Analyzing Performance Characteristics of PostgreSQL and MariaDB on NVMeVirt

Juhee Han
Seoul National University
Seoul, Korea
juheehan@dbs.snu.ac.kr

Yoojin Choi
Seoul National University
Seoul, Korea
cyj@dbs.snu.ac.kr

Abstract

The NVMeVirt paper analyzes the implication of storage performance on database engine performance to promote the tunable performance of NVMeVirt. They perform analysis on two very popular database engines, MariaDB and PostgreSQL. The result shows that MariaDB is more efficient when the storage is slow, but PostgreSQL outperforms MariaDB as I/O bandwidth increases. Although this verifies that NVMeVirt can support advanced storage bandwidth configurations, the paper does not provide a clear explanation of why two database engines react very differently to the storage performance.

To understand why the above two database engines have different performance characteristics, we conduct a study of the database engine's internals. We focus on three major differences in Multi-version concurrency control (MVCC) implementations: version storage, garbage collection, and index management. We also evaluated each scheme's I/O overhead using OLTP workload. Our analysis identifies the reason why MariaDB outperforms PostgreSQL when the bandwidth is low.

1 Introduction

The NVMeVirt is a versatile software-defined virtual NVMe device. Because the NVMeVirt supports advanced storage configurations, it can be used for database engine analysis and allows us to estimate the performance of database engines on future storage devices. In NVMeVirt paper [10], the authors conducted an evaluation on PostgreSQL [6] and MariaDB [3] with OLTP workload using sysbench [11]. They measured various performance metrics while running the benchmark. The result indicates that MariaDB and PostgreSQL react differently to the storage performance.

MariaDB fully utilizes the I/O bandwidth up to 500 MiB/s. However, I/O bandwidth utilization remains around 600 MiB/s even when the storage device provides higher bandwidth. On the other hand, PostgreSQL fully utilizes the I/O bandwidth up to 1,000 MiB/s, and the performance is saturated at approximately 1,800 MiB/s. The I/O bandwidth affects both database engines' performance, but PostgreSQL is much more sensitive than MariaDB. From the evaluation, the authors conclude that PostgreSQL is more promising on modern storage devices, whereas MariaDB is more efficient when the storage is low. The result verifies the tunable performance of NVMeVirt, but the problem is that the paper does not provide a clear explanation of what features of database engine internals make such differences.

In this paper, we analyze the implication of storage performance on database engine performance focusing on *database engine internals*. We aim to provide a clear explanation of why PostgreSQL is more sensitive to I/O bandwidth than MariaDB.

The contributions of this work are as follows:

- We perform experiments for both OLTP and OLAP workloads and analyze the different performance characteristics (Section 2).
- We analyze what differences in database engine internals make PostgreSQL more sensitive to I/O bandwidth compared to MariaDB (Section 3).
- We evaluated different MVCC schemes using OLTP workloads (Section 4).

2 Evaluation

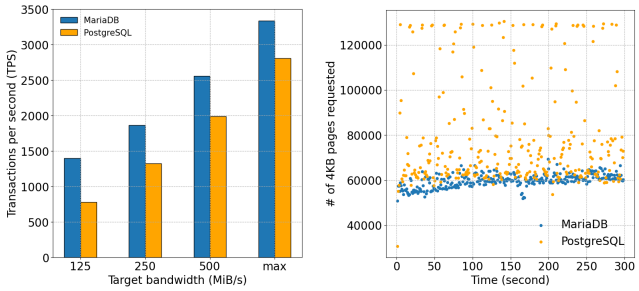
In this section, we present the evaluation results to demonstrate the performance characteristics of PostgreSQL and MariaDB on different bandwidths. We aim to answer the following questions:

- Is evaluation results from NVMeVirt paper reproducible in our environmental setup? (Section 2.2)
- How do PostgreSQL and MariaDB act differently to bandwidth when running OLAP workload? (Section 2.3)

2.1 Environmental Setup

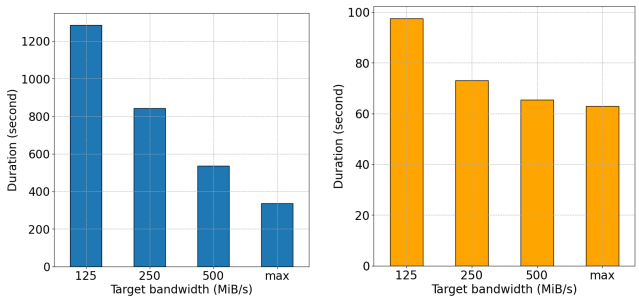
We used a Google Compute Engine instance running on Ubuntu 22.04 with kernel 6.1.14. The instance was equipped with one Intel Xeon processor operating at 2.20 GHz and has 24 cores and 128 GiB of memory in a NUMA configuration. For our evaluation, we dedicated 12 cores and 112 GiB of memory to NVMeVirt, and 12 cores and 16 GiB of memory were dedicated to the database engine.

To set up the environment as close as the environment used in the NVMeVirt paper, we established an NVMeVirt instance configured as an NVM SSD and set the I/O latency to a minimum. We configured the database instance with recommended settings obtained from optimization tools [5, 9].



(a) TPS vs. target bandwidth (b) I/O request

Figure 1. Performance comparison on MariaDB and PostgreSQL on various bandwidth configurations with OLTP workload.



(a) MariaDB (b) PostgreSQL

Figure 2. Performance comparison on MariaDB and PostgreSQL on various bandwidth configurations with OLAP workload.

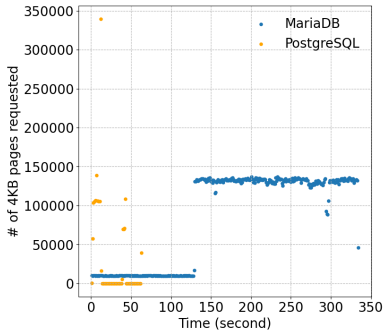


Figure 3. Number of 4KB pages requested by the MariaDB and PostgreSQL during OLAP query processing.

2.2 Online Transaction Processing Workload (OLTP)

We first compare the database engines' performance with the OLTP workload. We populated the database instance with sysbench to have ten tables of 25,000,000 bytes in size, a total size of approximately 60 GiB. Then we run the OLTP workload with sysbench with 12 threads for 30 minutes. We only report the first 5 minutes of trends since the performance became stable afterward.

Because the maximum bandwidth is bounded by the aggregated performance of the data processing units, the maximum target bandwidth in our environment is bounded by around 660 MiB/s. It's much smaller than that of NVMeVirt paper. Also, we set 12 CPU affinity to the database engine, whereas 36 cores were dedicated in the NVMeVirt paper. Due to these resource limitations, we could not produce the exact same result.

However, we still observed the three identical observations. Figure 1a compares processing performance measured in transactions per second (TPS) on various bandwidth limits. The first identical observation is that MariaDB outperforms PostgreSQL when the target bandwidth is low. In our evaluation, MariaDB outperforms PostgreSQL for every target bandwidth. Also, the performance gap becomes smaller as the target bandwidth increases. MariaDB outperforms PostgreSQL by 1.80x at a 125 MiB/s bandwidth limit and by 1.19x at maximum target bandwidth. This verifies that PostgreSQL is more sensitive to I/O bandwidth than MariaDB. Figure 1b shows the number of 4KB pages the database engine requests to the device for each second. The device is configured to have a target bandwidth of a maximum, which is 660 MiB/s. This shows that PostgreSQL requires a higher number of pages, resulting in a higher number of disk I/O operations, even when the TPS is lower than that of MariaDB.

2.3 Online Analytical Processing Workload (OLAP)

We also compare the database engines' performance with the OLAP workload. We populated the database instance with TPC-H [7] to have a 5 GiB dataset. Then we run the OLAP workload using 22 complex queries. We only report Query 18, which performs `JOIN` on multiple original tables.

The result is very different from the OLTP workload. Figure 2a and Figure 2b compare the duration spent processing the query. PostgreSQL outperforms MariaDB significantly for every target bandwidth. The performance gap becomes smaller as the target bandwidth increases. PostgreSQL outperforms MariaDB by 13.18x at a 125 MiB/s bandwidth limit and by 5.30x at the maximum target bandwidth. Figure 3 shows the number of 4KB pages requested over time. MariaDB requires a higher number of disk I/O compared to PostgreSQL.

2.4 Discussion

OLTP and OLAP workload evaluation show opposite results. With OLTP workload, PostgreSQL performs worse when the storage is slow but it significantly outperforms MariaDB for OLAP workload regardless of I/O bandwidth. This hints that the differences in OLTP and OLAP workload processing scheme make PostgreSQL more sensitive to I/O bandwidth than MariaDB.

OLTP workload is write-focused with simple operations, while the OLAP workload is read-focused with complex operations [12]. Also, the OLTP workload involves a large number of concurrent transactions. Database management systems (DBMSs) maintain the illusion of isolation and, at the same time, interleave the operations of concurrent transactions for better performance. To decide the proper interleaving of operations, DBMSs employ *concurrency control*. These are the main differences between OLTP and OLAP workload processing.

3 Characteristics of Database Engine

Based on the evaluation results, we have identified a potential explanation for the different performance characteristics: *Concurrency Control*. Concurrency control refers to the mechanisms employed by DBMSs to manage multiple transactions executing simultaneously, ensuring their isolation and preserving the consistency of the database [8]. These findings highlighted the fundamental distinction between OLTP and OLAP and the critical role of concurrency control in OLTP workload. As a result, we redirected our focus toward analyzing the factors responsible for the I/O overhead during concurrency control in PostgreSQL and MariaDB.

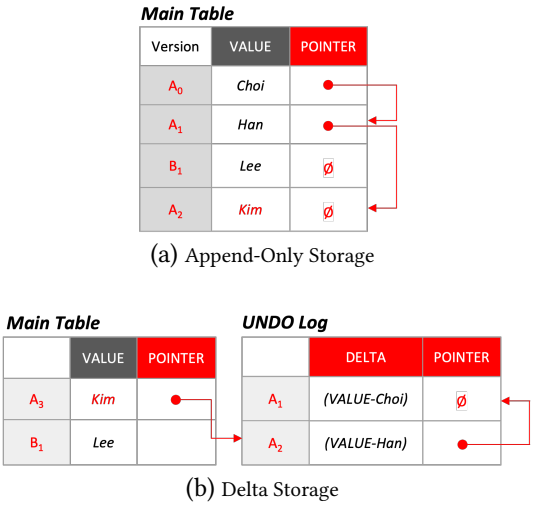
3.1 Multi-Version Concurrency Control

Both PostgreSQL and MariaDB employ a concurrency control method called multi-version concurrency control (MVCC). MVCC is widely adopted in modern relational DBMSs and encompasses concurrency control protocols, version storage, garbage collection, and index management [13]. However, there are notable differences in the design decisions of MariaDB and PostgreSQL regarding each of these elements.

3.2 Version Storage

Under the MVCC system, a new physical version of the tuple is created when a transaction updates a tuple. The storage scheme employed by the DBMS determines how these versions are stored and the information contained in each version. The DBMS utilizes the pointer field of tuples to establish a version chain. This version chain enables the DBMS to locate the specific version of a tuple that is visible to a transaction. We provide a detailed explanation of these storage schemes, focusing on the trade-offs involved in `UPDATE` operations.

Append-Only Storage PostgreSQL adopts the append-only approach as its version storage strategy [2]. In this strategy, all row versions of a table are stored in the same storage space. To modify an existing tuple, the DBMS follows a process where it allocates an empty slot from the table for the new version of the row. The content of the current version is then copied to the newly allocated slot, and the modifications are applied to this new version. Consequently,



(a) Append-Only Storage

(b) Delta Storage

Figure 4. PostgreSQL uses *append-only storage scheme* and MariaDB uses *delta storage scheme*.

there are two physical versions representing the same logical row.

To maintain the lineage of these versions for future reference, MVCC DBMSs create a *version chain* using a singly linked-list structure. The version chain is unidirectional to minimize storage and maintenance overhead. The DBMS must determine the order in which the versions are organized: newest-to-oldest (N2O) or oldest-to-newest (O2N). While most DBMSs, such as Oracle and MySQL, implement N2O, PostgreSQL employs the O2N approach. In the N2O order, each tuple version points to its previous version, and the head of the version chain always points to the latest version. On the other hand, in the O2N order, each tuple version points to its newer version, and the head represents the oldest tuple version. The O2N approach eliminates the need for the DBMS to update indexes to point to a newer version of the tuple every time it is modified. However, it may take longer for the DBMS to locate the latest version during query processing, potentially requiring the traversal of a lengthy version chain.

When a tuple is updated, the DBMS replicates all of its columns into the new version, regardless of whether the update affects a single column or all of them. This approach leads to significant data duplication and increased storage requirements. As a result, PostgreSQL requires more memory and disk space to store a database compared to other DBMSs.

Delta Storage MariaDB-InnoDB implements a more efficient method called delta storage for version storage. Instead of duplicating the entire tuple for a new version, it stores a compact delta representing the changes between the new and current versions, similar to a `git diff`. Consequently, when a query updates only a single column in a tuple with multiple

	Protocol	Version Storage	Garbage Collection	Index Management
PostgreSQL	MV2PL/SSI	Append-only (O2N)	Tuple-level (VAC)	Physical Pointers
MariaDB-InnoDB	MV2PL	Delta	Tuple-level (VAC)	Logical Pointers

Table 1. MVCC Implementations [13]

columns, the DBMS only stores a delta record containing the specific change. The DBMS manages the main versions of tuples in the primary table and a series of delta versions in a separate storage called the UNDO log in MariaDB.

In MariaDB, the current version of a tuple resides in the primary table. When updating an existing tuple, the DBMS acquires a contiguous space from the delta storage to create a new delta version. This delta version solely includes the original values of the modified attributes, rather than duplicating the entire tuple. Subsequently, the DBMS directly performs an in-place update to the master version in the primary table. This approach proves advantageous for `UPDATE` operations that modify only a subset of a tuple's attributes.

3.3 Garbage Collection

If a MVCC-based DBMS doesn't reclaim unnecessary versions, the system will eventually experience storage space issues. This leads to longer query execution times as the DBMS must navigate lengthy version chains. Therefore, the performance of MVCC DBMSs greatly relies on the effectiveness of its garbage collection (GC) mechanism in safely reclaiming space during transactions. During the GC process, the DBMS performs three important steps:

1. Detect expired versions of tuples.
2. Unlink these versions from their associated chains and indexes.
3. Reclaim the storage space occupied by these expired versions.

These steps are essential for maintaining the optimal functioning of the DBMS and ensuring transactional safety.

Both PostgreSQL and MariaDB employ tuple-level GC called *vacuum* as part of their MVCC systems. This approach is crucial for managing storage space effectively and improving query performance. The vacuum involves assessing the visibility of individual tuples within the database. While both databases utilize vacuum, their mechanisms differ significantly due to variations in their version storage methods.

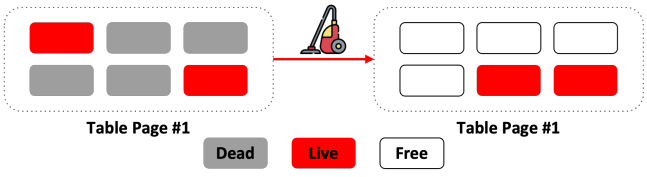


Figure 5. PostgreSQL vacuum process.

Vacuum Management Figure 5 illustrates how the vacuum process in PostgreSQL works. PostgreSQL creates copies of rows during updates. The next aspect to consider is handling *dead tuples*, older versions that need to be removed. In the original PostgreSQL version from the 1980s, dead tuples were not removed to support "time-travel" queries, allowing examination of past versions. However, this resulted in tables not shrinking when tuples were deleted and long version chains for frequently updated tuples, impacting query performance. To address this, PostgreSQL adds index entries to quickly access the correct version, but it increases index size, affecting performance.

In PostgreSQL, dead tuples occupy more space compared to delta versions. PostgreSQL uses the vacuum procedure to remove dead tuples. The procedure scans modified table pages since the last execution and identifies expired versions. An expired version is not visible to any active transaction and future transactions use the latest live version. Removing expired versions reclaims space for reuse, ensuring safety. PostgreSQL's *autovacuum* eventually removes dead tuples, but write-heavy workloads can cause accumulation faster than vacuuming, leading to continuous database growth.

MariaDB also uses a vacuum as its garbage collection mechanism. However, there is a difference in the areas where garbage collection occurs. Due to storing delta records in the undo log, MariaDB employs a *purge thread* that continuously scans the delta area in the background. When a transaction is committed, the purge thread removes the tuples stored in the undo log.

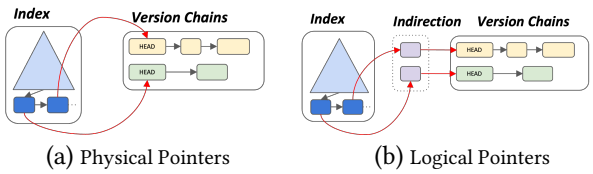
3.4 Index Management

MVCC-based DBMSs maintain a separation between versioning information and indexes. Index entries consist of key/value pairs, where the key represents the indexed attributes and the value is a pointer to the tuple. By following this pointer, the DBMS can access the tuple's version chain and find the version visible to a transaction. While false positive matches can occur, false negatives never happen with indexes.

Primary key indexes always point to the latest tuple version. In a delta scheme, the index points to the master version, reducing update frequency. In an append-only scheme, the index requires updates when new versions are created, such as when the primary key is modified.

Managing secondary indexes is more complex as both keys and pointers can change. Two approaches are used: physical pointers provide the exact tuple version's location as the

index value, while logical pointers use indirection to map to the physical location. Figure 6 illustrates how PostgreSQL and MariaDB use pointers for secondary index management.



(a) Physical Pointers (b) Logical Pointers

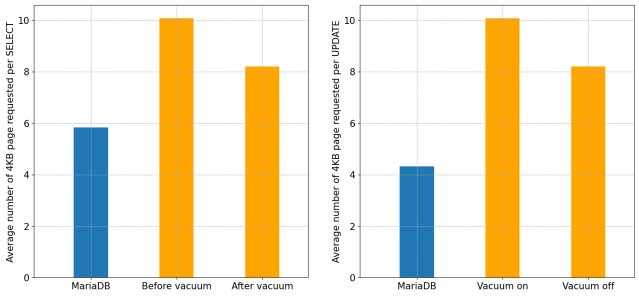
Figure 6. PostgreSQL uses physical pointers and MariaDB uses logical pointers for secondary index management.

Physical Pointers PostgreSQL utilizes the physical pointers method, storing the physical addresses of versions in index entries [2]. This approach is specifically designed for append-only storage, allowing direct referencing of versions within the same table through indexes. When a tuple in a table is updated, the DBMS inserts the newly created version into all secondary indexes. This enables the DBMS to search for a tuple using a secondary index without comparing the secondary key against all indexed versions.

Each row's header in PostgreSQL contains a tuple ID field that points to the next version or its own tuple ID if it is the latest version. Therefore, when a query requests the latest version of a row, the DBMS traverses the index, starting from the oldest version and following the pointer until it reaches the desired version. However, this complete traversal of the version chain can be inefficient, especially when most queries only require the latest version. Consequently, update queries can become slower due to increased workload. The DBMS incurs additional I/O operations to traverse each index and insert new entries, leading to lock/latch contention in both the index and internal data structures like the buffer pool's page table. Notably, PostgreSQL performs this maintenance work for all indexes in a table, even if some queries may never utilize them. These extra reads and writes can pose challenges.

To optimize disk I/O and minimize the need for multiple index entries and storing related versions across multiple pages, PostgreSQL employs a technique called *heap-only tuple* (HOT) updates [1]. When an update does not modify any columns referenced by the table's indexes and there is available space on the same data page as the old version, the DBMS creates a new copy of the tuple within the same disk page. This allows the index to still point to the old version, and queries retrieve the latest version by traversing the version chain. During regular operation, PostgreSQL further improves this process by removing old versions to prune the version chain.

Logical Pointers MariaDB adopts the logical pointers approach in index management [4]. The key distinction lies



(a) Read (b) Write

Figure 7. Comparing I/O overhead for different MVCC approaches.

in the architectural difference between the two. While PostgreSQL directly maps index records to on-disk locations, MariaDB employs a secondary structure. In MariaDB, secondary index records store a pointer to the primary key value instead of the on-disk row location, as done in PostgreSQL with the `ctid`. MariaDB implements an indirection layer that maps a tuple's identifier to the head of its version chain. This design choice eliminates the need to update all indexes of a table to point to a new physical location whenever a tuple is modified, even if the indexed attributes remain unchanged. Only the mapping entry needs to be updated. However, since the index does not directly point to the exact version, the DBMS traverses the version chain starting from the HEAD to locate the visible version.

4 Experimental Analysis

In this section, we present the experiment analysis to demonstrate the I/O overhead of PostgreSQL's MVCC scheme. For the experiment, we slightly modified the sysbench code and added a workload that only does `UPDATE` (a *write* operation). Also, we used the workload that only does `SELECT` (a *read* operation).

4.1 Version Chain Scan Overhead

To demonstrate the I/O overhead of scanning the version chain, we conducted an evaluation using the following procedures:

1. Set PostgreSQL's autovacuum configuration to off so that all the multi versions of updated tuples remain in the table space.
2. Run the workload that only does `UPDATE`.
3. Run the workload that only does `SELECT`.
4. Manually vacuum all the table spaces using `VACUUM` command in PostgreSQL so that all the logical tuples can have only one physical tuple.
5. Run the workload that only does `SELECT`.

MariaDB (more precisely, InnoDB) has no configuration that turns off the auto garbage collection. So we only run Step 2 and 3.

Figure 7a visualizes the average number of 4KB pages requested per `SELECT` query. Scanning a version chain requires approximately two more 4KB pages per read. Also, MariaDB requires approximately six 4KB pages per read which outperforms PostgreSQL even when the read happens after the vacuum. This shows that append-only storage results in higher I/O overhead for reads.

4.2 Garbage Collection Overhead

To demonstrate the I/O overhead of garbage collection, we conducted an evaluation. For PostgreSQL, we run the workload that only does `UPDATE` with autovacuum configuration on and off. For MariaDB, we also run the workload that only does `UPDATE`.

Figure 7b visualizes the average number of 4KB pages requested per `UPDATE` query. When autovacuum is on, PostgreSQL needs approximately two more 4KB pages per write. Also, MariaDB requires approximately four 4KB pages per write which outperforms PostgreSQL even when the autovacuum in PostgreSQL is off. This shows that, since the background vacuum process has to scan the table space that is interleaved with different versions of different tuples, it requires more disk I/O. Also, even when the autovacuum is off, the version chain needs to be scanned for writes since PostgreSQL uses the O2N scheme and results in higher number of I/O than the MariaDB.

5 Future Work

In our study, we could not observe the performance of the database engines with a bandwidth larger than 660 MiB/s. Therefore, conducting further analysis using higher resource capacities is important. This will allow us to better understand why PostgreSQL is a more promising database engine for future storage and MariaDB cannot fully utilizes the high I/O bandwidth. Also, it would be valuable to analyze the characteristics of OLAP workloads to investigate why PostgreSQL outperforms MariaDB in complex queries involving operations such as `JOIN` and `AGGREGATION`. This will provide insights into the specific features and optimizations the PostgreSQL uses.

6 Conclusion

We presented an analysis of database engine internals that explains why PostgreSQL utilizes the storage device more eagerly than MariaDB. We provide an explanation for different MVCC implementations that PostgreSQL and MariaDB uses. We also conducted the experimental analysis to evaluate the I/O overhead for MVCC design decisions.

References

[1] *Heap-only Tuple.* https://www.postgresql.org/docs/current/storage-hot.html

[2] *The Internals Of PostgreSQL.* https://www.interdb.jp/pg/

[3] *MariaDB.* https://mariadb.org

[4] *MySQL.* http://www.mysql.com

[5] *PGTune.* https://pgtune.leopard.in.ua

[6] *PostgreSQL.* https://www.postgresql.org

[7] *TPC-H.* https://www.tpc.org/tpch

[8] Naser Saleh Barghouti and Gail Elaine Kaiser. 1991. Concurrency Control in Advanced Database Applications. In *ACM Computing Surveys*, Vol. 23. Issue 3.

[9] Major Hayden. *MySQLTuner.* https://github.com/major/MySQLTuner-perl

[10] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. 2023. NVMeVirt: A Versatile Software-defined Virtual NVMe Device. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (USENIX FAST)*. Santa Clara, CA.

[11] Alexey Kopytov. *sysbench.* https://github.com/akopytov/sysbench

[12] Michael Stonebraker and Rick Cattell. 2011. 10 Rules for Scalable Performance in 'Simple Operation' Datastores. *Commun. ACM* 54, 6, 72–80.

[13] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *Proc. of the VLDB Endowment* 10, 7, 781–792.