General Incremental Sliding-Window Aggregation

Kanat Tangwongsan

Mahidol University International College, Thailand
(part of this work done at IBM Research, Watson)

Joint work with Martin Hirzel, Scott Schneider, and Kun-Lung Wu
IBM Research, Watson
Dynamic data is everywhere

often interested in a **sliding window**

Compute some aggregation on this data
Build a system to...

Answer the following questions every minute about the past 24 hrs:

- How long was the longest call?
- How many calls have that duration?
- Who is a caller with that duration?
Answer the following questions every minute about the past 24 hrs:
• How long was the longest call?
• How many calls have that duration?
• Who is a caller with that duration?

Basic Solution:

Maintain a sliding window (past 24 hr)

Walk through the window every minute

Simple but slow: $O(n)$ per query
Improvement Opportunities

Idea: When window slides, lots of common contents with the most recent query.

How to Reuse?

If **invertible**, keep a running sum: add on arrival, subtract on departure

Partial sum: bundle items that arrive and depart together to reduce # of items in the window.
This Work:

How to engineer sliding-window aggregation so that

- can add new aggregation operations easily
- can get good performance with little hassle

(using a combination of simple, known ideas)
**Performance**

**Prior Work**

- Good for small updates
  (e.g., Arasu-Widom VLDB’04, Moon et al. ICDE’00)

- Good for large updates
  (e.g., Cranor et al. SIGMOD’03, Krishnamurthi et al. SIGMOD’06)

**This Work**

- Good for Large & Small:
  If $m$ updates are made to a window of size $n$, use $O(m \log(n/m))$ time to compute aggregate.

**Generality**

**Prior Work**

- Require invertibility or commutativity or assoc.

- Require FIFO windows

**This Work**

- Require associativity (not but invertibility nor commutativity)

**OK** to be non-FIFO
Our Approach

High-level Idea

User writes aggregation code following an interface

Data Structure Template

User declares query

Linked with

Window-Management Library

Aggregation Interface

map-assoc. reduce-map

$w_1 \rightarrow t_1$ \hspace{1cm} $w_2 \rightarrow t_2$ \hspace{1cm} $w_3 \rightarrow t_3$ \hspace{1cm} $\cdots$ \hspace{1cm} $w_n \rightarrow t_n$

lift

reduce using combine

lower

$a$ \rightarrow final answer
**Example**

\[
\text{StdDev} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (w_i - \bar{x})^2}
\]

- **lift** \( x \mapsto \{c : 1, \Sigma : x, \sigma : x^2\} \)
- **combine** component-wise add
- **lower** \( \sqrt{\frac{1}{c} (\sigma - \Sigma^2 / c)} \)

- count
- sum of values
- sum of squares
Perfect Binary Tree

Keep Partial Sum in Internal Nodes

Depth: $\log n$

Changing $k$ leaves affects $\leq O(k \log(n/k))$ nodes
Data Structure Engineering

User writes code following an interface

Data Structure Template

Minimize pointer chasing
Allocate memory in bulk
Place data to help cache

Window-Management Library
Main idea:
keep a perfect binary tree, treating leaves as a circular buffer, all laid out as one array

physical

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

binary heap encoding

logical

```
2
/|
4 5
/|
8 9 10 11 12 13 14 15
```

Q: Non power of 2? Dynamic window size? non-FIFO?

The queue’s front and back locations give a natural demarcation.

Resize and rebuild as necessary. Amortized bounds (see paper)
But…
Circular Buffer Leaves !≠ Window-Ordered Data

**Fix?** When inverted:

```
  i j B
prefix  e f g h
suffix
```

ans = combine(suffix, prefix)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaf view</td>
<td>F</td>
<td>1</td>
<td>B</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>1</td>
<td>2</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>B</td>
<td>F</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

13
Experimental Analysis

1. What’s the throughput relative to non-incremental?

2. What’s the performance trend under updates of different sizes?

3. How does wildly-changing window size affect the overall throughput?
What’s the throughput relative to non-incremental?

### Graph

- **X-axis**: Window Size
- **Y-axis**: Million Tuples/Second

**Legend**:
- Baseline: Recompute all
- ISS Max
- ISS StdDev
- RA Max
- RA StdDev
- Our scheme

**Notable Points**:
- Small windows: slow down ≤ 10%
- Crossover for StdDev (~ 10)
- Crossover for Max (~ 260)
- Break 10x as early as \( n \) in the thousands

**Questions**:
- Faster than the baseline at what window size?
What’s the performance trend under updates of different sizes?

Our Theory: Process $k$ events in $O(1 + \log (n/k))$ time per event.
How does wildly-changing window size affect the overall throughput?

![Graph showing the relationship between window size and average cost per tuple](image)

- RA StdDev (fixed)
- RA Max (fixed)
- RA StdDev (osc.)
- RA Max (osc.)

**faster**
Take-Home Points

This work: sliding-window aggregation

- easily extendible by users and has good performance
- careful systems + algorithmic design and engineering (blend of known ideas)
- general (non-FIFO, only need associativity) and fast for large & small windows

If \( m \) updates have been made on a window of size \( n \), use \( O(m \log(n/m)) \) time to derive aggregate.

**More details:** see paper/come to the poster session.