Physically Independent Stream Merging

Badrish Chandramouli David Maier Jonathan Goldstein

March 12, 2013

Presenters: Amol Bhangadiya,Pushkar Khadilkar

Overview

1. Introduction
   - Motivation
   - Challenges
2. Stream Formalism
   - Theoretical Treatment
   - Stream Elements
   - Logical Merge
3. Algorithms for LMerge
4. Evaluation

Motivation

- DSMS

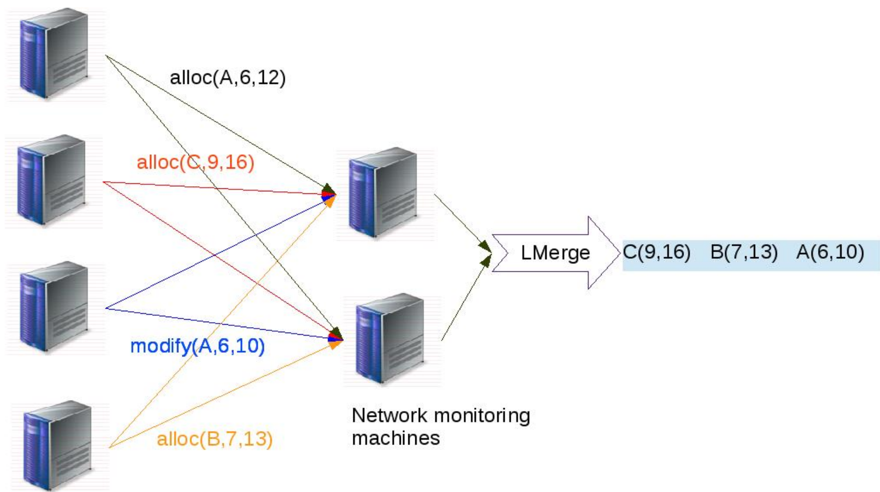
Motivation

- DSMS
- Continuous Queries
  - Recovery
  - Re-optimization
  - Load Balancing

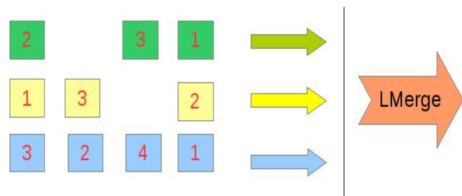
Motivation

- DSMS
- Continuous Queries
    - Recovery
    - Re-optimization
    - Load Balancing
- LMerge
    - Data Stream Operator
    - Combine Physically Different but Logically Equivalent Streams

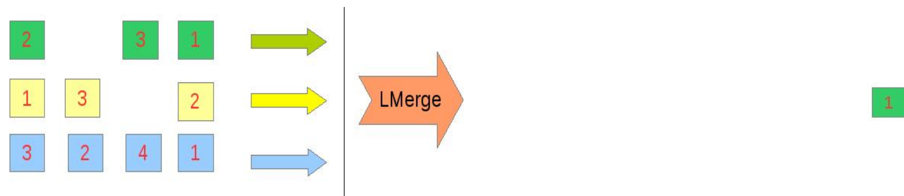
Network monitoring



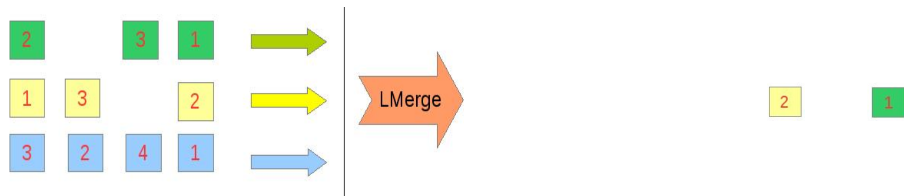
Motivation



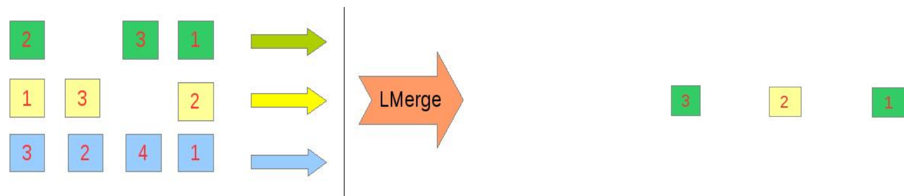
Motivation



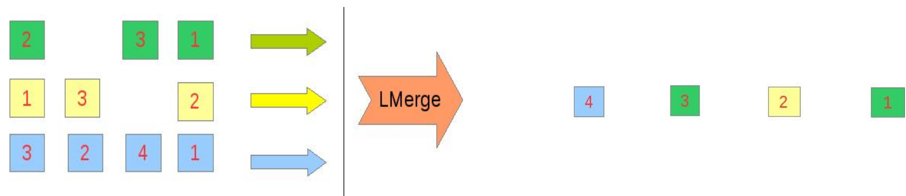
Motivation



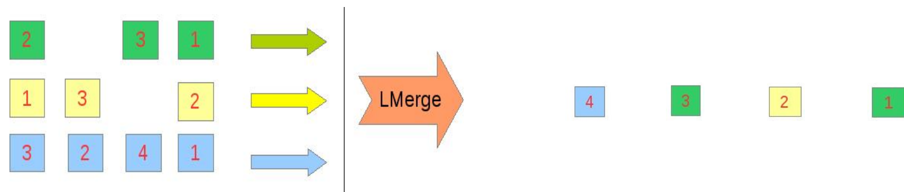
Motivation



Motivation

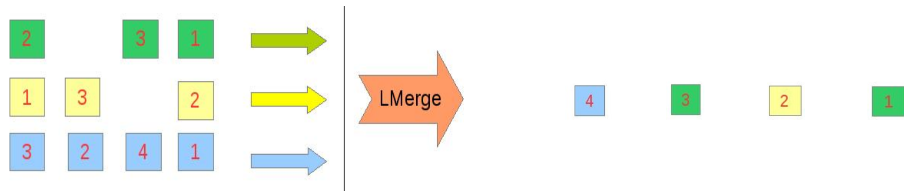


Motivation



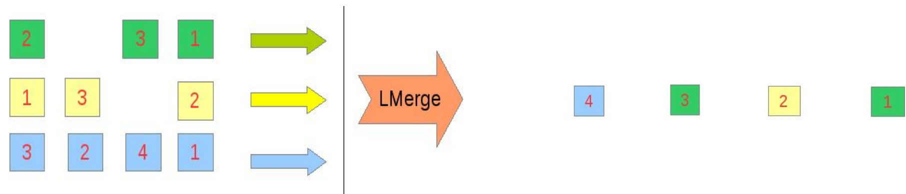
- Desirable properties

Motivation



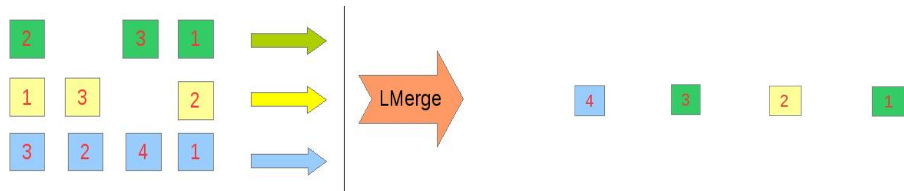
- Desirable properties
  - High Availability

Motivation



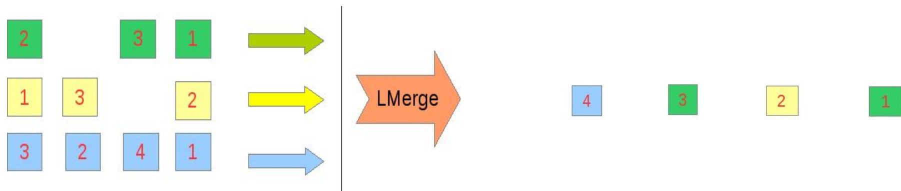
- Desirable properties
  - High Availability
  - Fast Availability

Motivation



- Desirable properties
  - High Availability
  - Fast Availability
  - Query Jump-start

Motivation



- Desirable properties
  - High Availability
  - Fast Availability
  - Query Jump-start
  - Query Cut-over

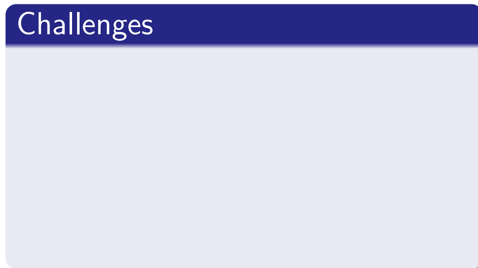
Challenges/Requirements

Phy1	Phy2
	alloc(A,6,7)
	alloc(B,8,15)
alloc(B,8,inf)	modify(A,6,12)
alloc(A,6,12)	
modify(B,8,1o)	modify(B,8,10)

Table: Two Streams Tracking DHCP Leases

Userid	Lease
A	[6,12)
B	[8,10)

Table: Effective DHCP Lease Summary



Challenges/Requirements

Phy1	Phy2
	alloc(A,6,7)
	alloc(B,8,15)
alloc(B,8,inf)	modify(A,6,12)
alloc(A,6,12)	
modify(B,8,1o)	modify(B,8,10)

Table: Two Streams Tracking DHCP Leases

Userid	Lease
A	[6,12)
B	[8,10)

Table: Effective DHCP Lease Summary

Challenges

- Example: DHCP leases

Challenges/Requirements

Phy1	Phy2
	alloc(A,6,7)
	alloc(B,8,15)
alloc(B,8,inf)	modify(A,6,12)
alloc(A,6,12)	
modify(B,8,1o)	modify(B,8,10)

Table: Two Streams Tracking DHCP Leases

Userid	Lease
A	[6,12)
B	[8,10)

Table: Effective DHCP Lease Summary

Challenges

- Example: DHCP leases
- Disorder

Challenges/Requirements

Phy1	Phy2
	alloc(A,6,7)
	alloc(B,8,15)
alloc(B,8,inf)	modify(A,6,12)
alloc(A,6,12)	
modify(B,8,1o)	modify(B,8,10)

Table: Two Streams Tracking DHCP Leases

Userid	Lease
A	[6,12)
B	[8,10)

Table: Effective DHCP Lease Summary

Challenges

- Example: DHCP leases
- Disorder
- Revisions

Challenges/Requirements

Phy1	Phy2
	alloc(A,6,7)
	alloc(B,8,15)
alloc(B,8,inf)	modify(A,6,12)
alloc(A,6,12)	
modify(B,8,1o)	modify(B,8,10)

Table: Two Streams Tracking DHCP Leases

Userid	Lease
A	[6,12)
B	[8,10)

Table: Effective DHCP Lease Summary

Challenges

- Example: DHCP leases
- Disorder
- Revisions
- Processing variations

Challenges/Requirements

Phy1	Phy2
	alloc(A,6,7)
	alloc(B,8,15)
alloc(B,8,inf)	modify(A,6,12)
alloc(A,6,12)	
modify(B,8,1o)	modify(B,8,10)

Table: Two Streams Tracking DHCP Leases

Userid	Lease
A	[6,12)
B	[8,10)

Table: Effective DHCP Lease Summary

Challenges

- Example: DHCP leases
- Disorder
- Revisions
- Processing variations
- Handling failures

Theoretical Treatment

- Stream as Temporal Database(TDB)
- TDB schema $(p, [V_s, V_e))$
- Stream prefix $S[i] = e_1, e_2, ..., e_i$
- Reconstitution function $tdb(S, i)$
- Prefix equivalence $S[i] \equiv U[j] \iff tdb(S, i) = tdb(U, j)$
- Stream equivalence $S \equiv U \iff tdb(S)$ and $tdb(U)$ are well defined and equal

Stream S
open(A,1)
open(B,2)
open(C,3)
close(A,4)
close(B,5)

Table: Sample Stream

p	V_s	V_e
A	1	4
B	2	5
C	3	∞

Table: tdb(S,5)

Prefix Equivalence

S	**U**	**W**
open(A,1)	open(A,1)	open(B,2)
open(B,2)	close(A,4)	close(B,6)
open(C,3)	open(B,2)	open(A,1)
close(A,4)	close(B,5)	open(C,3)
close(B,5)	open(C,3)	close(A,4)
		close(B,5)

Table: Sample Stream Prefixes

p	V_s	V_e
A	1	4
B	2	5
C	3	∞

Table: Equivalent TDB

Stream Elements

- Stream elements for algorithms
- Used in Microsoft StreamInsights

Stream Elements

- Stream elements for algorithms
- Used in Microsoft StreamInsights
- $insert(p, V_s, V_e)$

Stream Elements

- Stream elements for algorithms
- Used in Microsoft StreamInsights
- $insert(p, V_s, V_e)$
- $adjust(p, V_s, V_{old}, V_e)$

Stream Elements

- Stream elements for algorithms
- Used in Microsoft StreamInsights
- $insert(p, V_s, V_e)$
- $adjust(p, V_s, V_{old}, V_e)$
- $stable(Vc)$

Stream Elements

- Stream elements for algorithms
- Used in Microsoft StreamInsights
- $insert(p, V_s, V_e)$
- $adjust(p, V_s, V_{old}, V_e)$
- $stable(Vc)$
  - State reduction
  - Freezes certain parts of TDB

p	V_s	V_e	status
A	2	16	HF
B	3	10	FF
C	4	18	HF
D	15	20	UF

Table: TDB event freeze status after $stable(14)$

Freeze status

HF: Half frozen FF: Full Frozen UF: Unfrozen

Examples of freeze status

- Example 1

  ins(A,6,10)
  ins(B,7,15)
  stable(12)
  adjust(A,6,10,11)   Not allowed since $Vc < Vold$
  adjust(B,7,15,13)   Allowed since $Vc > Vold$

- Example 2

  ins(A,6,10)
  ins(B,15,20)
  stable(12)
  adjust(B,15,20,15)   Allowed and effectively removes event B

Logical Merge

- **Reference Stream**:
  - Absolute complete stream seen by omnipresent observer
  - Logical entity

Logical Merge

- **Reference Stream**:
  - Absolute complete stream seen by omnipresent observer
  - Logical entity
- Input Stream Assumption: Events are never lost.

Logical Merge

- **Reference Stream**:
  - Absolute complete stream seen by omnipresent observer
  - Logical entity
- Input Stream Assumption: Events are never lost.
- **Mutual Consistency**: Required property for input streams
  - Stream prefixes $I_1[K_1], I_2[K_2], ..., I_n[K_n]$ mutually consistent
  - if exists finite E_i, F_i, $1 <= i <= n$
  - $E_1 : I_1[K_1] : F_1 \equiv E_2 : I_2[K_2] : F_2 \equiv .... \equiv E_n : I_n[K_n] : F_n$

Logical Merge

- **Reference Stream**:
  - Absolute complete stream seen by omnipresent observer
  - Logical entity
- Input Stream Assumption: Events are never lost.
- **Mutual Consistency**: Required property for input streams
  - Stream prefixes $I_1[K_1], I_2[K_2], ..., I_n[K_n]$ mutually consistent
  - if exists finite E_i, F_i, $1 <= i <= n$
  - $E_1 : I_1[K_1] : F_1 \equiv E_2 : I_2[K_2] : F_2 \equiv .... \equiv E_n : I_n[K_n] : F_n$

- $Lmerge(I_1, I_2, ..., I_n) = O$
  - $I_1, I_2, ..., I_n, O$ mutually consistent without extending O and O is minimal

Logical Merge

- **Reference Stream**:
  - Absolute complete stream seen by omnipresent observer
  - Logical entity
- Input Stream Assumption: Events are never lost.
- **Mutual Consistency**: Required property for input streams
  - Stream prefixes $I_1[K_1], I_2[K_2], ..., I_n[K_n]$ mutually consistent
  - if exists finite $E_i, F_i, 1 <= i <= n$
  - $E_1 : I_1[K_1] : F_1 \equiv E_2 : I_2[K_2] : F_2 \equiv .... \equiv E_n : I_n[K_n] : F_n$

$\equiv$

- $Lmerge(I_1, I_2, ..., I_n) = O$
  - $I_1, I_2, ..., I_n, O$ mutually consistent without extending O and O is minimal

Output Policies

- **Aggressive** : propagating every change from the inputs as it is seen. e.g. out1
- **Conservative** : delaying elements until they are known to be stable. e.g. out2
- **In-between** : outputs first element with a given payload and start, saves any modification until they are known to be stable. e.g. out3

Table: Chattiness: Input and output streams

In1	In2	Out1	Out2	Out3
ins(A,6,10)		ins(A,6,10)		ins(A,6,10)
	ins(A,6,12)	adj(A,6,10,12)		
	ins(B,7,14)	ins(B,7,14)		ins(B,7,14)
adj(A,6,10,15)		adj(A,6,12,15)		
	adj(A,6,12,15)			
	stable(16)	stable(16)	ins(A,6,15) ins(B,7,14) stable(16)	adj(A,6,10,15) stable(16)

LMerge R0 Algorithm

- Input Stream Assumptions
  - insert() and stable() elements with strictly increasing Vs.
  - Stream has deterministic order with no duplicates.

Algorithm

```
1  MaxVs=-inf;
2  MaxStable=-inf;
3  void Insert(element y,stream r){
4      if(y.Vs > MaxVs){
5          OutputInsert(y); // Add event to output stream
6          MaxVs=y.Vs
7      }
8  }
9
10 void Stable(timestamp t,Stream r)
11 {
12     if(t > MaxStable){
13         OutputInsert(stable(t));
14         MaxStable=t;
15     }
16
17 }
```

LMerge algorithms and Stream Properties

- R1
  - Input Stream Assumptions
    - insert() and stable elements with non-decreasing Vs.
    - Order among elements with equal Vs deterministic.
  - Algorithm
    - Maintains MaxStable, MaxVs, and an array with one counter for each input stream.
    - Counter counts the number of elements on stream with Vs=MaxVs.

LMerge algorithms and Stream Properties

- R1
  - Input Stream Assumptions
    - insert() and stable elements with non-decreasing Vs.
    - Order among elements with equal Vs deterministic.
  - Algorithm
    - Maintains MaxStable, MaxVs, and an array with one counter for each input stream.
    - Counter counts the number of elements on stream with Vs=MaxVs.
- R2
  - Properties
    - insert() and stable elements with non-decreasing Vs.
    - Order among elements with equal Vs non-deterministic.
    - (P,Vs) is a key of TDB for any prefix
  - Algorithm
    - Maintains MaxStable, MaxVs, and a hash table.
    - Hash table indexes (using payload as a key) all elements with Vs=MaxVs.

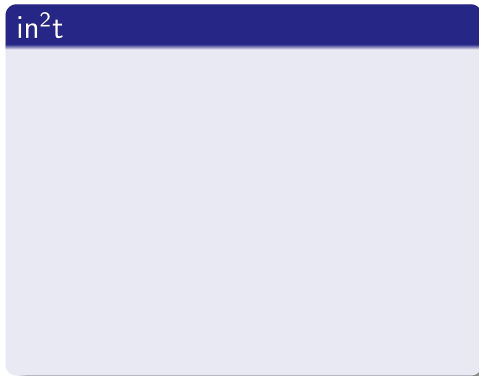
LMerge Algorithms and Input Properties

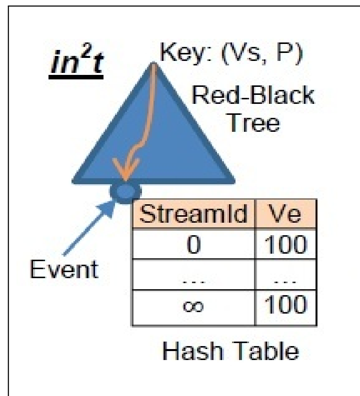
- R3
  - Input Stream Assumptions
    - No constraint on time order, except imposed by stable elements
    - (P,Vs) is a key of TDB for any prefix
  - Algorithm
    - Explained later

LMerge Algorithms and Input Properties

- R3
  - Input Stream Assumptions
    - No constraint on time order, except imposed by stable elements
    - (P,Vs) is a key of TDB for any prefix
  - Algorithm
    - Explained later
- R4
  - Input Stream Assumptions
    - No constraints.
  - Algorithm
    - Explained later

Data structure for case R3 (in^2t) of LMerge

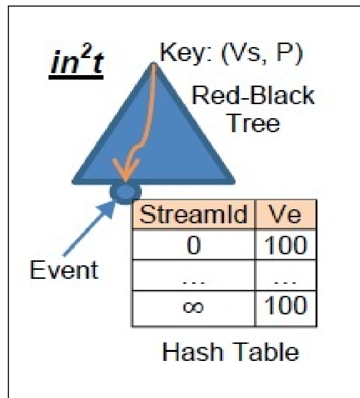




Data structure for case R3 (in^{2}t) of LMerge

in^{2}t

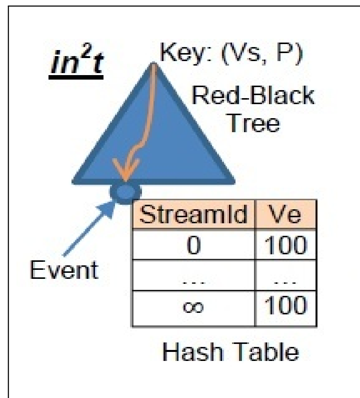
- Top tier of in^{2}t is a red-black tree keyed by (Vs,Payload).



Data structure for case R3 (in^{2}t) of LMerge

in^{2}t

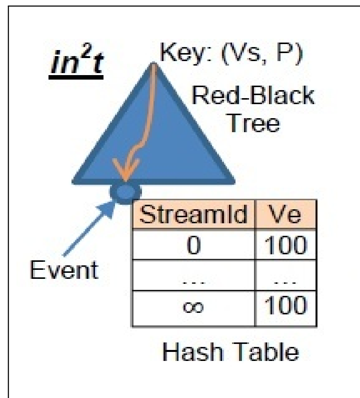
- Top tier of in^{2}t is a red-black tree keyed by (Vs,Payload).
- Points to a second-tier index implemented as a hash table.



Data structure for case R3 (in^2t) of LMerge

in^2t

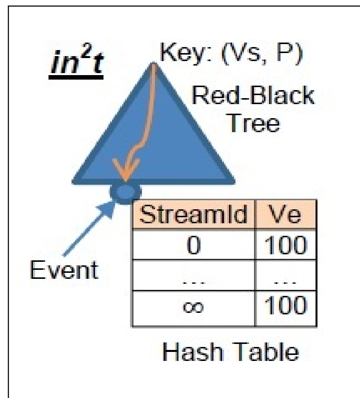
- Top tier of in^2t is a red-black tree keyed by (Vs,Payload).
- Points to a second-tier index implemented as a hash table.
- Hash table contains, entries (r,Ve value for stream r).



Data structure for case R3 (in^2t) of LMerge

in^2t

- Top tier of in^2t is a red-black tree keyed by (Vs,Payload).
- Points to a second-tier index implemented as a hash table.
- Hash table contains, entries (r,Ve value for stream r).
- Additional entry with a special key ∞.



Algorithm R3: Logical Merge for case R3

Insert

```
1  void insert(element y, stream r){
2  node f = index.SameVsPayload(y)
3  if (!exists(f)){
4    if (y.Vs < MaxStable) return;
5    f = index.AddNode(y);
6    OutputInsert(y);
7    f.AddHashEntry(inf, y.Ve); // hash entry for o/p
8    }
9  f.AddHashEntry(r, y.Ve); // hash entry for i/p
10 }
```

Algorithm R3: Logical Merge for case R3

Insert

```
1  void insert(element y, stream r){
2  node f = index.SameVsPayload(y)
3  if (!exists(f)){
4    if (y.Vs < MaxStable) return;
5    f = index.AddNode(y);
6    OutputInsert(y);
7    f.AddHashEntry(inf, y.Ve); // hash entry for o/p
8    }
9  f.AddHashEntry(r, y.Ve); // hash entry for i/p
10 }
```

Adjust

```
1  void adjust(element y, stream r){
2      node f = index.sameVsPayload(y);
3      if ( !exists(f)) return;
4      f.updateHashEntry(r, y.Ve);
5  }
```

Algorithm R3: Logical Merge for case R3

Stable

```
1  void stable(timestamp t, stream r) {
2      if ( t <= MaxStable) return;
3      iterator it = index.findHalfFrozen(t);
4      while ( node f = it.next()) {
5          InVe = f.getHashEntry(r);
6          if (!exists(InVe)) InVe = f.getEvent().Vs;
7          OutVe = f.getHashEntry(inf);
8          if (InVe != OutVe && (InVe <t or OutVe < t)) {
9              outputAdjust(f.getEvent(), Ve: InVe);
0              f.updateHashEntry(inf, InVe);
1          }
2          if (InVe < t) //fully frozen
3              index.deleteNode(f);
4      //update MaxStable and output a stable() element
5      MaxStable = t;
6      outputStable(t);
7  }
```

Algorithm R3: Logical Merge for case R3

r'	r	O
(A,10,15)		(A,10,15)
	Stable(12)	
		Adjust(A,10,10) Stable(12)

Table: Adjustment: Delete output element

Stable

```
1  void stable(timestamp t, stream r) {
2      if ( t <= MaxStable) return;
3      iterator it = index.findHalfFrozen(t);
4      while ( node f = it.next()) {
5          InVe = f.getHashEntry(r);
6          if (!exists(InVe)) InVe = f.getEvent().Vs;
7          OutVe = f.getHashEntry(inf);
8          if (InVe != OutVe && (InVe <t or OutVe < t)) {
9              outputAdjust(f.getEvent(), Ve: InVe);
10             f.updateHashEntry(inf, InVe);
11         }
12         if (InVe < t) //fully frozen
13             index.deleteNode(f);
14     //update MaxStable and output a stable() element
15     MaxStable = t;
16     outputStable(t);
17 }
```

Algorithm R3: Logical Merge for case R3

Stable

```
1  void stable(timestamp t, stream r) {
2      if ( t <= MaxStable) return;
3      iterator it = index.findHalfFrozen(t);
4      while ( node f = it.next()) {
5          InVe = f.getHashEntry(r);
6          if (!exists(InVe)) InVe = f.getEvent().Vs;
7          OutVe = f.getHashEntry(inf);
8          if (InVe != OutVe && (InVe <t or OutVe < t)) {
9              outputAdjust(f.getEvent(), Ve: InVe);
10             f.updateHashEntry(inf, InVe);
11         }
12         if (InVe < t) //fully frozen
13             index.deleteNode(f);
14     //update MaxStable and output a stable() element
15     MaxStable = t;
16     outputStable(t);
17 }
```

r'	r	O
(A,10,15)		(A,10,15)
	Stable(12)	
		Adjust(A,10,10) Stable(12)

Table: Adjustment: Delete output element

r'	r	O
(A,10,15)		(A,10,15)
	(A,10,20)	
	Stable(12)	
		Adjust(A,10,20) Stable(18)

Table: Adjustment: Half freeze output element

Algorithm R3: Logical Merge for case R3

Stable

```
1  void stable(timestamp t, stream r) {
2      if ( t <= MaxStable) return;
3      iterator it = index.findHalfFrozen(t);
4      while ( node f = it.next()) {
5          InVe = f.getHashEntry(r);
6          if (!exists(InVe)) InVe = f.getEvent().Vs;
7          OutVe = f.getHashEntry(inf);
8          if (InVe != OutVe && (InVe <t or OutVe < t)) {
9              outputAdjust(f.getEvent(), Ve: InVe);
0              f.updateHashEntry(inf, InVe);
1          }
2          if (InVe < t) //fully frozen
3              index.deleteNode(f);
4      //update MaxStable and output a stable() element
5      MaxStable = t;
6      outputStable(t);
7  }
```

r’	r	O
(A,10,15)		(A,10,15)
	Stable(12)	
		Adjust(A,10,10) Stable(12)

Table: Adjustment: Delete output element

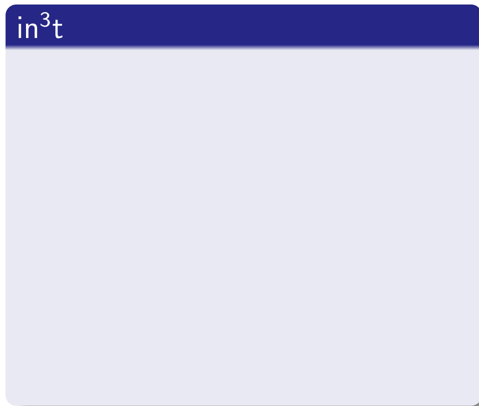
r’	r	O
(A,10,15)		(A,10,15)
	(A,10,20)	
	Stable(12)	
		Adjust(A,10,20) Stable(18)

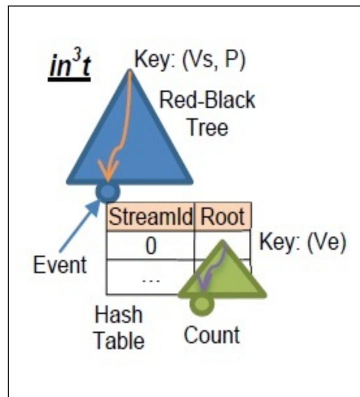
Table: Adjustment: Half freeze output element

r’	r	O
(A,10,20)		(A,10,20)
	(A,10,17)	
	Stable(18)	
		Adjust(A,10,17) Stable(18)

Table: Adjustment: Full freeze output element

Data structure for case R4 (in^{3}t) of LMerge

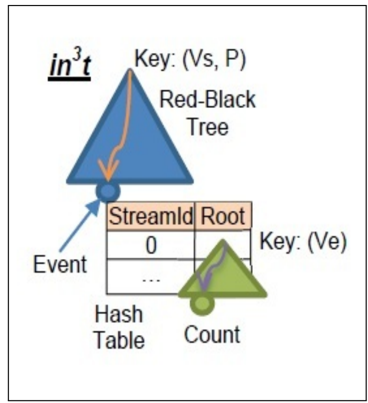




Data structure for case R4 (in^{3}t) of LMerge

in^{3}t

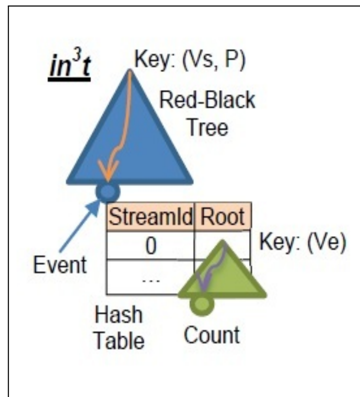
- Top tier of in^{3}t is a red-black tree keyed by (Vs,Payload).



Data structure for case R4 (in^{3}t) of LMerge

in^{3}t

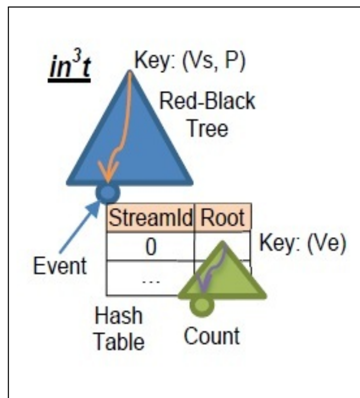
- Top tier of in^{3}t is a red-black tree keyed by (Vs,Payload).
- Points to a second-tier index implemented as a hash table.



Data structure for case R4 (in^{3}t) of LMerge

in^{3}t

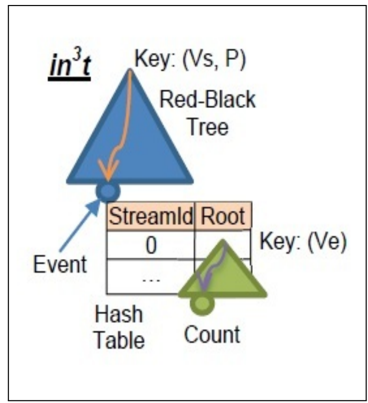
- Top tier of in^{3}t is a red-black tree keyed by (Vs,Payload).
- Points to a second-tier index implemented as a hash table.
- Hash table contains, entries (r,small index (red-black-tree) on Ve).



Data structure for case R4 (in^{3}t) of LMerge

in^{3}t

- Top tier of in^{3}t is a red-black tree keyed by (Vs,Payload).
- Points to a second-tier index implemented as a hash table.
- Hash table contains, entries (r,small index (red-black-tree) on Ve).
- Each Ve is associated with its count.



Algorithm R4: Logical Merge for Case R4

Insert

```
1  void insert(element y, stream r){
2      node f = index.sameVsPayload(y);
3      if (!exists(f)) {
4          if (y.Vs < MaxStable) return;
5          f = index.AddNode(y);
6      }
7      if ((y.vs>=MaxStable) && (f.getCount>f.getCount(inf))){
8          outputInsert(y);
9          f.incrementCount(inf, y.Ve);
10     }
11 }
```

Algorithm R4: Logical Merge for Case R4

Insert

```
1  void insert(element y, stream r){
2      node f = index.sameVsPayload(y);
3      if (!exists(f)) {
4          if (y.Vs < MaxStable) return;
5          f = index.AddNode(y);
6      }
7      if ((y.vs>=MaxStable) && (f.getCount>f.getCount(inf))){
8          outputInsert(y);
9          f.incrementCount(inf, y.Ve);
10     }
11 }
```

Adjust

```
1  void adjust(element y, stream r) {
2      node f = index.sameVsPayload(y);
3      if (!exists(f)) return;
4      f.incrementCount(r, y.ve);
5      f.decrementCount(r, y.vold);
6  }
```

Algorithm R4: Logical Merge for Case R4

Stable

```
1  void stable(timestamp t, stream r) {
2      if (t <= MaxStable) return;
3      iterator it = index.FindHalfFrozen(t);
4      while ( node f = it.next()) {
5          if(f.Vs >= maxStable) { //elements getting half frozen
6          //ensure #o/p events = #i/p events for that (Vs, P)
7          adjustOutputCount(f);
8      }
9      iterator itIn = f.findAllVe(r);
10     iterator itOut = f.findAllVe(inf);
11     // Make o/p reflect i/p for all FF (Ve < t) nodes
12     adjustOutput(f, t, itIn, itOut);
13     if (f.getMaxVe(r) < t) { // Done processing that (Vs, P)
14          index.delete(f);
15     }
16     MaxStable = t;
17     OutputStable(t);
18 }
```

Setup and Implementation

Infrastructure

Processor: 2.3GHz 2 Processors, 8 Cores **Memory**: 16GB **OS**: Windows Server 2008 R2

Algorithms

- LMR^0
- LMR^1
- LMR^2
- LMR^{3+}: LMR3 explained earlier
- LMR^{3-}: Separate index for each input stream
- LMR^4

Metrics and Workload

Metrics

- Throughput
- Memory
- Output size

Stream Parameters

- StableFreq: Percentage of stable elements. Default 1 percent
- EventDuration: Event validity interval. Default 10k active
- MaxGap: Default 20s
- Disorder: Fraction of disordered elements. Default 20 percent

LMerge Over Ordered Streams

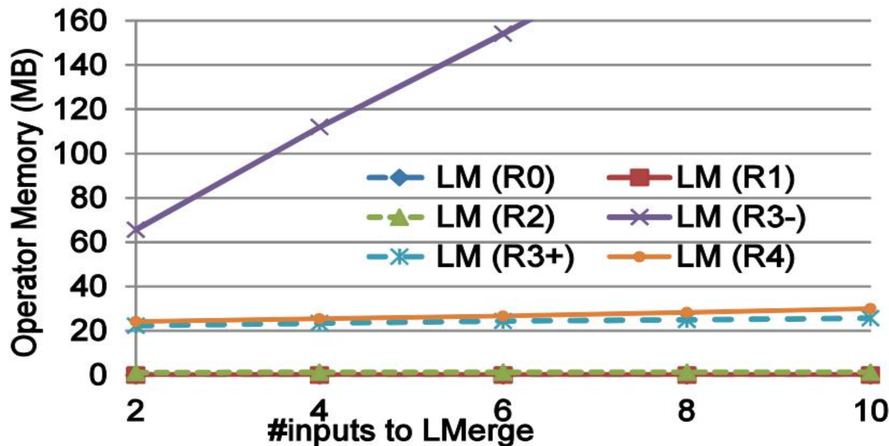


Figure: Memory, Number of input streams

LMerge: Increasing Disorder

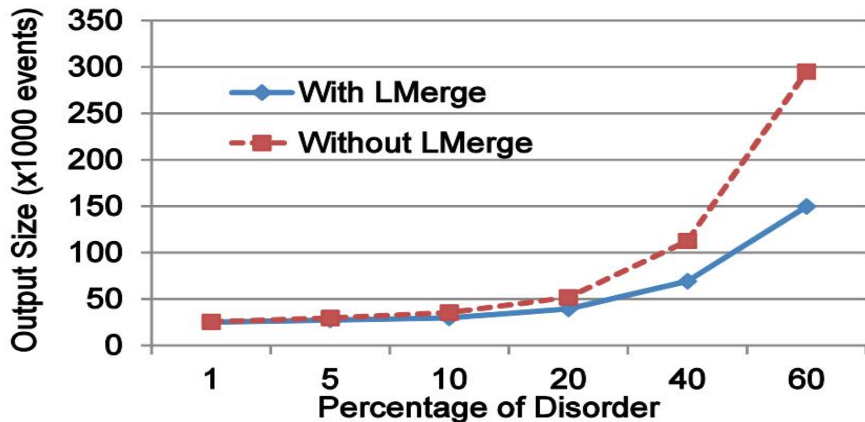


Figure: Output size, Disorder Percentage

LMerge Scenarios

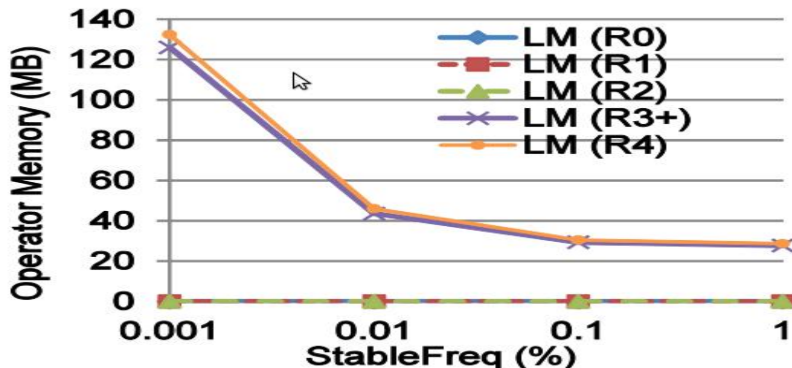


Figure: LMerge Memory usage over increasing StableFreq

LMerge Scenarios

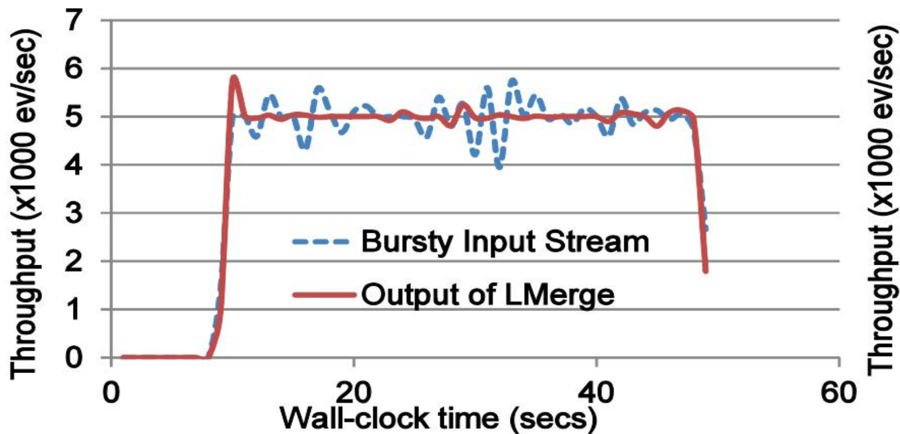


Figure: Handling Bursty Input

LMerge Scenarios

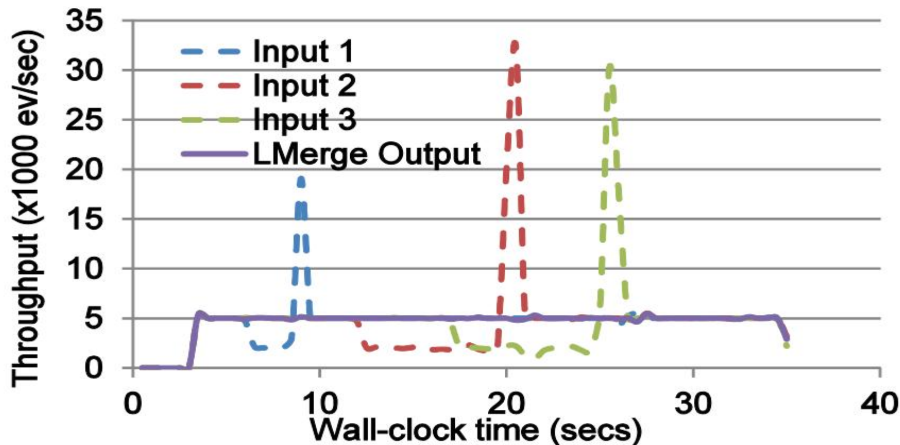


Figure: Handling Network Congestion

Thank You!

Questions?