MaSM: Efficient Online Updates in Data Warehouses

Manos Athanassoulis¹  Shimin Chen²  Anastasia Ailamaki¹  Radu Stoica¹
Phillip Gibbons²

¹EPFL  ²Intel Labs
Freshness vs Performance

- Data warehouse workload
  - Read-only queries (scans)
  - Scattered updates
  - Difficult to combine efficiently

- Traditionally two choices
  - **Freshness**: in-place updates
  - **Performance**: batch updates

- Ideally, zero overhead
Freshness vs Performance

• Data warehouse workload
  – Read-only queries (scans)
  – Scattered updates
  – Difficult to combine efficiently

• Traditionally two choices
  – **Freshness**: in-place updates
  – **Performance**: batch updates

• Ideally, zero overhead

Is zero overhead possible?
Freshness **AND** Performance

In Memory Buffered Updates

- Apply them online
- Apply them as differential updates
- Large memory overhead
- Trade-off migration overhead for memory footprint

To sum up

<table>
<thead>
<tr>
<th>Update Approach</th>
<th>Freshness</th>
<th>Performance</th>
<th>↓ mem overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batched</td>
<td>X</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>In place</td>
<td>☺</td>
<td>X</td>
<td>☺</td>
</tr>
<tr>
<td>In-memory differential</td>
<td>☺</td>
<td>☺</td>
<td>☻</td>
</tr>
</tbody>
</table>

[Stonebraker et al.’05] [Heman et al.’10]

1. "1"
2. "10"
3. "100"
4. "1000"

- cache updates in memory
- normalized migration overhead
- in-memory buffer size

---

1. "16MB"
2. "128MB"
3. "1GB"
4. "8GB"

- ideal
- in-memory buffer size

---

"AiAS"
Freshness **AND** Performance

**In Memory Buffered Updates**

- Apply them online
- Apply them as differential updates
- Large memory overhead
- Trade-off migration overhead for memory footprint

To sum up

<table>
<thead>
<tr>
<th>Update Approach</th>
<th>Freshness</th>
<th>Performance</th>
<th>↓ mem overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batched</td>
<td>X</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>In place</td>
<td>☺</td>
<td>X</td>
<td>☺</td>
</tr>
<tr>
<td>In-memory differential</td>
<td>☺</td>
<td>☺</td>
<td>☺ X</td>
</tr>
</tbody>
</table>

*Can we have the cake and eat it too?*
Use MaSM!

- Buffer updates on Flash instead of memory
  - Flash has *larger capacity* and *smaller price*
- **But:** Flash friendly design is important
  - Avoid random writes
  - Limit total writes
  - e.g. Log-Structure Merge Tree incurs a high number writes per update

[O’ Neil et al.’96]
Use MaSM!

- **Buffer updates on Flash instead of memory**
  - Flash has *larger capacity* and *smaller price*

- **But**: Flash friendly design is important
  - Avoid random writes
  - Limit total writes
    - e.g. Log-Structure Merge Tree incurs a high number writes per update
  
  [O’ Neil et al.’96]
MaSM core idea

 Updates (U)  
<table>
<thead>
<tr>
<th>K</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>V1’</td>
<td>Mod</td>
</tr>
<tr>
<td>5</td>
<td>V5”</td>
<td>Mod</td>
</tr>
<tr>
<td>9</td>
<td>N/A</td>
<td>Del</td>
</tr>
<tr>
<td>19</td>
<td>V19’</td>
<td>Mod</td>
</tr>
<tr>
<td>125</td>
<td>V125</td>
<td>Ins</td>
</tr>
<tr>
<td>5</td>
<td>V5”</td>
<td>Mod</td>
</tr>
</tbody>
</table>

 Current data

 ✓ Outer join: $D \bowtie U$
 ✓ Keep latest update only

 Efficient execution

 ✓ Discard duplicates
 ✓ Re-use information for future queries

 Sort-Merge Join

 ✓ Intuitively does both

 Data (D)  
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>V1</td>
</tr>
<tr>
<td>2</td>
<td>V2</td>
</tr>
<tr>
<td>3</td>
<td>V3</td>
</tr>
<tr>
<td>4</td>
<td>V4</td>
</tr>
<tr>
<td>5</td>
<td>V5</td>
</tr>
<tr>
<td>6</td>
<td>V6</td>
</tr>
<tr>
<td>7</td>
<td>V7</td>
</tr>
<tr>
<td>8</td>
<td>V8</td>
</tr>
<tr>
<td>9</td>
<td>V9</td>
</tr>
</tbody>
</table>
MaSM core idea

**Current data**
- ✔ Outer join: \( D \bowtie U \)
- ✔ Keep latest update only

**Efficient execution**
- ✔ Discard duplicates
- ✔ Re-use information for future queries

**Sort-Merge Join**

MaSM merges data with updates using sort-merge join and materializing sorted runs

### Updates (U)

<table>
<thead>
<tr>
<th>K</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>V1'</td>
<td>Mod</td>
</tr>
<tr>
<td>5</td>
<td>V5''</td>
<td>Mod</td>
</tr>
<tr>
<td>9</td>
<td>N/A</td>
<td>Del</td>
</tr>
<tr>
<td>19</td>
<td>V19'</td>
<td>Mod</td>
</tr>
<tr>
<td>125</td>
<td>V125</td>
<td>Ins</td>
</tr>
<tr>
<td>5</td>
<td>V5''</td>
<td>Mod</td>
</tr>
</tbody>
</table>

### Data (D)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>V1</td>
</tr>
<tr>
<td>2</td>
<td>V2</td>
</tr>
<tr>
<td>3</td>
<td>V3</td>
</tr>
<tr>
<td>4</td>
<td>V4</td>
</tr>
<tr>
<td>5</td>
<td>V5</td>
</tr>
<tr>
<td>6</td>
<td>V6</td>
</tr>
<tr>
<td>7</td>
<td>V7</td>
</tr>
<tr>
<td>8</td>
<td>V8</td>
</tr>
<tr>
<td>9</td>
<td>V9</td>
</tr>
</tbody>
</table>
Outline

• Introduction
  – Prior work: Differential Updates
  – MaSM sneak peak

• **MaSM architecture**

• Evaluation
  – Query response time
  – Sustained update rate

• Conclusions
MaSM in detail

Main memory

\[ M = \sqrt{|SSD|} \]

Disks (main data) e.g. TBs

SSD e.g. GBs

M pages
MaSM in detail

Main memory

Incoming query

Merge data & updates

M=

M pages

M=

SSD

e.g. GBs

Disks
(many data)
e.g. TBs

Table Range Scan

Run Scan

Run Scan

Run Scan

Mem Scan

M pages

M pages
MaSM in detail

Main memory

Incoming query

Merge data & updates

M = \sqrt{|SSD|}

M pages

M pages

Disks
(main data)
e.g. TBs

Table Range Scan

Run Scan

Run Scan

Run Scan

M = \sqrt{|SSD|}

Merge pages from HDD, SSD and RAM with negligible overhead!

SSD
e.g. GBs
Reducing MaSM memory

Main memory

Incoming query

Merge data & updates

Merge updates

αM-S pages

α ≤ 2, S ≤ M

Disks (main data) e.g. TBs

Run Scan

Table Range Scan

Run Scan

Run Scan

Mem Scan

SSD e.g. GBs

2-pass runs

1-pass runs
Reducing MaSM memory

\[ \alpha \leq 2, S \leq M \]

- **Main memory**
- **Disks** (main data) e.g. TBs
- **Incoming query**
- **Table Range Scan**
- **Merge data & updates**
- **Merge updates**
- **Mem Scan**
- **αM-S pages**
- **2-pass runs**
- **1-pass runs**
- **Trade-off extra writes for memory size.**

SSD e.g. GBs
Impact of $\alpha$ on SSD wear

Memory footprint = $\alpha M$  

$M = \sqrt{|SSD|} \quad f(M) \leq \alpha \leq 2 \quad \text{e.g., } M=1000, \ 0.2 \leq \alpha \leq 2$
Impact of $\alpha$ on SSD wear

Memory footprint = $\alpha M$

$M = \sqrt{|SSD|}$

$2 - 0.25\alpha^2$

$0.2M \leq f(M) < \alpha \leq 2$ for $\alpha \leq 2$

M = 1000

10x smaller memory for 2x more writes!
Outline

• Introduction
  – Prior work: Differential Updates
  – MaSM sneak peak

• MaSM architecture

• Evaluation
  – Query response time
  – Sustained update rate

• Conclusions
Experimental setup

- Dell Precision 690 Workstation
  - Intel Xeon Quad (2.33MHz, 8MB L2), 4GB DRAM, Ubuntu Linux, kernel 2.6.24

- Dedicated SATA disk for main data
  - 7200rpm Seagate Barracuda, 77MB/s sequential bandwidth

- Intel X25-E SSD for caching updates
  - 250 MB/s sequential read, 170MB/s sequential write bandwidth; 35,000 4KB-sized random reads/second

- Prototype row store:
  - Implemented in-place updates, indexed updates, MaSM
Query performance on synthetic data

- MaSM has negligible impact on 10MB or larger scans
- MaSM with fine-grain index incurs 4% overhead for 4KB ranges (modeling point queries)
• Replay TPCH disk traces recorded from commercial row store; random online updates
TPCH replay experiment

- Replay TPCH disk traces recorded from commercial row store; random online updates

Queries with MaSM see less than 1% overhead!
Update performance

- **in-place updates**: 48
- MaSM 2GB SSD: 3.5K
- MaSM 4GB SSD: 6K
- MaSM 8GB SSD: 12K

Update Rate (upd/s)
Update performance

Efficient usage of a few GB of flash can increase update rate up to 258x!
To sum up

MaSM enables **on-line updates** in DW

- Negligible query overhead (less than 1% for TPCH)
- Supports a **high update rate (up to 12k)**
- Tunable *memory footprint vs SSD wear*
- Low migration cost (one-time 2.2x)
- SSD-friendly behavior
  - Limited number of writes per updates
  - No random writes on SSD
- Easy DBMS integration
- Ensure ACID properties
To sum up

MaSM enables **on-line updates** in DW

- Negligible query overhead (less than 1% for TPCH)
- Supports a high update rate (up to 12k)
- Tunable memory footprint vs SSD wear
- Low migration cost (one-time 2.2x)
- SSD-friendly behavior
  - Limited number of writes per updates
  - No random writes on SSD
- Easy DBMS integration
- Ensure ACID properties

<table>
<thead>
<tr>
<th>Update Approach</th>
<th>Freshness</th>
<th>Performance</th>
<th>↓ mem overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batched</td>
<td>X</td>
<td>😊</td>
<td>😊</td>
</tr>
<tr>
<td>In place</td>
<td>😊</td>
<td>X</td>
<td>😊</td>
</tr>
<tr>
<td>In-memory differential</td>
<td>😊</td>
<td>😊</td>
<td>X</td>
</tr>
<tr>
<td><strong>MaSM and SSD</strong></td>
<td>😊</td>
<td>😊</td>
<td>😊</td>
</tr>
</tbody>
</table>

- Limited number of writes per updates
- No random writes on SSD

**Thank you!**