Team SimpleMind

Ismail Oukid (TU Dresden), Ingo Müller (KIT), Iraklis Psaroudakis (EPFL)
ACM SIGMOD 2015 Programming Contest @ SIGMOD 2015 (June 2)

Public
Agenda

- Programming Contest Overview
- Transaction Processing
- Data Structures for Validation
- Validation Processing
- Parallelization: Bulk-Synchronous
- Implementation Details
- Runtime Break-Down
Context: “Optimistic Concurrency Control”

- Given a sequence of transactions,
  - i.e., insert or delete statements
- A sequence of validation queries,
  - i.e., select statements on data modified by a range of transaction
- Detect for each validation whether it conflicted or not,
  - i.e., non-empty result set
Example Sequence: Loading + Transactions
(copied from http://db.in.tum.de/sigmod15contest/task.html)

Loading:
- `defineschema [3, 4]`
- `transaction 0 [] [0 [1 1 2 2 1 2 3 4 5 7 7 7]
  1 [1 0 0 0 3 0 0 1 4 1 1 1]]`

Transactions:
- `transaction 1 [ ] [0 [6 5 4]]`
- `transaction 2 [1 [4]]`
- `transaction 3 [0 [3]] [0 [3 5 6]]`

Primary keys of rows to delete:
- table id 0
- table id 1
- table id 3

Rows to insert:
- table id 0
- table id 3
Example Sequence (cont’d): Validations

Validation:

<table>
<thead>
<tr>
<th>Validation</th>
<th>TX range</th>
<th>Table id</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation 0</td>
<td>1 2</td>
<td>0 c0=4</td>
<td>1 c1&gt;8</td>
</tr>
<tr>
<td>validation 1</td>
<td>1 2</td>
<td>1 c2=1</td>
<td></td>
</tr>
<tr>
<td>validation 2</td>
<td>1 3</td>
<td>0 c0=3 c1=2</td>
<td>0 c2=4</td>
</tr>
</tbody>
</table>

Task:

For every validation, check for conflict, i.e., check whether a transaction from the given range modified data that matches the predicates of the validation.

Example Output: 0 1 1

Workflow:

Validations only need to answered when a „Flush“ is triggered.
Programming Contest: Data Sets + Statistics

Data Sets:
• Three sizes: “small” (90MB), “medium” (900MB), “large” (9GB?)
• “Small” and “medium” available for testing,
• “Large” used to determine 5 finalists in online submission system
• Winner announced on SIGMOD with an “extra-large” data set

Statistics (approximate):
• 80% of the messages are validations
• <10% of the validations conflict
• 80% of the transactions go to one table
• 90% of the predicates are equality (=)
• 50% of the queries use the primary key columns
Transaction Processing

Each relation consists of:

- A **row-store** of valid and deleted rows
- A **primary key (PK) index** (PK → valid rows) for fast updates
- A two-level “**history index**” for fast validation of single rows:
  Transaction ID (TX ID) → list of ptrs to modified rows → row
Data Structures for Validation

The **modified rows** are converted periodically to **column-wise format**. Additional metadata include:

- **A single level “history index”** (TX ID → offset of first modified row)
- **8-bit fingerprint columns** (for superfast approximate scans)
- **A sample of distinct values per column** (to **estimate selectivity**)

![Diagram showing modified row conversion and metadata](image)

- **History index**: 64-bit/8-bit
- **C1 (PK)**: 4 distincts
  - Min: 1 Max: 4
- **C2**: 4 distincts
  - Min: 0 Max: 623
- **C3**: 3 distincts
  - Min: 1 Max: 834
Validation Processing (1/2)

Simple nested loops:
1. Validations in request stream
2. Queries in validation
3. Predicates in query
4. Rows in transaction range
Validation Processing (2/2)

Very fast predicate evaluation:
- Everything is a **scan**
- Result is filter for the next scan
- Heuristic selects **selective scans first**
- First scan is **approximate** (if possible)
  - 8 bit values, vectorized
Parallelization: Bulk-Synchronous

- The row-store is **hash-partitioned**. Each thread only executes transactions of its partition. Validations are queued.
- On flush request, the partitions are merged into the column-store.

  ![Diagram of Parallelization](image)

- Afterwards, threads process validations from the queue, now accessing all data structures in a read-only fashion.
- Additional flushes to overcome slow test driver.
Implementation Details

Simple
- **1268 lines of code** (according to **sloccount**)  
  - vs. 165 of the reference implementation
- Simple parallel regions with **OpenMP**  
  - plus a bit of last-minute mess with boost threads
- Extensive use of **STL** (and **c++11**) , a bit of **boost**, nothing else
- Indented 4 **spaces**

A few noticeable tweaks (>10% gain)
- „**Infinite“ vectors** thanks to Linux‘ overallocation  
  - **malloc(system_mem_size)**
- **Branch-free** scans
- History index is a **boost::flat_map**
- **Recycle memory** to avoid (serial!) mapping by OS
- **Simple** scan selection mechanism
Runtime Break-Down

This is a screenshot of the execution flow from Intel VTune Amplifier.

TX processing faster than test driver

Test driver + reader thread

Flush received

Transaction processing

Build column-store and history index

Validation processing

Additional validation threads

Repeat
Contact information: i.oukid@sap.com, ingo.mueller@kit.edu, iraklis.psaroudakakis@epfl.ch
SAP HANA Campus: students-hana@sap.com, http://tinyurl.com/hanacampus

Thank you