A Top-Down Approach to Achieving Performance Predictability in Database Systems

Harunobu Daikoku
HPCS Lab., University of Tsukuba
SIGMOD 2017 - Conference Overview

• Date: 5/14 (Sun) ~ 5/19 (Fri)
• Venue: Hilton Chicago, IL, USA
• # attendees: approx. 800

<table>
<thead>
<tr>
<th></th>
<th>2015</th>
<th>2016</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>submitted</td>
<td>413</td>
<td>569</td>
<td>489</td>
</tr>
<tr>
<td>accepted</td>
<td>106  (25%)</td>
<td>116  (20%)</td>
<td>96  (19%)</td>
</tr>
<tr>
<td>Industrial</td>
<td></td>
<td></td>
<td>invitation only</td>
</tr>
<tr>
<td>submitted</td>
<td>18</td>
<td>50</td>
<td>invitation only</td>
</tr>
<tr>
<td>accepted</td>
<td>18</td>
<td>17 + 4 (invited)</td>
<td>4 (invited)</td>
</tr>
<tr>
<td>Demos</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>submitted</td>
<td>86</td>
<td>126</td>
<td>90</td>
</tr>
<tr>
<td>accepted</td>
<td>30</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>Tutorials</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>submitted</td>
<td>11</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>accepted</td>
<td>4</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>
A Top-Down Approach to Achieving Performance Predictability in Database Systems

Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, Thomas F. Wenisch
University of Michigan

* Some of the figures in this document are taken from the original literature.
tl;dr

• Stop focusing only on raw performances (e.g. throughput, mean latency).

• Should be looking at performance predictability as well.

• TProfiler: a performance tracing tool that identifies sources of latency variance in DBMSs.

• Successfully identifies and mitigates major sources of performance unpredictability in MySQL, PostgreSQL and VoltDB.
Performance Predictability

• **Predictability: Variance**

• Why so important?
  • DB-backed web services *(latency directly affects user experience)*
  • Service-Level Agreements *("if violated, ...result in financial penalties")*

• How bad is it?

---

**Figure 6**: Mean, standard deviation, and 99th percentile latencies in MySQL (left), Postgres (center), and VoltDB (right).
Sources of Unpredictability

- **Avoidable** (internal): caused by internal components of DBMSs (e.g. I/Os, contention, data structures, algorithms)

- Inherent (external): caused by varying amounts of work (e.g. “a transaction that updates 10 tables inherently involves more work than one that updates only one table”)

TProfiler (VProfiler) - Overview

• Given the source codes of a DBMS (w/ explicit annotations of txns.), identifies sources of latency variance by generating a call graph called “a variance tree”

• Open-sourced: https://web.eecs.umich.edu/vprofiler/

  (VProfiler: a generalized version of TProfiler presented @ Eurosys 2017)
• Existing tools (e.g. DTrace [37]) are ignorant of...
  • Transaction-related code sequences inside the codebase
  • Mathematical nature of variance - “the variance of a parent function is always strictly greater than the variance of its children…”

\[
\text{Var}(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} \text{Var}(X_i) + 2 \sum_{1 \leq i \leq j \leq n} \text{Cov}(X_i, X_j)
\]

Figure 1: A call graph and its corresponding variance tree (here, body_A represents the time spent in the body of A).
TProfiler- Scoring Function

- Considers both **variance** and **depth** within the call hierarchy
- Intuition: "functions deeper in the call graph implement more specific functionality", thus are more informative

$$specificity(\phi) = (height(call\_graph) - height(\phi))^2$$

$$score(\phi) = specificity(\phi) \sum_i V(\phi_i)$$
TProfiler vs DTrace [37]

• DTrace: instruments the **binary code** rather than the source code, “use heavy-weight mechanisms to inject generalized instrumentation code at run-time”
TProfiler vs Naïve Profiler

- Naïve Profiler: decomposes every single non-leaf functions in a call graph rather than a few important ones.
Case Studies

• Workload: TPC-C

• Analyzed 3 popular open-source DBMSs
  • MySQL 5.6.23 (a thread-per-connection model)
    • 128 WHs w/ 30 GB buffer pool (high contention on records)
    • 2 WHs w/ 128 MB buffer pool (high contention on the buffer pool)
  • PostgreSQL 9.6 (a process-per-connection model)
    • 32 WHs w/ 30 GB buffer pool
  • VoltDB (an event-based server model)
Case Studies – MySQL (128 WHs)

- **os_event_wait()**: used to put a thread to sleep when it requested a lock on a record that cannot be granted due to a conflict ([A]: SELECT statements, [B]: UPDATE statements) -> **AVOIDABLE**

- **row_ins_clust_index_entry_low()**: inserts a new record into a clustered index, takes varying code paths based on the state of the index -> **INHERENT**

<table>
<thead>
<tr>
<th>Config</th>
<th>Function Name</th>
<th>Percentage of Overall Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-WH</td>
<td>os_event_wait [A]</td>
<td>37.5%</td>
</tr>
<tr>
<td>128-WH</td>
<td>os_event_wait [B]</td>
<td>21.7%</td>
</tr>
<tr>
<td>128-WH</td>
<td>row_ins_clust_index_entry_low</td>
<td>9.3%</td>
</tr>
</tbody>
</table>
Case Studies – MySQL (2 WHs)

- **buf_pool_mutex_enter**: acquires the lock of the LRU list that manages buffer pages -> **AVOIDABLE**
- **btr_cur_search_to_nth_level**: traverses an index tree, varies with the depth -> **INHERENT**
- **fil_flush()**: flush redo logs (WAL) -> **INHERENT**
  (can be mitigated with faster I/O devices)

<table>
<thead>
<tr>
<th>2-WH</th>
<th>Function</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-WH</td>
<td>buf_pool_mutex_enter</td>
<td>32.92%</td>
</tr>
<tr>
<td>2-WH</td>
<td>img_btr_cur_search_to_nth_level</td>
<td>8.3%</td>
</tr>
<tr>
<td>2-WH</td>
<td>fil_flush</td>
<td>5%</td>
</tr>
</tbody>
</table>
Case Studies - PostgreSQL

- **LWLockAcquireOrWait()**: acquires a single global lock (WALWriteLock) to ensure that only one txn. is flushing at a time
  -> **AVOIDABLE** (I/O acceleration or parallel logging)

- **ReleasePredicateLocks()**: releases predicate locks (for avoiding phantom problems)
  -> **INHERENT** (negligible)

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Percentage of Overall Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWLockAcquireOrWait</td>
<td>76.8%</td>
</tr>
<tr>
<td>ReleasePredicateLocks</td>
<td>6%</td>
</tr>
</tbody>
</table>
Case Studies - VoltDB

• VoltDB: an event-based system

• Each event waits in a queue before a worker thread is assigned

• **99.9%** of latency variance comes from the varying waiting time of the event queues -> **AVOIDABLE**
  
  (adjust # worker threads and control the queue size)
Mitigation Ideas

• MySQL
  • os_event_wait -> schedule txns. in a variance-aware manner (VATS)
  • buf_pool_mutex_enter -> update LRU list lazily (LLU)

• PostgreSQL
  • LWLockAcquireOrWait -> parallelize WAL

• VoltDB
  • Event queuing time -> adjust # worker threads
Mitigation Ideas

• MySQL
  • os_event_wait -> **schedule txns. in a variance-aware manner (VATS)**
  • buf_pool_mutex_enter -> update LRU list lazily (LLU)

• PostgreSQL
  • LWLockAcquireOrWait -> **parallelize WAL**

• VoltDB
  • Event queuing time -> **adjust # worker threads**
VATS vs FCFS - Example

- Protocol: Strict 2-Phase Locking + Wait-Die Deadlock Prevention

T₁
- write(X)
- write(Z)
- write(Y)
- commit()

T₂
- write(Z)
- write(Y)
- write(X)
- commit()

T₃
- write(Y)
- write(X)
- write(Z)
- commit()
VATS vs FCFS – FCFS (1/5)

**FCFS** (First-Come-First-Served): Grants locks to the txns. at the head

- $L_X$: $T_1$ is locked by $T_3$, then $T_2$ enters and aborts $T_3$.
- $L_Y$: $T_3$ is locked by $T_2$, then $T_1$ enters and aborts $T_3$.
- $L_Z$: $T_2$ is locked by $T_3$, then $T_1$ enters and aborts $T_3$.
FCFS (First-Come-First-Served): Grants locks to the txns. at the head

- $L_X$: $T_1$, $T_2$, $T_3$
- $L_Y$: $T_2$, $T_1$, $T_3$
- $L_Z$: $T_2$, $T_1$, $T_3$

 abort $T_2$
FCFS (First-Come-First-Served): Grants locks to the txns. at the head

\[ L_X \quad \text{commit } T_1 \]

\[ L_Y \]

\[ L_Z \]
FCFS (First-Come-First-Served): Grants locks to the txns. at the head

L_X

\[ \text{T}_3 \quad \text{T}_2 \]

L_Y

\[ \text{T}_3 \quad \text{T}_2 \]

L_Z

\[ \text{T}_3 \quad \text{T}_3 \]

commit T_3
VATs vs FCFS – FCFS (5/5)

**FCFS (First-Come-First-Served):** Grants locks to the txns. at the head

T_2

commit T_2
VATS vs FCFS – VATS (1/3)

**VATS:** Grants lock to the eldest txns.

- **L_x**
  - $T_1$
  - $T_3$
  - $T_2$

- **L_y**
  - $T_3$
  - $T_2$
  - $T_1$

- **L_z**
  - $T_2$
  - $T_1$
  - $T_3$

*commit $T_1$*
VATS vs FCFS – VATS (2/3)

**VATS**: Grants lock to the eldest txns.

\[ L_X \quad T_3 \quad T_2 \quad T_3 \]

\[ L_Y \quad T_3 \quad T_2 \quad T_3 \]

\[ L_Z \quad T_2 \quad T_3 \quad T_2 \]

commit \( T_2 \)
**VATS vs FCFS – VATS (3/3)**

**VATS**: Grants lock to the eldest txns.

\[
\begin{align*}
L_x & \quad \text{commit } T_3 \\
L_y & \\
L_z & 
\end{align*}
\]
VATS

• Loss function
  • Variance: not suited (adding a large delay to every txn. can achieve a near-zero variance, but significantly increase mean latency)
  • $L_p$ norm: indirectly reduce both mean and variance latencies ($l_i$: latency of txn. $i$, $p$: 2 in practice)
    \[ L_p = \left| \left| \langle l_1, \cdots, l_n \rangle \right| \right|_p = (\sum_{i=1}^{n} |l_i|^p)^{1/p} \]
  • $L_p$ norm of VATS scheduler is optimal against all schedulers
    (Theorem 1, proof in Section 5.3)
VATS – Experiment (1/2)

- Workload: TPC-C

RS: Randomized Scheduling

![Chart showing performance comparison between VATS and RS](chart.png)

<table>
<thead>
<tr>
<th>System</th>
<th>Name of the Identified Function</th>
<th>Original contribution to overall variance</th>
<th>Modification</th>
<th>Modified lines of code or config</th>
<th>Ratio of overall latency variances (Orig. / Modified)</th>
<th>Ratio of overall 99th latencies (Orig. / Modif.)</th>
<th>Ratio of overall mean latencies (Orig. / Modif.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>os_event_wait</td>
<td>59.2%</td>
<td>replace FCFS with VATS</td>
<td>189</td>
<td>5.6x</td>
<td>2.0x</td>
<td>6.3x</td>
</tr>
<tr>
<td>MySQL</td>
<td>buf_pool_mutex.enter</td>
<td>32.92%</td>
<td>replace mutex with spin lock</td>
<td>46</td>
<td>1.6x</td>
<td>1.4x</td>
<td>1.1x</td>
</tr>
<tr>
<td>MySQL</td>
<td>fil_flush</td>
<td>5%</td>
<td>parameter tuning</td>
<td>2</td>
<td>1.4x</td>
<td>1.2x</td>
<td>1.2x</td>
</tr>
<tr>
<td>Postgres</td>
<td>LWLockAcquireOrWait</td>
<td>76.8%</td>
<td>parallel logging</td>
<td>355</td>
<td>1.8x</td>
<td>1.3x</td>
<td>2.4x</td>
</tr>
<tr>
<td>VoltDB</td>
<td>[waiting in queue]</td>
<td>99.9%</td>
<td>add # of worker threads</td>
<td>1</td>
<td>2.6x</td>
<td>1.4x</td>
<td>5.7x</td>
</tr>
</tbody>
</table>
VATS – Experiment (2/2)

- SEATS [62]: airline ticketing system (highly contended)
- TATP [68]: caller location system ("not as contended as TPC-C")
- Epinions [48]: customer reviewing system
- YCSB [30]: no lock contentions

<table>
<thead>
<tr>
<th>Contended</th>
<th>Workload</th>
<th>Mean Latency</th>
<th>Variance</th>
<th>99th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TPCC</td>
<td>6.3x</td>
<td>5.6x</td>
<td>2.0x</td>
</tr>
<tr>
<td>SEATS</td>
<td>1.1x</td>
<td>1.3x</td>
<td>1.1x</td>
<td></td>
</tr>
<tr>
<td>TATP</td>
<td>1.2x</td>
<td>1.6x</td>
<td>1.3x</td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td>2.9x</td>
<td>2.8x</td>
<td>1.5x</td>
<td></td>
</tr>
<tr>
<td>No Contention</td>
<td>Epinions</td>
<td>1.4x</td>
<td>2.6x</td>
<td>1.0x</td>
</tr>
<tr>
<td></td>
<td>YCSB</td>
<td>1.0x</td>
<td>1.1x</td>
<td>1.1x</td>
</tr>
</tbody>
</table>
Mitigation Ideas

- MySQL
  - `os_event_wait` -> schedule txns. in a variance-aware manner (VATS)
  - `buf_pool_mutex_enter` -> update LRU list lazily (LLU)

- PostgreSQL
  - `LWLockAcquireOrWait` -> parallelize WAL

- VoltDB
  - Event queuing time -> adjust # worker threads
(Relaxed) LRU Buffer Pool in MySQL

- Consists of two sub-lists: young & old
(Relaxed) LRU Buffer Pool in MySQL

- No precise LRU ordering within the “young” sub-list

```c
buf_pool_mutex_enter();
buf_LRU_make_block_young();
buf_pool_mutex_exit();
```
(Relaxed) LRU Buffer Pool in MySQL

• No precise LRU ordering within the “young” sub-list

buf_pool_mutex_enter();
buf_LRU_make_block_young();
buf_pool_mutex_exit();
(Relaxed) LRU Buffer Pool in MySQL

- No precise LRU ordering within the "young" sub-list

```
buf_pool_mutex_enter();
buf_LRU_make_block_young();
buf_pool_mutex_exit();
```
Lazy LRU Update (LLU)

- The mutex can be a bottleneck when the working sets is larger than 5/8 of the buffer pool -> **Further relax LRU ordering**
  - Replace the mutex with a spin lock w/ timeout
  - If failed to acquire the lock within 0.01 ms, defer the update until successfully acquire the lock for another update
Lazy LRU Update (LLU) - Experiment

- Workload: TPC-C (2-WH)

<table>
<thead>
<tr>
<th>System</th>
<th>Name of the Identified Function</th>
<th>Original contribution to overall variance</th>
<th>Modification</th>
<th>Modified lines of code or config</th>
<th>Ratio of overall latency variances (Orig. / Modified)</th>
<th>Ratio of overall 99th latencies (Orig. / Modif.)</th>
<th>Ratio of overall mean latencies (Orig. / Modif.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>os_event_wait</td>
<td>59.2%</td>
<td>replace FCFS with VATS</td>
<td>189</td>
<td>5.6x</td>
<td>2.0x</td>
<td>6.3x</td>
</tr>
<tr>
<td>MySQL</td>
<td>buf_pool_mutex_enter</td>
<td>32.92%</td>
<td>replace mutex with spin lock</td>
<td>46</td>
<td>1.6x</td>
<td>1.4x</td>
<td>1.1x</td>
</tr>
<tr>
<td>MySQL</td>
<td>fil_flush</td>
<td>5%</td>
<td>parameter tuning</td>
<td>2</td>
<td>1.4x</td>
<td>1.2x</td>
<td>1.2x</td>
</tr>
<tr>
<td>Postgres</td>
<td>LWLockAcquireOrWait</td>
<td>76.8%</td>
<td>parallel logging</td>
<td>355</td>
<td>1.8x</td>
<td>1.3x</td>
<td>2.4x</td>
</tr>
<tr>
<td>VoltDB</td>
<td>[waiting in queue]</td>
<td>99.9%</td>
<td>add # of worker threads</td>
<td>1</td>
<td>2.6x</td>
<td>1.4x</td>
<td>5.7x</td>
</tr>
</tbody>
</table>
Mitigation Ideas

- MySQL
  - `os_event_wait` -> **schedule txns. in a variance-aware manner (VATS)**
  - `buf_pool_mutex_enter` -> **update LRU list lazily (LLU)**

- PostgreSQL
  - `LWLockAcquireOrWait` -> **parallelize WAL**

- VoltDB
  - Event queuing time -> **adjust # worker threads**
Simple Parallel WAL - Overview

- Uses two hard disks for storing two sets of logs
- Only acquires the global lock when both sets are in use
Simple Parallel WAL - Experiment

- Workload: TPC-C
Mitigation Ideas

• MySQL
  • os_event_wait -> **schedule txns. in a variance-aware manner (VATS)**
  • buf_pool_mutex_enter -> **update LRU list lazily (LLU)**

• PostgreSQL
  • LWLockAcquireOrWait -> **parallelize WAL**

• VoltDB
  • Event queuing time -> **adjust # worker threads**
Adjusting # Workers - Experiment

Default # threads: 2

<table>
<thead>
<tr>
<th>System</th>
<th>Name of the Identified Function</th>
<th>Original contribution to overall variance</th>
<th>Modification</th>
<th>Modified lines of code or config</th>
<th>Ratio of overall latency variances (Orig. / Modified)</th>
<th>Ratio of overall 99th percentiles (Orig. / Modif.)</th>
<th>Ratio of overall mean latencies (Orig. / Modif.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>os_event_wait</td>
<td>59.2%</td>
<td>replace FCFS with VATS</td>
<td>189</td>
<td>5.6x</td>
<td>2.0x</td>
<td>6.3x</td>
</tr>
<tr>
<td>MySQL</td>
<td>buf_pool_mutex_enter</td>
<td>32.92%</td>
<td>replace mutex with spin lock</td>
<td>46</td>
<td>1.6x</td>
<td>1.4x</td>
<td>1.1x</td>
</tr>
<tr>
<td>MySQL</td>
<td>fill_flush</td>
<td>5%</td>
<td>parameter tuning</td>
<td>2</td>
<td>1.4x</td>
<td>1.2x</td>
<td>1.2x</td>
</tr>
<tr>
<td>Postgres</td>
<td>LWLockAcquireOrWait</td>
<td>76.8%</td>
<td>parallel logging</td>
<td>355</td>
<td>1.8x</td>
<td>1.3x</td>
<td>2.4x</td>
</tr>
<tr>
<td>VoltDB</td>
<td>[waiting in queue]</td>
<td>99.9%</td>
<td>add # of worker threads</td>
<td>1</td>
<td>2.6x</td>
<td>1.4x</td>
<td>5.7x</td>
</tr>
</tbody>
</table>
Variance-Aware Tuning (MySQL)

- buffer pool size:
  - 33% (default), 66%, 100% of the entire DB size

- log flushing policies:
  - eager flush, lazy flush, lazy write
Variance-Aware Tuning (PostgreSQL)

- I/O block (log buffer) size: 8 (default), 16, 32, 64 KB
- logs may occupy only a small portion of a large block

![Graph showing the ratio of 4K to block size for mean, variance, and 99th percentile across different block sizes (8K, 16K, 32K, 64K).]
Real-World Adoption

- VATS has been adopted by MariaDB, and now is its default scheduling policy - https://github.com/MariaDB/server/pull/248

MDEV-11039 - Add new scheduling algorithm for reducing tail latencies (for 10.2) #248

Merged janlindstrom merged 19 commits into MariaDB:10.2 from senssz:10.2-vats on 24 Oct 2016

Contribution 16 Comits 19 Files changed 6

senssz commented on 23 Oct 2016 • edited

This branch introduces a new scheduling algorithm (Variance-Aware-Transaction-Scheduling, VATS) for the record lock manager of InnoDB based on MariaDB 10.2. Instead of using First-Come-First-Served (FCFS), the newly introduced algorithm prefers the eldest transaction. A configuration parameter (innodb_lock_schedule_algorithm) is introduced for users to choose between VATS and FCFS (the default one). We've extensively tested this algorithm in many workloads. The algorithm is very simple, and the changes are very local, but it significantly improves performance (in terms of average latency and throughput) and predictability (in terms of reduction of tail and quantile latencies) For more details, please refer to this paper http://arxiv.org/abs/1602.01871

Reviewers
janlindstrom

Assignees
janlindstrom

Labels
None yet
Summary

- **Performance predictability** is getting more important than ever before for modern (OLTP) workloads.

- **TProfiler** has identified major sources of performance unpredictability in MySQL, PostgreSQL, and VoltDB.

- The default FCFS scheduler in MySQL is one major source of performance unpredictability, and VATS scheduler successfully improves predictability, as well as mean latencies.