &amp; Sorevik</td>
<td>1995</td>
<td>$O(np \log p)$</td>
</tr>
<tr>
<td>Olstadt &amp; Manne</td>
<td>1995</td>
<td>$O(np)$</td>
</tr>
</tbody>
</table>

**Other Results:**

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nicol</td>
<td>1991</td>
<td>$O(n + p^2 \log^2 n)$</td>
</tr>
<tr>
<td>Charikar, Chekuri &amp; Motwani</td>
<td>1996</td>
<td>$O(n + p^2 \log^2 n)$</td>
</tr>
<tr>
<td>Han, Narahari &amp; Choi</td>
<td>1992</td>
<td>$O(n + p^{1+\epsilon})$, for any $\epsilon &gt; 0$</td>
</tr>
</tbody>
</table>

**Approach based on the Probe Algorithm:**

<table>
<thead>
<tr>
<th>Author</th>
<th>Year</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frederickson</td>
<td>1991</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
**Probe Algorithm**

**Probe**($B$):
- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:** $p = 3$, $\sum_i X_i = 92$, try **Probe**($31$)

```
5 6 11 2 9 14 3 8 1 11 22
```
**Probe Algorithm**

**Probe**($B$):

- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return `true` if successful, otherwise `false`

**Example:** $p = 3$, $\sum_i X_i = 92$, try **Probe**($31$)

```
5 6 11 2 9 14 3 8 1 11 22
```

5
**Probe Algorithm**

\[ \text{Probe}(B): \]
- Traverse \( X \) from left-to-right setting up maximal partitions so that partition weights do not exceed \( B \)
- Return \texttt{true} if successful, otherwise \texttt{false}

**Example:**  \( p = 3, \sum X_i = 92 \), try \texttt{Probe(31)}

\[
5 \quad 6 \quad 11 \quad 2 \quad 9 \quad 14 \quad 3 \quad 8 \quad 1 \quad 11 \quad 22
\]

11
**Probe Algorithm**

**Probe**(B):
- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:** $p = 3$, $\sum_i X_i = 92$, try $\text{Probe}(31)$

```
5  6  11  2  9  14  3  8  1  11  22
```

22
**Probe Algorithm**

**Probe**($B$):

- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:** $p = 3$, $\sum_i X_i = 92$, try **Probe**($31$)

5 6 11 2 9 14 3 8 1 11 22

24
**Probe Algorithm**

**Probe**($B$):
- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:** $p = 3$, $\sum_i X_i = 92$, try **Probe**(31)

```
5    6    11   2   |   9   14   3   8   1   11   22
```

$24 + 9 = 33 > 31$
\textbf{Probe Algorithm}

\texttt{Probe}(B):

- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return \texttt{true} if successful, otherwise \texttt{false}

\textbf{Example}: \quad p = 3, \sum_i X_i = 92, \text{ try } \texttt{Probe}(31)

\begin{align*}
5 & \quad 6 & \quad 11 & \quad 2 & | & \quad 9 & \quad 14 & \quad 3 & \quad 8 & \quad 1 & \quad 11 & \quad 22 \\
\end{align*}

9
**Probe Algorithm**

\textbf{Probe}(B):

- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return \texttt{true} if successful, otherwise \texttt{false}

\textbf{Example:} $p = 3$, $\sum_i X_i = 92$, try \texttt{Probe(31)}

\[
\begin{array}{cccc}
  5 & 6 & 11 & 2 \\
  9 & 14 & 3 & 8 & 1 & 11 & 22
\end{array}
\]

23
**Probe Algorithm**

**Probe**($B$):

- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:** $p = 3$, $\sum_i X_i = 92$, try $\text{Probe}(31)$

\[
\begin{array}{cccc|cccccc}
5 & 6 & 11 & 2 & 9 & 14 & 3 & 8 & 1 & 11 & 22
\end{array}
\]

26
**Probe Algorithm**

\textbf{Probe}(B):

- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return \texttt{true} if successful, otherwise \texttt{false}

\textbf{Example:} $p = 3$, $\sum_i X_i = 92$, try \texttt{Probe(31)}

\begin{array}{|c|c|c|c|c|}
\hline
5 & 6 & 11 & 2 & \hline
\hline
9 & 14 & 3 & \hline
\hline
8 & 1 & 11 & 22 & \hline
\end{array}

$26 + 8 = 34 > 31$
**Probe Algorithm**

**Probe**($B$):
- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:** $p = 3$, $\sum_i X_i = 92$, try **Probe**($31$)

```
  5  6  11  2  |  9  14  3  |  8  1  11  22
```

8
**Probe Algorithm**

**Probe**($B$):
- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return *true* if successful, otherwise *false*

**Example:**  $p = 3$, $\sum_i X_i = 92$, try $\text{Probe}(31)$

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>11</th>
<th>2</th>
<th>9</th>
<th>14</th>
<th>3</th>
<th>8</th>
<th>1</th>
<th>11</th>
<th>22</th>
</tr>
</thead>
</table>

9
**Probe Algorithm**

\text{Probe}(B): 
- Traverse \( X \) from left-to-right setting up maximal partitions so that partition weights do not exceed \( B \) 
- Return true if successful, otherwise false

**Example:** \( p = 3, \sum_i X_i = 92 \), try \text{Probe}(31)

\[
\begin{array}{cccc|ccc|cccc}
5 & 6 & 11 & 2 & 9 & 14 & 3 & 8 & 1 & 11 & 22 \\
\end{array}
\]
**Probe Algorithm**

**Probe**($B$):
- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:** $p = 3$, $\sum_i X_i = 92$, try **Probe**($31$)

\[
\begin{array}{cccc|cccc}
5 & 6 & 11 & 2 & 9 & 14 & 3 & 8 & 1 & 11 & 22 \\
\end{array}
\]

$20 + 22 = 42 > 31 \rightarrow \text{return false.}$

Last partition larger than 31 $\rightarrow$ optimal bottleneck $B^* \geq 32$
**Probe Algorithm**

**Probe**($B$):
- Traverse $X$ from left-to-right setting up maximal partitions so that partition weights do not exceed $B$
- Return true if successful, otherwise false

**Example:**  $p = 3$, $\sum_i X_i = 92$, try **Probe**(31)

$$
\begin{array}{cccccccc}
5 & 6 & 11 & 2 & | & 9 & 14 & 3 & | & 8 & 1 & 11 & 22 \\
\end{array}
$$

$20 + 22 = 42 > 31 \rightarrow \text{return false.}$

Trivial Bounds on $B^*$: ($m = \max X_i$)

$$1 \leq B^* \leq nm$$

Binary search: log $mn$ calls to **Probe** $\rightarrow O(n \log(mn))$ algorithm
Streaming Algorithms for Partitioning Integer Sequences
Baseline Algorithm

Observation:
\textsc{Probe} is a one-pass streaming alg. with $O(p \log n + \log(mn))$ space

One-pass Streaming Algorithm using \textsc{Probe}
- Suppose $m, n$ are known in advance
- Then optimal bottleneck value $B^*$ is bounded: $1 \leq B^* \leq mn$
- Run \textsc{Probe}(B) for $B = 1, (1 + \epsilon), (1 + \epsilon)^2, \ldots, mn$ in parallel

\[X_1X_2X_3X_4 \ldots X_{n-1}X_n\]

\textbf{Return:} Partitioning with smallest feasible value

$\rightarrow (1 + \epsilon)$-approximation using $\Theta(\log(mn)/\epsilon)$ copies of \textsc{Probe}
Baseline Algorithm

**Observation:**
\( \text{PROBE} \) is a one-pass streaming alg. with \( O(p \log n + \log(mn)) \) space

**One-pass Streaming Algorithm using Probe**

- Suppose \( m,n \) are known in advance
- Then optimal bottleneck value \( B^* \) is bounded: \( 1 \leq B^* \leq mn \)
- Run \( \text{PROBE}(B) \) for \( B = 1, (1 + \epsilon), (1 + \epsilon)^2, \ldots, mn \) in parallel

\[ X_1X_2X_3X_4 \ldots X_{n-1}X_n \]

**Return:** Partitioning with smallest feasible value

\( \rightarrow (1 + \epsilon) \)-approximation using \( \Theta(\log(mn)/\epsilon) \) copies of \( \text{PROBE} \)

\( \Theta(\log p/\epsilon) \)
Our Results

**Algorithm:** One-pass \((1 + \epsilon)\)-approximation streaming algorithm with

1. \(O(\log(mn)p/\epsilon)\) space,
2. Optimal \(O(1)\) update-time.

**Lower Bounds:**

- \(\Omega(n)\) is needed for exact algorithms
- \(\Omega(\frac{1}{\epsilon} \log n)\) is needed for \((1 + \epsilon)\)-approximation
New Technique: Coarsening

**Technique in Computer Science:**
Replace large (complicated) object by smaller (simpler) objects that capture important properties of initial object sufficiently well

E.g. Kernelization, Distance Oracles, Graph Sparsification, ...
Technique in Computer Science:
Replace large (complicated) object by smaller (simpler) objects that capture important properties of initial object sufficiently well.

E.g. Kernelization, Distance Oracles, Graph Sparsification, ...

Partitioning Sequences: Coarse Version

\[
\begin{array}{ccccccccccc}
4 & 3 & 2 & 8 & 7 & 2 & 1 & 7 & 7 & 8 & 5 & 2 & 3 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
\ & \ & \ & \\
\end{array}
\]
**New Technique: Coarsening**

**Technique in Computer Science:**
Replace large (complicated) object by smaller (simpler) objects that capture important properties of initial object sufficiently well

E.g. Kernelization, Distance Oracles, Graph Sparsification, . . .

**Partitioning Sequences: Coarse Version**

4 3 2 8 7 2 1 7 7 8 5 2 3 1

↓

9 15 10 15 11

- Compute coarse version of smaller size
New Technique: Coarsening

**Technique in Computer Science:**
Replace large (complicated) object by smaller (simpler) objects that capture important properties of initial object sufficiently well

E.g. Kernelization, Distance Oracles, Graph Sparsification, ...

**Partitioning Sequences:** Coarse Version

1. Compute coarse version of smaller size
2. Partition coarse version exactly \((p = 2)\)
3. Deduce partitioning of original version \((B = 34, B^* = 33)\)
Coarse Versions

**Definition: c-coarse Version**

\[ X: \begin{array}{cccccccccc}
4 & 3 & 2 & 8 & 7 & 2 & 1 & 7 & 7 & 8 & 5 & 2 & 3 & 1
\end{array} \]

\[ Y: \begin{array}{cccccc}
9 & 15 & 10 & 15 & 11
\end{array} \]

\[ (4, 5) \quad (8, 7) \quad (2, 9) \quad (7, 8) \quad (5, 6) \]

(base, increment)

- Split elements of coarse version \( Y \) into base and increment
- \( c \)-coarse version → maximal increment at most \( c \) (here: 9-coarse)

**Lemma:** Let \( B' \) be bottleneck value of opt. partitioning of \( c \)-coarse version \( Y \). Then opt. partitioning of \( X \) has bottleneck value \( B^* + c \geq B' \).

- \( \frac{S\epsilon}{p} \)-coarse version suffices, \( S = \sum_i X_i \) total weight), since \( B^* \geq S/p \)
- Length of coarse version: \( O(p/\epsilon) \) independent of \( n \)!
Example: \( p = 2, \epsilon = 1/2 \) (i.e., compute a 1.5-approximation)

\[
S: \quad 4 \quad 3 \quad 2 \quad 8 \quad 7 \quad 2 \quad 1 \quad 7 \quad 7 \quad 8 \quad 5 \quad 2 \quad 3
\]

Algorithm:
1. Fill memory with items from stream
2. Compress into \( \frac{S\epsilon}{p} \)-coarse version and repeat

Mem: _______ _______ _______ _______ _______ _______ _______
Example: \( p = 2, \epsilon = 1/2 \) (i.e., compute a 1.5-approximation)

\[
S: \quad 4 \quad 3 \quad 2 \quad 8 \quad 7 \quad 2 \quad 1 \quad 7 \quad 7 \quad 8 \quad 5 \quad 2 \quad 3
\]

Algorithm:
1. Fill memory with items from stream
2. Compress into \( \frac{S_\epsilon}{p} \)-coarse version and repeat

Mem: \[
(4, 0) \quad (3, 0) \quad (2, 0) \quad (8, 0) \quad (7, 0) \quad (2, 0) \quad (1, 0)
\]

\[
\frac{S_\epsilon}{p} = \frac{27 \cdot \frac{1}{2}}{2} = 6.75
\]
Example:  \( p = 2, \epsilon = 1/2\) (i.e., compute a 1.5-approximation)

\[
S: \begin{array}{cccccccccccc}
4 & 3 & 2 & 8 & 7 & 2 & 1 & 7 & 7 & 8 & 5 & 2 & 3
\end{array}
\]

Algorithm:

1. Fill memory with items from stream
2. Compress into \( \frac{S \epsilon}{p}\)-coarse version and repeat

\[
\text{Mem:} \begin{array}{cccc}
(4, 5) & & (8, 0) & (7, 3)
\end{array}
\]

\[
\frac{S \epsilon}{p} = \frac{27 \cdot \frac{1}{2}}{2} = 6.75
\]
**Example:** \( p = 2, \epsilon = 1/2 \) (i.e., compute a 1.5-approximation)

\[
S: \quad 4 \quad 3 \quad 2 \quad 8 \quad 7 \quad 2 \quad 1 \quad 7 \quad 7 \quad 8 \quad 5 \quad 2 \quad 3
\]

**Algorithm:**

1. Fill memory with items from stream
2. Compress into \( \frac{S_\epsilon}{p} \)-coarse version and repeat

\[
\text{Mem:} \quad (4, 5) \quad (8, 0) \quad (7, 3)
\]
Example: \( p = 2, \epsilon = 1/2 \) (i.e., compute a 1.5-approximation)

\[
S: \begin{array}{cccccccccccc}
4 & 3 & 2 & 8 & 7 & 2 & 1 & 7 & 7 & 8 & 5 & 2 & 3
\end{array}
\]

Algorithm:

1. Fill memory with items from stream
2. Compress into \( \frac{S\epsilon}{p} \)-coarse version and repeat

Mem: \[
\begin{array}{cccccccc}
(4, 5) & (8, 0) & (7, 3) & (7, 0) & (7, 0) & (8, 0) & (5, 0)
\end{array}
\]

\[
\frac{S\epsilon}{p} = \frac{54 \cdot \frac{1}{2}}{2} = 13.5
\]
Example: \( p = 2, \epsilon = 1/2 \) (i.e., compute a 1.5-approximation)

\[
S: \quad 4 \quad 3 \quad 2 \quad 8 \quad 7 \quad 2 \quad 1 \quad 7 \quad 7 \quad 8 \quad 5 \quad 2 \quad 3
\]

Algorithm:

1. Fill memory with items from stream
2. Compress into \( \frac{S\epsilon}{p} \)-coarse version and repeat

\[
\text{Mem:} \quad (4, 13) \quad (7, 10) \quad (7, 13)
\]

\[
\frac{S\epsilon}{p} = \frac{54 \cdot \frac{1}{2}}{2} = 13.5
\]
Example: $p = 2, \epsilon = 1/2$ (i.e., compute a 1.5-approximation)

$S$: 4 3 2 8 7 2 1 7 7 8 5 2 3

Algorithm:
1. Fill memory with items from stream
2. Compress into $\frac{S\epsilon}{p}$-coarse version and repeat

Mem: (4, 13) (7, 10) (7, 13)
Example: \( p = 2, \epsilon = 1/2 \) (i.e., compute a 1.5-approximation)

\[
S: \quad 4 \quad 3 \quad 2 \quad 8 \quad 7 \quad 2 \quad 1 \quad 7 \quad 7 \quad 8 \quad 5 \quad 2 \quad 3
\]

Algorithm:

1. Fill memory with items from stream
2. Compress into \( \frac{S\epsilon}{p} \)-coarse version and repeat

Mem: 

\[
\begin{array}{cccccc}
(4, 13) & (7, 10) & (7, 13) & (2, 0) & (3, 0)
\end{array}
\]

\[
\frac{S\epsilon}{p} = \frac{59 \cdot 1/2}{2} = 14.75
\]
Example: \( p = 2, \epsilon = 1/2 \) (i.e., compute a 1.5-approximation)

\[
S: \quad 4 \quad 3 \quad 2 \quad 8 \quad 7 \quad 2 \quad 1 \quad 7 \quad 7 \quad 8 \quad 5 \quad 2 \quad 3
\]

Algorithm:

1. Fill memory with items from stream
2. Compress into \( \frac{S\epsilon}{p} \)-coarse version and repeat

Mem: \[
(4,13) \quad (7,10) \quad (7, 13) \quad (2, 3)
\]

- Coarse version: 17 17 20 5
- Bottleneck value of resulting partitioning: \( B = 34 \)
- Optimal bottleneck value: \( B^* = 32 \)
**Ω(n) Lower Bound for Exact Algorithms**

**Hard Communication Problem:** *Index Problem*

\[
\begin{align*}
\text{Alice} & \quad \xrightarrow{M} \quad \text{Bob} \quad \rightarrow S[I]
\end{align*}
\]

\[S \in \{0, 1\}^N\quad I \in \{1, 2, \ldots, N\}\]

**Fact:** \(|M| \in \Omega(n)\), for randomized protocols with bounded error

**Reduction:**
- Alice: \(S = 0, 1, 0, 0, 1\) generates \(X_1 = 13 31 13 13 31\)
- Bob: \(I = 4\) generates \(X_2 = 4 \ldots 4 \ 2 = 4 4 2\)
  \[2I - N - 1\]
- Optimal split of \(X_1 \circ X_2 : 13 31 13 13 \mid 31 4 4 2\), no perfect split
- If \(S[4] = 1\) then: \(13 31 13 3 \mid 1 31 4 4 2\), perfect split
Algorithm: Summary

**Algorithm**
- Compute \((S\epsilon/p)\)-coarse version of length \(O(p/\epsilon)\) in one pass
- **Post-processing:** Partition coarse version optimally and deduce \((1 + \epsilon)\)-partitioning of initial instance

**Properties of Algorithm**
- \(O(p \log(mn)/\epsilon)\) space
- Can be implemented with optimal \(O(1)\) update-time

**What is the correct space complexity?**
- \(\Omega(n)\) for exact algorithms
- \(\Omega(\log(n)/\epsilon)\) for \((1 + \epsilon)\)-approximations
Streaming Algorithms for Partitioning Trees
Coarse Version of Trees

**Structure Tree**
- Compute coarse structure tree consisting of $O\left(p^2/\epsilon\right)$ nodes
- Pick subset of breakpoint nodes $U = \{u_1, u_2, \ldots\}$ ordered w.r.t. a depth-first-traversal
- Let $L = \{\text{lca}(u_i, u_{i+1}) : i\}$ be the set of lowest-common-ancestors of consecutive breakpoints
- Structure tree built on nodes $L \cup U$

**Figure:** $U$: highlighted nodes. $L$: nodes within boxes.
Good Breakpoints

Breakpoints

- Compute coarse-version of sequence of down-steps $X'$ of depth-first-traversal $X$:

$$X' = 24123232121323$$

- 5-coarse version of $X'$:

$$77673$$

$$241|232|3212|132|3$$

- Bold elements define $U$

$\rightarrow$ Reduction to Sequences
Algorithm: Summary

Algorithm

- 2 passes required for computing structure tree
- **Post-processing:** Partition structure tree optimally and deduce $(1 + \epsilon)$-approximate partitioning

Properties of Algorithm

- $O(p^2 \log(mn)/\epsilon)$ space
- Two passes
- Can be implemented with optimal $O(1)$ update-time

Open Questions

- Can space be reduced to $O(p \log(mn)/\epsilon)$?
- One pass?
Conclusion

- Modern applications provide new perspectives on old problems
- New insight: Coarsening

Where to go from here?

- XML documents: Partitioning respecting underlying structure
- Leightweight streaming algorithms for other partitioning problems?
- Prove space optimality
Thank You for Listening.
Questions?