Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time

Kanat Tangwongsan
Mahidol University International College
kanat.tan@mahidol.edu

Martin Hirzel
IBM Research
hirzel@us.ibm.com

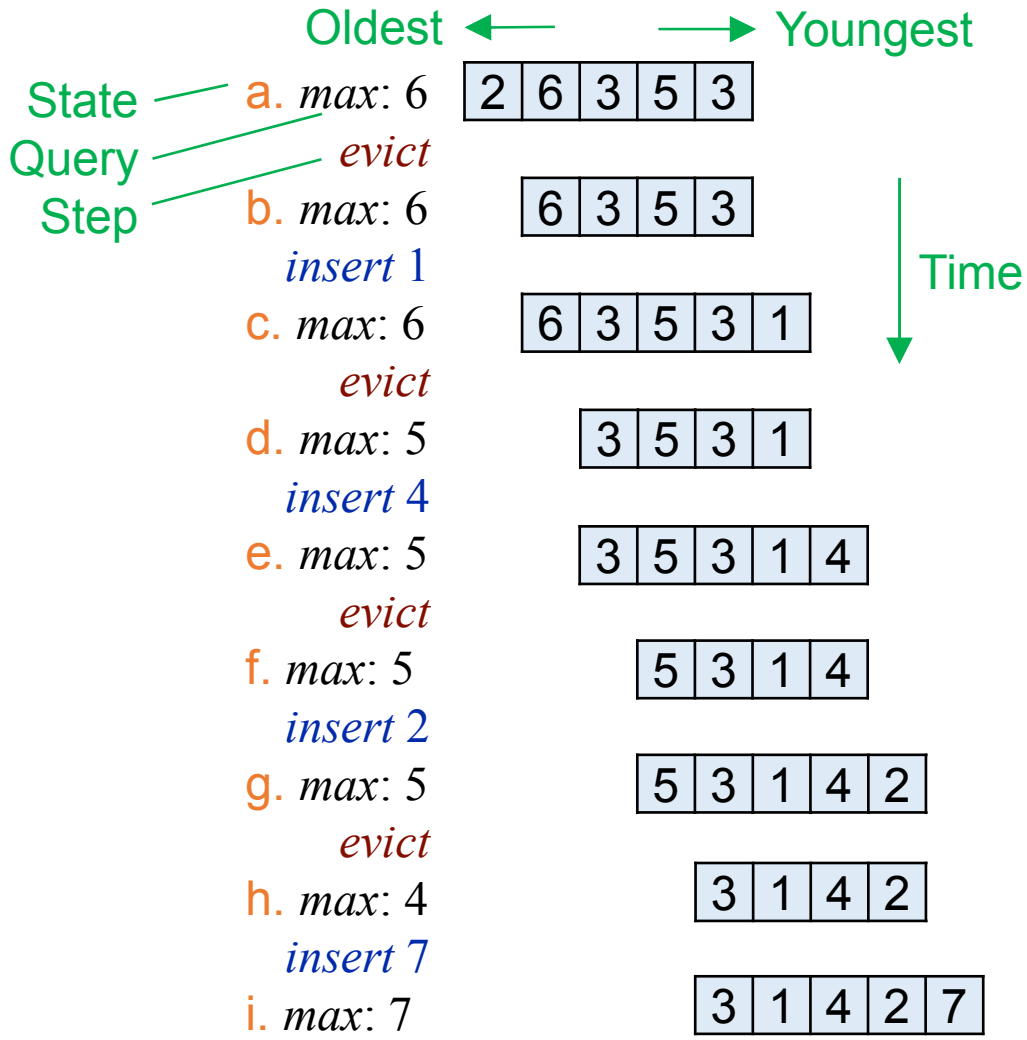
Scott Schneider
IBM Research
scott.a.s@us.ibm.com

DEBS 2017

Title Explained

What	Why
Low-Latency	QoS or Real-time
Sliding-Window	Recent $\cong$ relevant
Aggregation	Stream “reduce”
Worst-Case	No latency spikes
Constant Time	$O(1)$ better than $O(\log n)$

Running Example



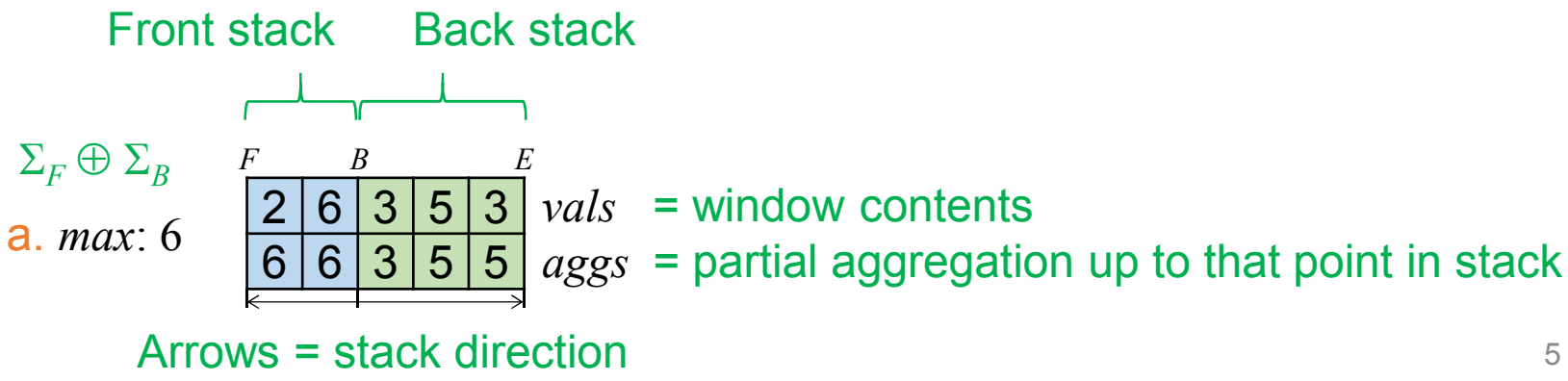
In general:

- Any associative aggregation operation (not just max ⇒ sum, geoMean, Bloom, …)
- Any interleaving of insert and evict (not just alternating ⇒ variable-sized windows)

Two-Stacks Algorithm

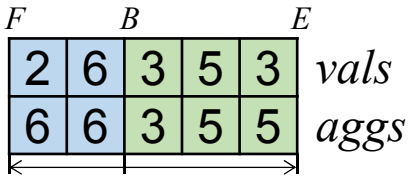
What	Why
Low-Latency	QoS or Real-Time
Sliding-Window	Recent $\cong$ relevant
Aggregation	Stream "reduce"
Amortized ~~Worst-Case~~	Some ~~No~~ latency spikes
Constant Time	$O(1)$ better than $O(\log n)$

Two-Stacks Example



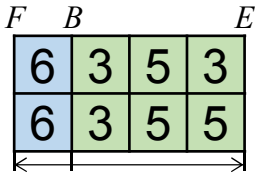
Two-Stacks Example

a. *max*: 6



evict

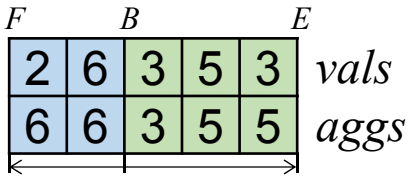
b. *max*: 6



F.pop()
// *O*(1) time

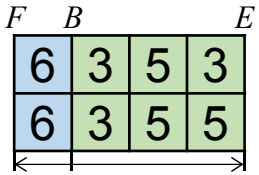
Two-Stacks Example

a. *max*: 6



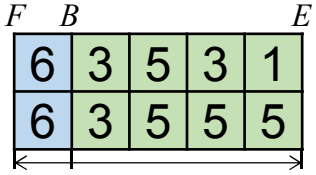
evict

b. *max*: 6



insert 1

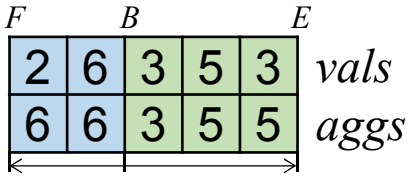
c. *max*: 6



$B.push(v, \Sigma_B \oplus v)$
// $O(1)$ time

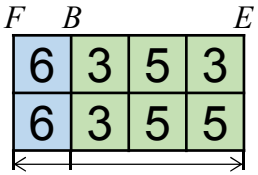
Two-Stacks Example

a. *max*: 6



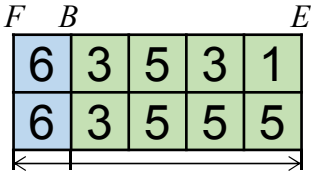
evict

b. *max*: 6



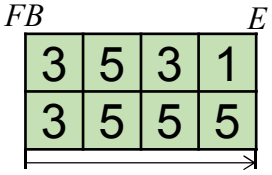
insert 1

c. *max*: 6



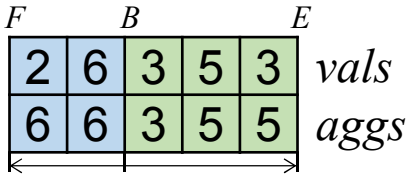
evict

d. *max*: 5



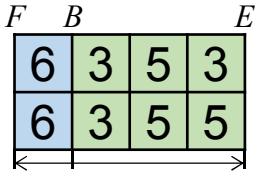
Two-Stacks Example

a. *max*: 6



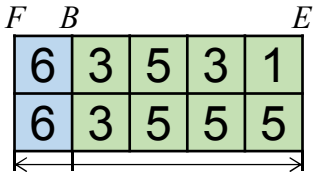
evict

b. *max*: 6



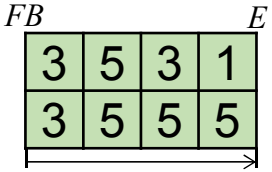
insert 1

c. *max*: 6



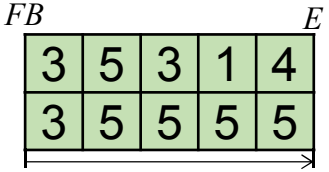
evict

d. *max*: 5



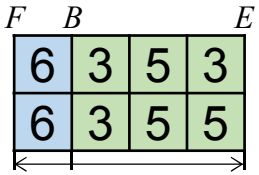
insert 4

e. *max*: 5



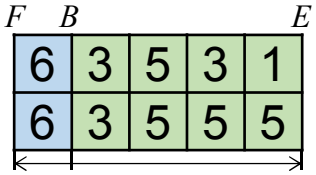
Two-Stacks Example

b. *max*: 6



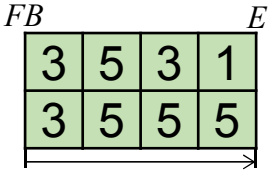
insert 1

c. *max*: 6



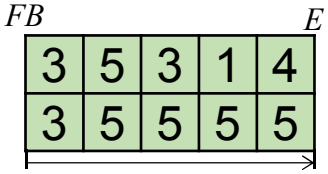
evict

d. *max*: 5



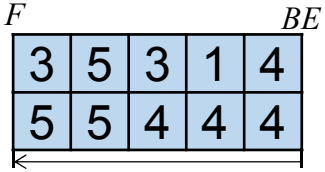
insert 4

e. *max*: 5



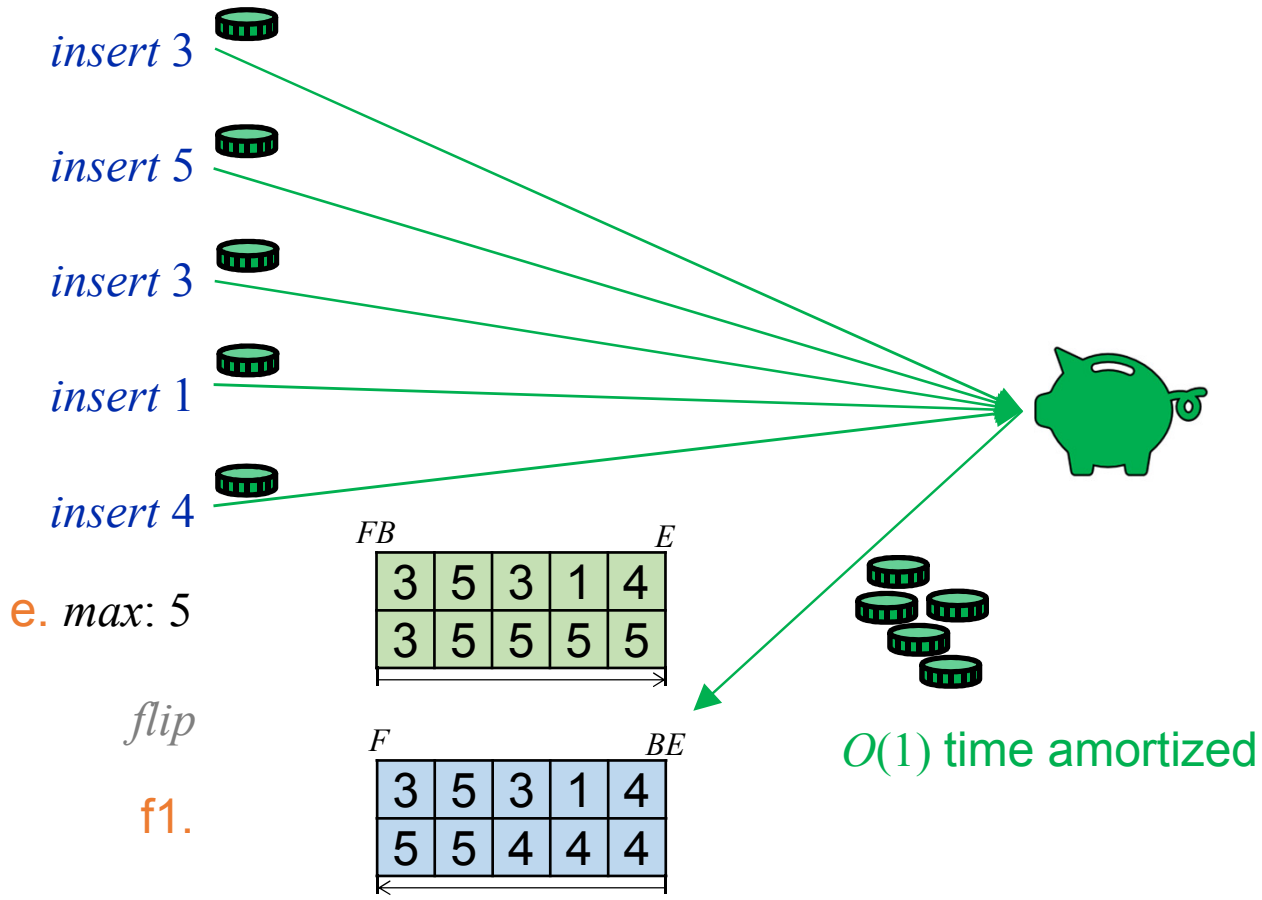
flip

f1.



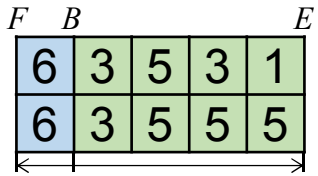

```
if F.isEmpty()
  while not B.isEmpty()
    F.push(B.top().val, B.top().val ⊕ Σ_F)
    B.pop()
// O(n) time, latency spike
```

Banker's Amortized Analysis

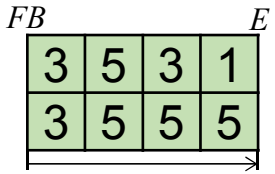


Two-Stacks Example

c. *max*: 6

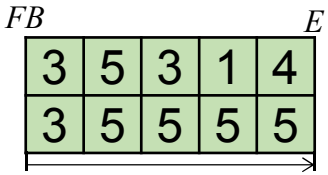


evict



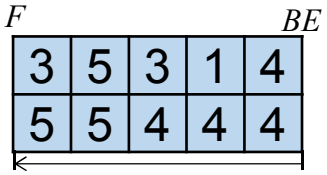
d. *max*: 5

insert 4



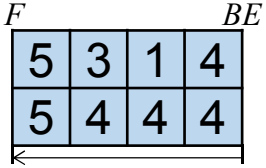
e. *max*: 5

flip



f1.

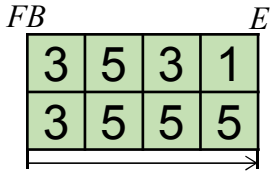
evict



f. *max*: 5

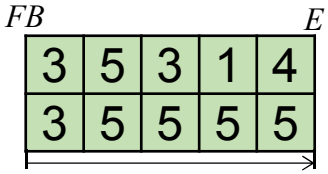
Two-Stacks Example

d. *max*: 5



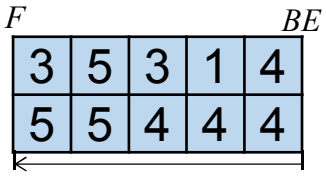
insert 4

e. *max*: 5



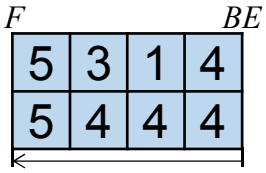
flip

f1.



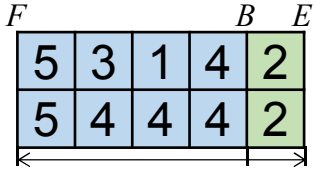
evict

f. *max*: 5



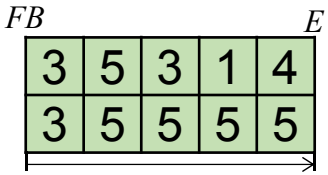
insert 2

g. *max*: 5



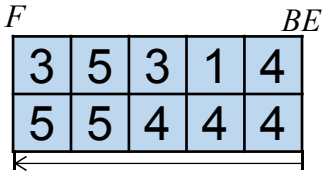
Two-Stacks Example

e. *max*: 5

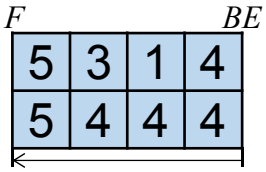


flip

f1.

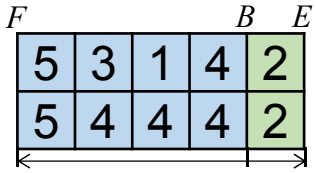


evict



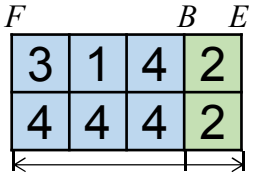
f. *max*: 5

insert 2



g. *max*: 5

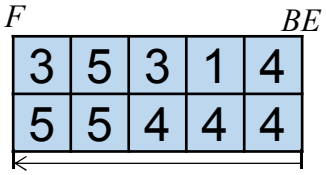
evict



h. *max*: 4

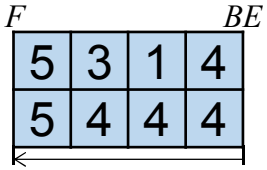
Two-Stacks Example

f1.



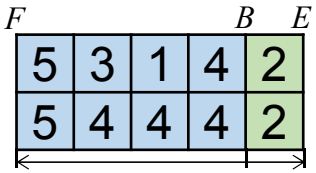
evict

f. *max*: 5



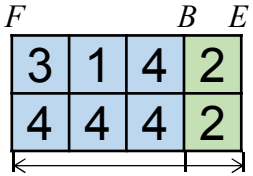
insert 2

g. *max*: 5



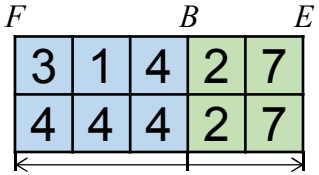
evict

h. *max*: 4



insert 7

i. *max*: 7



Beyond Two-Stacks

Two-Stacks is $O(1)$, but only amortized.

Want $O(1)$ worst-case to prevent latency spikes.

Beyond Two-Stacks

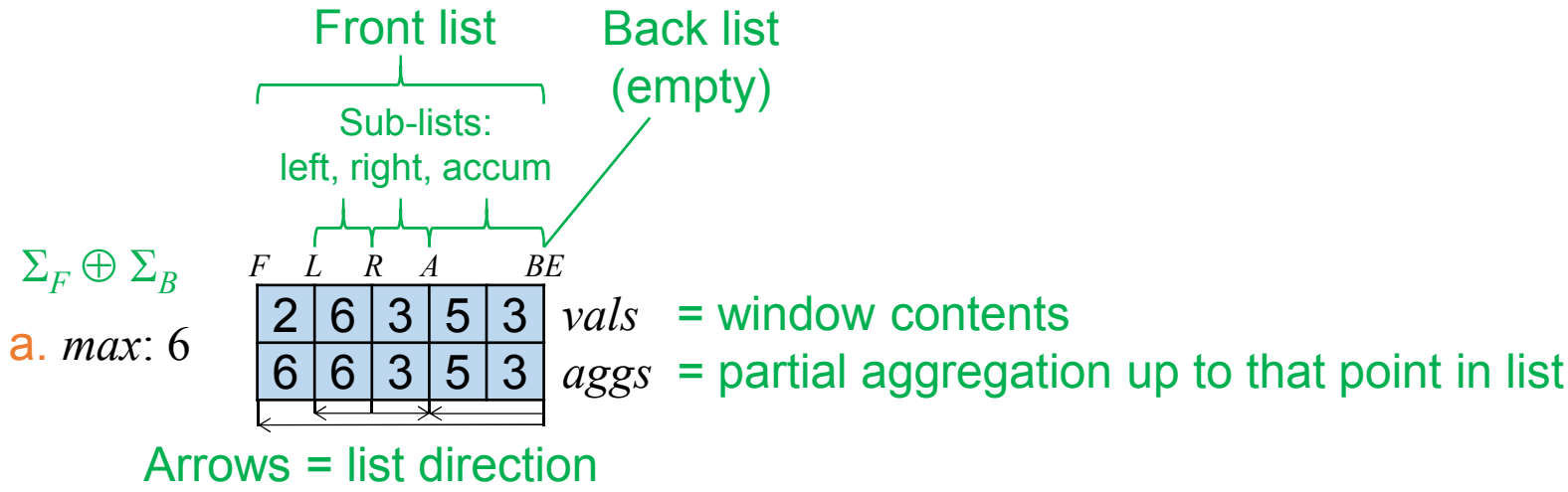
Two-Stacks is $O(1)$, but only amortized.

Want $O(1)$ worst-case to prevent latency spikes.

Okasaki [JFP'95]: FIFO queue in
$O(1)$ worst-case, but no aggregation.

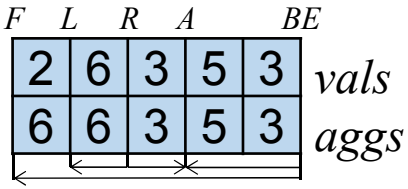
DABA (De-Amortized Banker's Aggregator)
inspired by Okasaki's, supports aggregation.

DABA Example

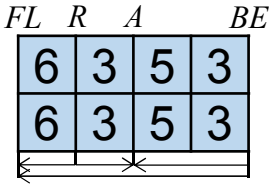


DABA Example

a. *max*: 6



evict

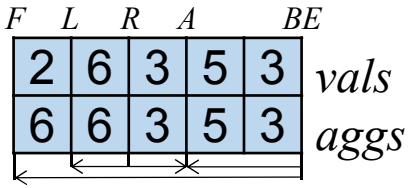


b1.

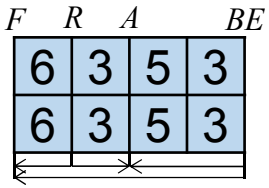
// evict
vals.popFront( )
aggs.popFront( )

DABA Example

a. *max*: 6

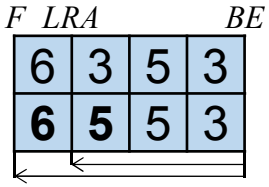


evict



b1.

shrink

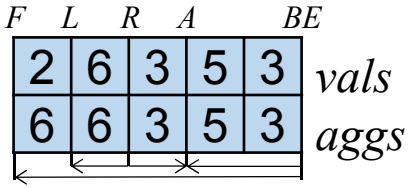


b. *max*: 6

// evict
vals.popFront( )
aggs.popFront( )
// fixup → shrink case
$aggs[L] \leftarrow \Sigma_L \oplus \Sigma_R \oplus \Sigma_A$
$L \leftarrow L + 1$
$aggs[A-1] \leftarrow vals[A-1] \oplus \Sigma_A$
$A \leftarrow A - 1$
// $O(1)$ time

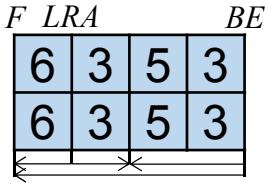
DABA Example

a. *max*: 6



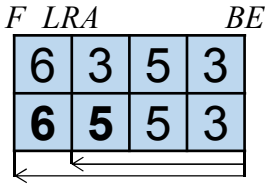
evict

b1.



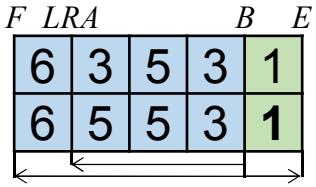
shrink

b. *max*: 6



insert 1

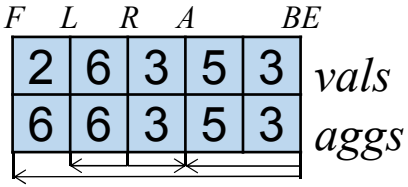
c1.



// insert
vals.pushBack(v)
aggs.pushBack($\Sigma_B \oplus v$)

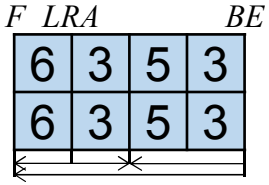
DABA Example

a. *max*: 6



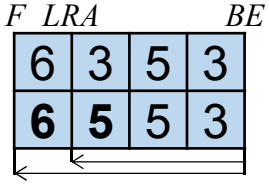
evict

b1.

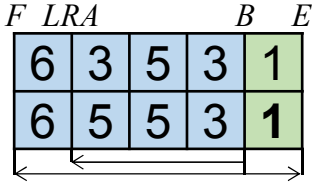


shrink

b. *max*: 6

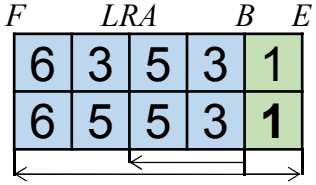


insert 1



c1.

shift

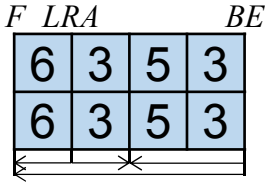


c. *max*: 6

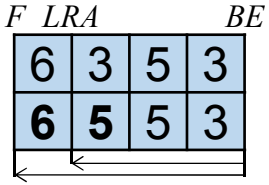
// insert
vals.pushBack(v)
aggs.pushBack($\Sigma_B \oplus v$)
// fixup → shift case
$A \leftarrow A + 1$
$R \leftarrow R + 1$
$L \leftarrow L + 1$
// $O(1)$ time

DABA Example

b1.

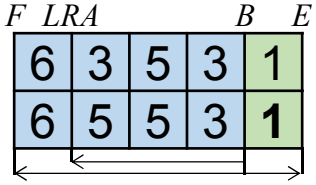


shrink



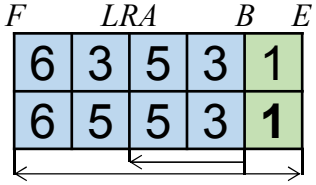
b. *max*: 6

insert 1



c1.

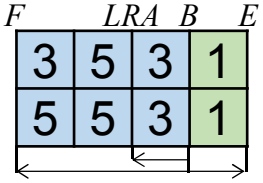
shift



c. *max*: 6

evict

shift



d. *max*: 5

DABA Example

b. *max*: 6

insert 1

c1.

shift

c. *max*: 6

evict

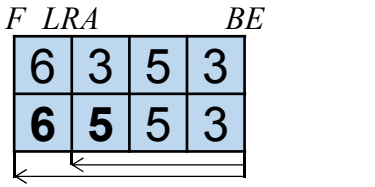
shift

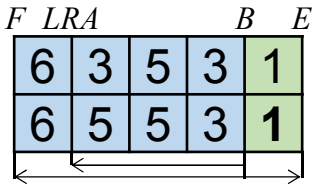
d. *max*: 5

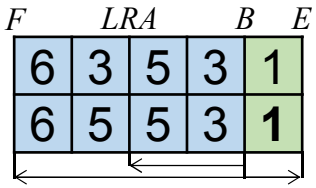
insert 4

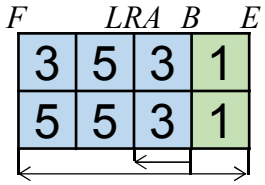
shift

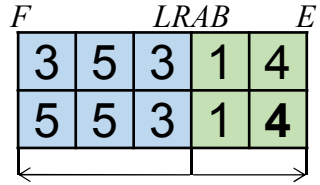
e. *max*: 5





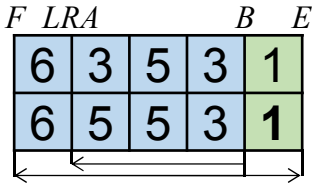




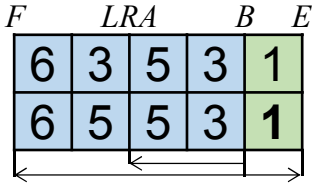


DABA Example

c1.



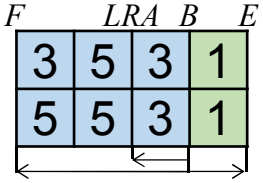
shift



c. *max*: 6

evict

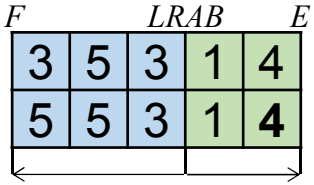
shift



d. *max*: 5

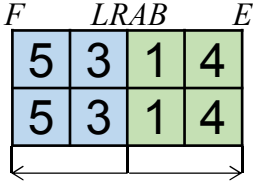
insert 4

shift



e. *max*: 5

evict



f1.

// evict
vals.popFront( )
aggs.popFront( )

DABA Example

c. *max*: 6

evict

shift

d. *max*: 5

insert 4

shift

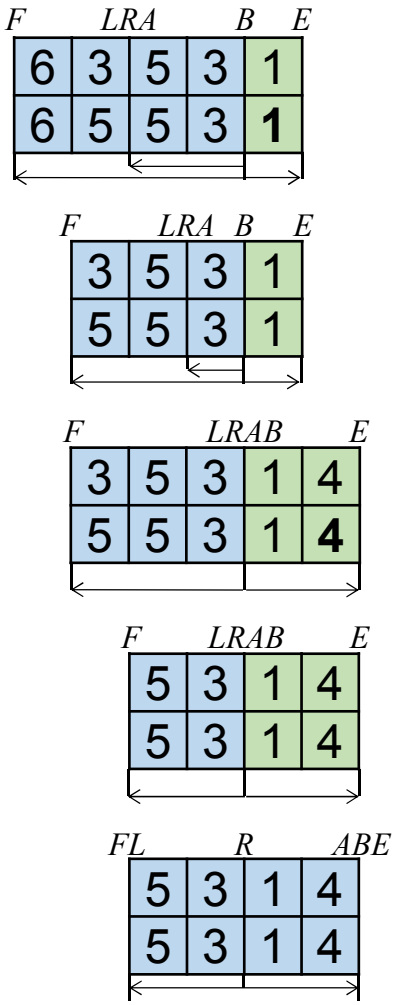
e. *max*: 5

evict

f1.

flip

f2.



// evict

vals.popFront( )

aggs.popFront( )

// fixup → flip case

L ← *F*

A ← *E*

B ← *E*

DABA Example

d. *max*: 5

insert 4

shift

e. *max*: 5

evict

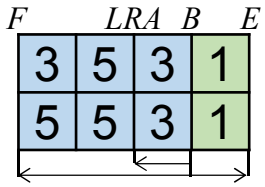
f1.

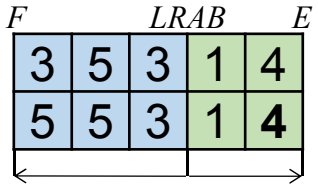
flip

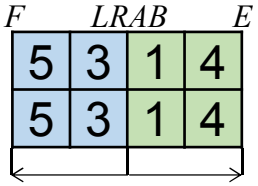
f2.

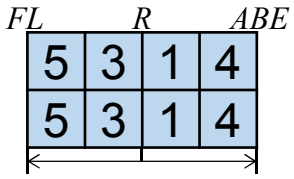
shrink

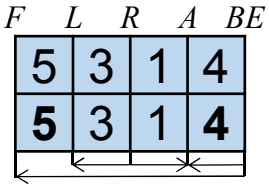
f. *max*: 5









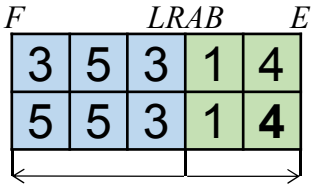


// evict
vals.popFront( )
aggs.popFront( )
// fixup → flip case
$L \leftarrow F$
$A \leftarrow E$
$B \leftarrow E$
// fixup → shrink case
$aggs[L] \leftarrow \Sigma_L \oplus \Sigma_R \oplus \Sigma_A$
$L \leftarrow L + 1$
$aggs[A-1] \leftarrow vals[A-1] \oplus \Sigma_A$
$A \leftarrow A - 1$
// $O(1)$ time

DABA Example

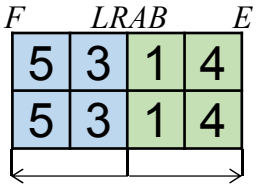
e. *max*: 5

evict



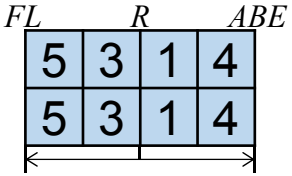
f1.

flip



f2.

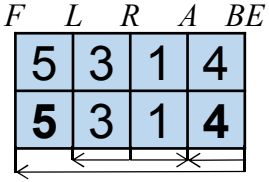
shrink



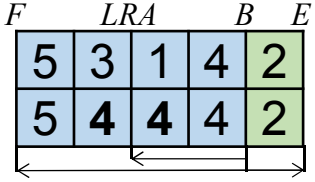
f. *max*: 5

insert 2

shrink



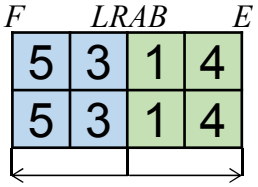
g. *max*: 5



DABA Example

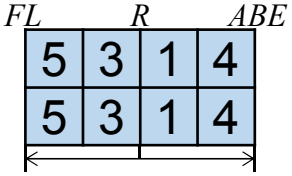
f1.

flip



f2.

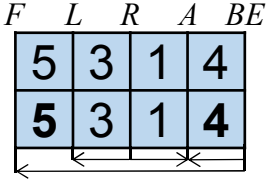
shrink



f. *max*: 5

insert 2

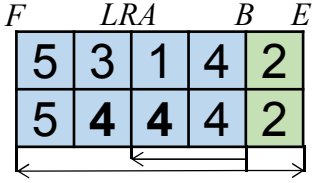
shrink

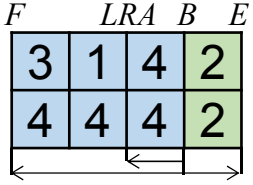


g. *max*: 5

evict

shift



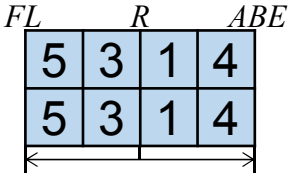


h. *max*: 4

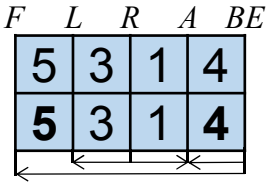
DABA Example

f2.

shrink



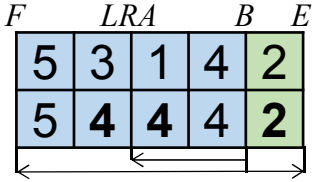
f. *max*: 5



insert 2

shrink

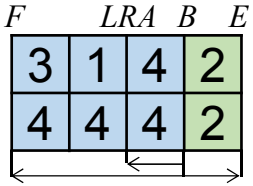
g. *max*: 5



evict

shift

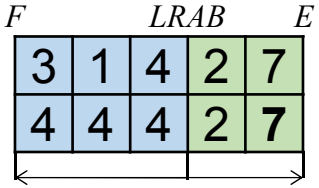
h. *max*: 4



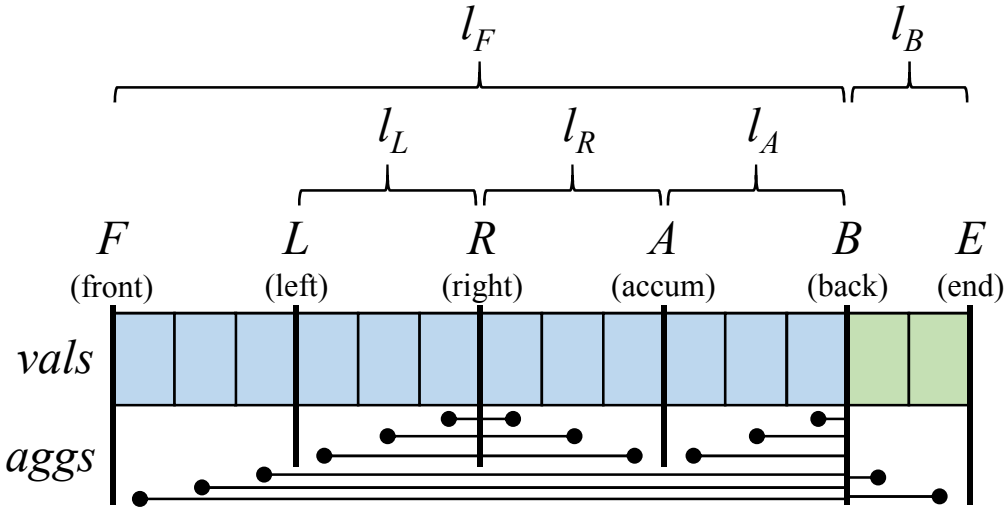
insert 7

shift

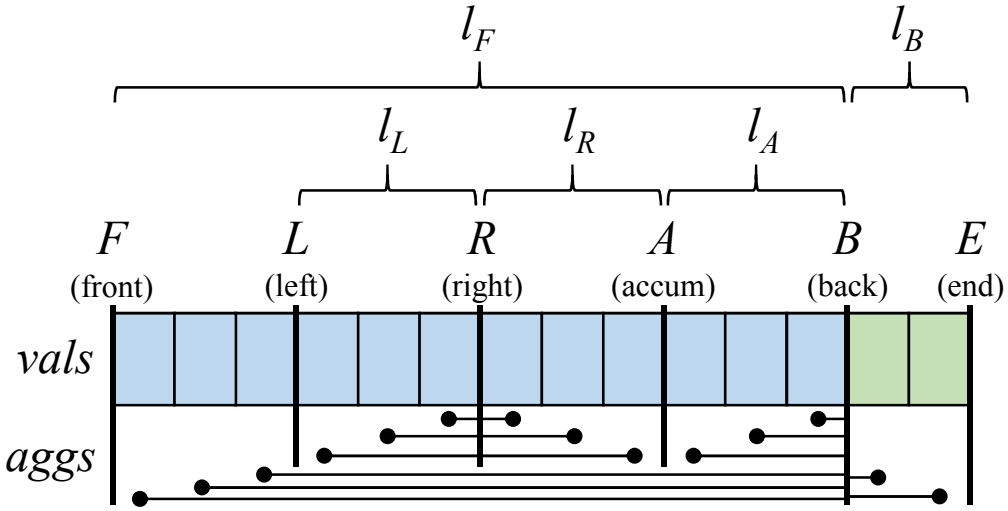
i. *max*: 7



Aggregation Invariants
(Answering queries correctly)



Size Invariants
(Getting work done on time)



$$\left(|l_F| = 0 \quad \textbf{\textit{and}} \quad |l_B| = 0\right)$$

or

$$\left(|l_L| = |l_R| \quad \textbf{\textit{and}} \quad |l_L| + |l_R| + |l_A| + 1 = |l_F| - |l_B|\right)$$

Related Work

Algorithm	Time	Space	Invertible	FIFO
Subtract on evict	worst $O(1)$	$O(n)$	needed	no
Recalculate from scratch	worst $O(n)$	$O(n)$	no	no
Tree-based (e.g. reactive)	amzd. $O(\log n)$	$O(n)$	no	no
Two-stacks	amzd. $O(1)$	$O(n)$	no	needed
DABA	worst $O(1)$	$O(n)$	no	needed

Results

Latencies [cycles], $n = 2^{14}$, average $\pm$ standard deviation

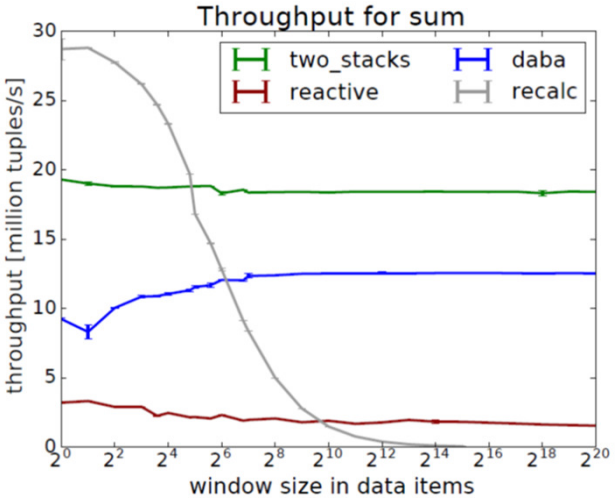
Operator	Two-Stacks	DABA
Sum	127 $\pm$ 2,980	193 $\pm$ 113
Max	125 $\pm$ 2,991	200 $\pm$ 122
GeoMean	299 $\pm$ 3,777	366 $\pm$ 168
Bloom	5,532 $\pm$ 298,982	9,021 $\pm$ 4,289

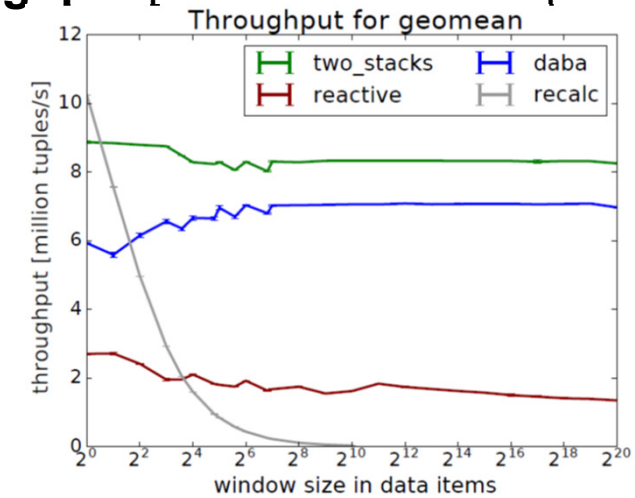
Results

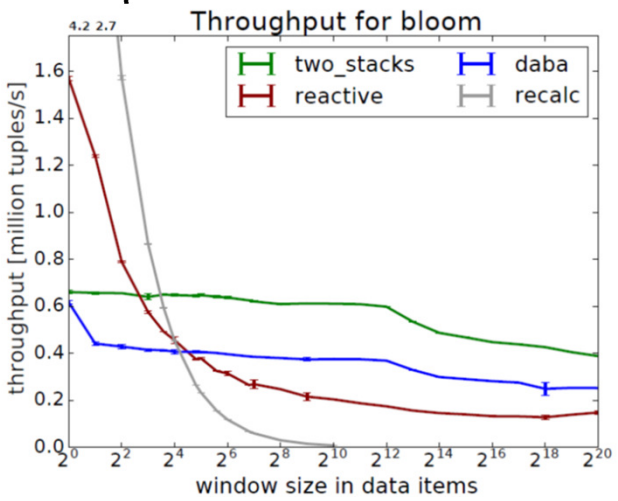
Latencies [cycles], $n = 2^{14}$, average $\pm$ standard deviation

Operator	Two-Stacks	DABA
Sum	127 $\pm$ 2,980	193 $\pm$ 113
Max	125 $\pm$ 2,991	200 $\pm$ 122
GeoMean	299 $\pm$ 3,777	366 $\pm$ 168
Bloom	5,532 $\pm$ 298,982	9,021 $\pm$ 4,289

Throughput [million data items per second]







Conclusions

DABA aggregates FIFO sliding windows in $O(1)$ worst-case time and $O(n)$ space

DABA works for any associative binary aggregation operator $\oplus$

Orthogonal optimizations (not in this talk): data parallelism; coarse-grained sliding