To be considered for an index scan, an index must match one or more restriction clauses or join clauses from the query's condition, and match the query's `ORDER BY` condition. Filtering the indexes for the query is done by the `create_index_paths()` which is called for regular and now also for MOT tables, but not for previous foreign tables. The planning uses the FDW `GetForeignPaths()` function to get the plan for the query. Previous FDW is simply returning one general plan, while MOT considers the optional indexes, selects the one that will minimize the set of traversed rows, and adds it to the plan.

The query execution iteration is using the best index from the relation. It opens a cursor and iterates on the possibly relevant rows. Each row is inspected by the GaussDB envelope, and according to the query conditions, an update or delete is called for it. The FDW update or delete are called from the GaussDB `execUpdate()` or `execDelete()` if the relation is a foreign table.

3.2 Integrating MOT with HA

A database storage engine is an internal software component that a database server uses to store, read, update and delete data in the underlying memory and storage systems. Logging, checkpointing and recovery are not parts of the storage engine, especially as some transactions encompass multiple tables with different storage engines. As we still need to persist and replicate the data we use the high-availability facilities from the GaussDB envelope as follows:

- **Logging:** We persist write-ahead logging (WAL) records using the GaussDB's `XLOG` interface. By doing that we also gain GaussDB's replication capabilities that are using the same APIs. Although all data is maintained in volatile main-memory, MOT uses a persistent log to supply ACID transactions. GaussDB parallel logging and new SSD hardware minimize the overhead of persistence. The parallel logging in GaussDB is based on the implementation of [17] in PostgreSQL. This implementation uses multiple slots for WAL, allowing multiple backends to insert WAL records to the WAL buffers concurrently. Only at commit, after releasing its locks (early lock release), the transaction tries to take ownership of the log and flush all buffers to persistence. If a committer failed to take ownership, it waits for the buffers to be flushed by the current owner.
- **Checkpointing:** MOT checkpoint is enabled by registering a callback to GaussDB's checkpointer. Whenever a checkpoint is performed, MOT's checkpoint is called as well. MOT keeps the checkpoint's LSN (Log Sequence Number) in order to be aligned with GaussDB's recovery. The checkpointing algorithm is taken from [28] so it is asynchronous and is not stopping concurrent transactions. The only overhead is that if a checkpoint creation is in progress, an update to a row will keep the original version of the row until the checkpoint collection is done.
- **Recovery:** Upon startup, GaussDB first calls an MOT callback that recovers the MOT checkpoint and after that starts WAL recovery. During WAL recovery, MOT WAL records are replayed according to the checkpoint's LSN: older records if exists are not replayed. MOT checkpoint is recovered in parallel using multiple threads, each one reading a different data segment. This makes checkpoint recovery quite fast on many-core hardware but still it adds overhead compared to disk-based tables where only WAL records are replayed.

3.3 VACUUM and DROP

To complete the functionality of MOT we added support for VACUUM, DROP TABLE, and DROP INDEX. All three execute with exclusive table lock, i.e. without concurrent transactions on the table. The system VACUUM calls a new FDW function to perform the MOT vacuuming, while DROP was added to the `ValidateTableDef()`function.

3.3.1 Deleting Pools

Each index and table tracks all memory pools it uses. When dropping an index the metadata is removed and then the pools are deleted as one consecutive block. The MOT VACUUM is only doing compaction of the used memory, as memory reclamation is done continuously in the background by the epoch based garbage collector. To perform the compaction we switch the index or the table to new pools, traverse all the live data, delete each row and insert it using the new pools, and finally delete the pools as done for drop.

3.3.2 GC and Pools Deletion

Our epoch based GC is waiting until all transactions that were live at the time of row R deletion finish execution before reclaiming R and its related sentinels and index nodes into their designated pools. However, table or index dropping only take an exclusive lock on the table and then reclaims the pools directly, and not through the GC mechanism. As a result, a pool may be deleted when a buffer from GC that belongs to this pool is not yet reclaimed. When this buffer will be returned to the deleted pool, it will probably crash the system. Our solution is to mark each buffer in the GC with its origin pool, and before the pool P is going to be deleted, under table exclusive lock, recycle all deleted buffers that belong to P, so P will not be used after it has been deleted.

3.4 JIT for Query Acceleration

The FDW adapter for the MOT engine contains a lite execution path that employs Just-In-Time (JIT) compiled query execution using the LLVM compiler (JIT, for short). Current GaussDB contains two main forms of JIT query optimizations: (1) Accelerating expression evaluation (such as in WHERE clauses, target lists, aggregates and projections), and (2) Inlining small function invocations. These optimizations are partial (in the sense they do not optimize the entire interpreted operator tree or replace it altogether), and are targeted mostly at CPU-bound complex queries, typically seen in OLAP use cases. The approach adopted by MOT was to accelerate specific classes of queries that are known to be common in OLTP scenarios, such as point queries and simple range queries, rather than solving the general case. This approach allowed to generate LLVM code in MOT for entire queries.

It is noteworthy that current JIT optimization in GaussDB still executes queries in a pull-model (Volcano-style processing) using an interpreted operator tree. In contrast, MOT provides LLVM code for entire queries that qualify JIT optimization by MOT. This stands in stark contrast to existing JIT optimizations in GaussDB, which still keeps the interpreted operator tree for execution (albeit partially JIT

optimized), whereas MOT generates LLVM code for direct execution of the entire query over MOT tables, abandoning completely the interpreted operator model, and resulting in practically hand-written LLVM code generated for specific query execution. As explained in [10], although the data-centric model suggested there exhibits better performance than the traditional operator-centric model, it is still argued that hand-written plans in most practical cases exhibit the best performance.

Although the execution plans for the query classes chosen for JIT optimization in MOT result in rather small and simplified interpreted operator trees (lacking any pipeline breakers that force materialization points), the overhead of executing such operator trees is significantly higher (up to 30%) than executing hand-written queries over MOT Tables. This outcome results directly from the efficiency of MOT index access. When looking at the interpreted operator model in the context of GaussDB, we see that each leaf node represents table data access, while inner nodes represent the processing of the query executor. In the case of MOT, the relative part of table data access is much smaller than in the case of GaussDB, such that overall query execution spends relatively more time in the query executor. Therefore, replacing the query executor with a hand-written query results in a visible performance gain, even with rather simplified interpreted operator trees.

Another significant conceptual difference between MOT and current JIT optimization in GaussDB, is that LLVM code is generated only for prepared queries, during the PREPARE phase of the query, rather than during each execution of the query. This approach is inevitable due to the rather short runtime of OLTP queries (relative to OLAP scenarios), which cannot allow for code generation and relatively long query compilation time during each query execution. Still, this allows using more aggressive compiler optimizations without performance concerns during the PREPARE phase, resulting in more efficient code ready for repeated execution. On the other hand, this approach does not allow taking into consideration any runtime plan costs, in case there is more than one way to execute a query (e.g. JOIN queries). Considering the query classes chosen for JIT optimization, this was not a significant factor. If such cases become significant, two or more JIT plans can be generated during query preparation, as well as the code for estimating in runtime which plan is better.

Due to complexity of implementation, queries containing aggregate operators that require materialization (such as `GROUP BY` and `COUNT DISTINCT`) are not supported.

4. EXPERIMENTAL RESULTS

In this section we focus on the performance of MOT when it is a part of GaussDB. The OCC algorithm is already evaluated in numerous papers and we used a wide range of microbenchmarks to verify the adjustments from Section 2 maintain that performance. We will analyze MOT improvement to GaussDB performance on a Huawei ARM many-core server [14] and an Intel x86 based server, where all network is 10 Gb Ethernet.

4.1 Hardware

We tested our database on two separate hardware environments:

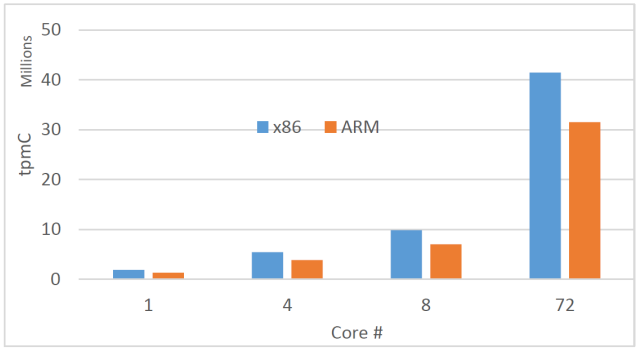


Figure 5: MOT standalone microbenchmark on ARM(96) vs. x86

- **Huawei ARM64:** TaiShan servers based on Huawei Kunpeng 920 – 4826 [14] processors with 48 cores and Kunpeng 920 – 6426 [12] with processors 64 cores. The server provides computing cores at a 2.6 GHz frequency, 1 TB of DRAM and 4TB Huawei ES3000 V5 series NVMe PCIe SSD with a bandwidth of 3.5 GB/s. We use three different configurations of the servers:
  - 2-sockets of 48 cores, adding to 96 cores.
  - 2-sockets of 64 cores, adding to 128 cores.
  - 4-sockets of 64 cores, adding to 256 cores.

  In the experiments we write ARM(n) where n denotes the number of cores in the server.

- **Intel x86:** A 2-socket machine, each socket contains a Intel(R) Xeon(R) Gold 6154 CPU, with 18 cores running at 3 GHz. Each core runs two hyperthreads, to a total of 72 hyperthreads. The system includes SAMSUNG MZILS400HEGR0D3 SSD of 0.4 TB.

4.2 Benchmark and Configuration

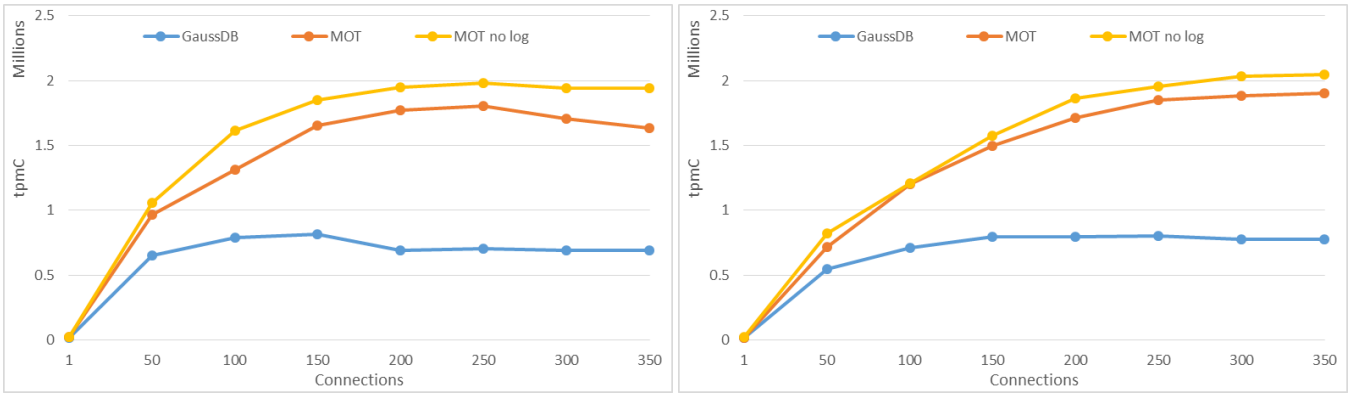
The clients in both cases are running on unloaded machines which are connected with 10Gb Ethernet to the server.

For all the following tests, the disk-based GaussDB had enough buffers to place all data and indexes in memory. This is fair as in this configuration both MOT and the original storage engine use the same amount of DRAM, and the root of performance difference is not IO latency.

4.3 Results

For Figure 5 we used hand written and hard coded TPC-C transactions which run to completion and use the MOT directly in the style of [37]. In this implementation we eliminate the impact of networking and durability. The benchmark emulates full TPC-C and runs a hand written version of the standard TPC-C consistency tests at the end to verify correctness. Being hard coded make its absolute performance irrelevant to the final product, but it can show the upper limit of MOT performance. We used it also to verify the GC and optimistic multi-index did not hurt performance.

This benchmark demonstrates that a core in ARM [12] is comparable to 0.75 hyperthread of x86. This result is consistent with the results in Figure 6 where 96 ARM cores performance is roughly comparable to 72 x86 hyperthreads,



(a) x86 **(b)** ARM(96)

Figure 6: Full TPC-C performance with scale factor of 500 warehouses and low contention on ARM and x86.

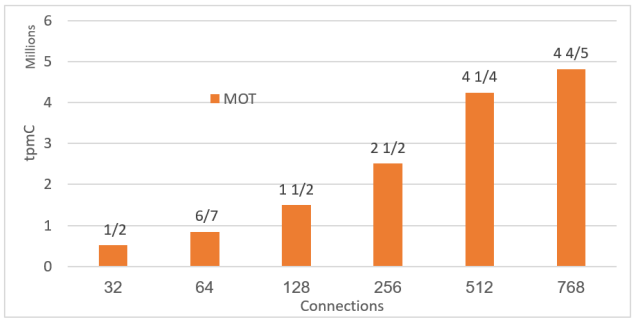


Figure 7: TPC-C with 1000 warehouses on ARM(256)

i.e., ARM is doing with N cores the work of 0.75N hyper-threads.

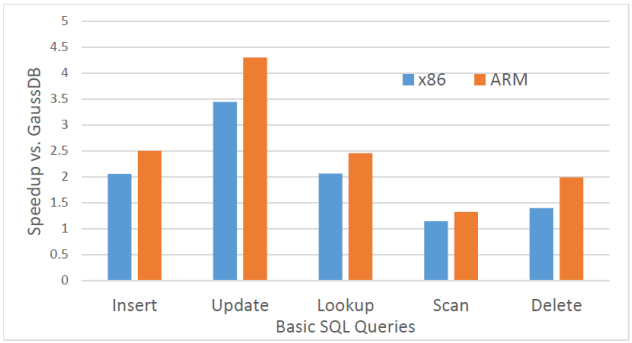


Figure 8: Speedup of Basic SQL queries

To understand the performance of GaussDB with MOT we start by looking at the performance of a single query on a single core, and later look in the performance of a single TPC-C transaction, and conclude by showing the scalability of MOT to multicore and its reaction to contention. MOT speedup of the basic SQL queries on a table with two integer fields which one of them is used for primary indexing, is presented in Figure 8, to understand the basic components of MOT performance. We are only interested in the SQL commands that access row data directly as that the area MOT is taking place. To measure the performance of range queries we measure the time of scanning 20 rows, so the initial Lookup operation to the first row is amortized and we actually compare the cursor `next` method. As seen in Figure 8 the lowest speedup is demonstrated by the scan operation, and the reason is that cursor increments are very light also in GaussDB, which is already tuned for analytical workloads.

Inserts which require chasing pointers through page offsets in the disk-based GaussDB, to locate B-Tree nodes and data, are more than 2x faster in MOT which is using Masstree highly optimized data structure. The ability to use cache friendly and lock-free data structures such as Masstree is an advantage of being in-memory. The same speedup is observed on a plain lookup. Actually an insert is a lookup with an additional store. A delete is again not far from GaussDB but while MOT is actually removing the row from the indexes and the table, GaussDB is only marking it for a future VACUUM operation which is not executed in this test. A point update is giving the highest speedup as it involves both a more efficient Lookup in MOT, and an inefficient locking operation in GaussDB. One interesting point is that our speedup is higher on ARM than on x86. The reason for this difference is that ARM relaxed memory model suffers more from the memory barriers which are introduced by the 2PL concurrency control and concurrent B-Tree implementation of GaussDB, and minimized by MOT OCC and Masstree index implementations.

The rest of this section is using the TPC-C benchmark, where the database and workload [1] were created by the BenchmarkSQL tool [24].

In Figure 9 we see a comparison of the TPC-C transactions speedup with and without JIT vs. GaussDB, on a single connection and with 300 connections.

StockLevel transaction does not show any improvement since it contains an aggregate operator (`COUNT DISTINCT` operator), which does not qualify for MOT JIT optimization (since it contains pipeline breaker requiring operator materialization). However all the other transactions gain about 30% from the JIT compilation.

[1]Like those of other in-memory systems (e.g. [39]), our results do not satisfy the TPC-C scaling rules for number of warehouses.

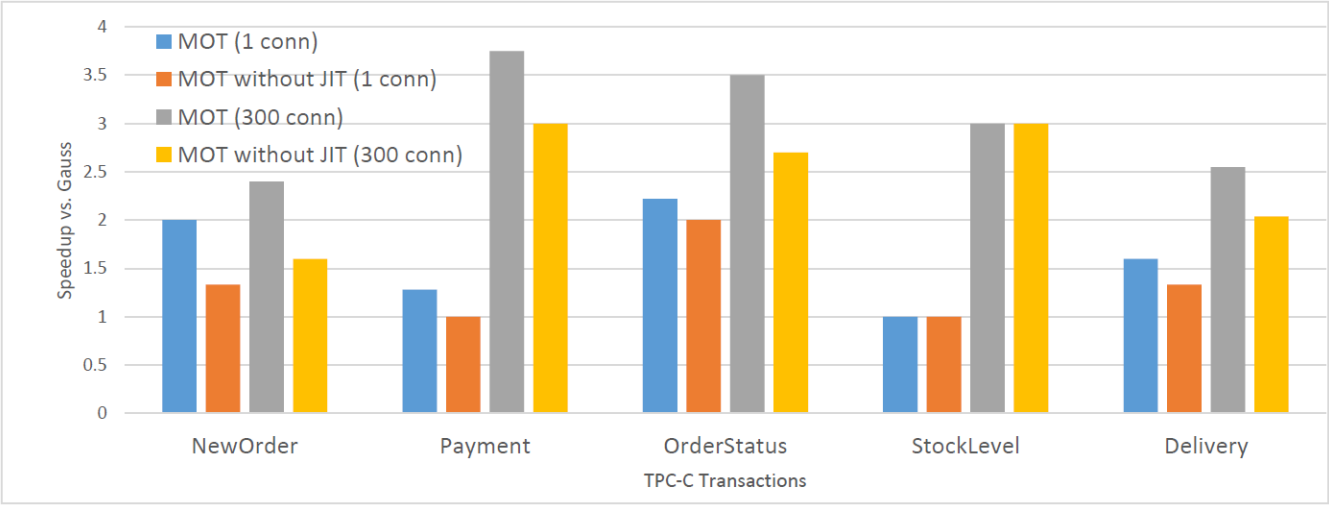


Figure 9: Speedup of TPC-C transactions (x86)

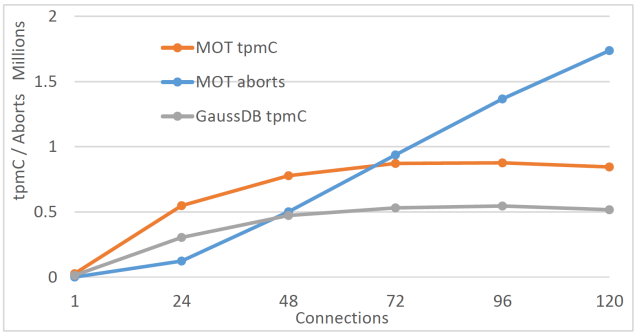


Figure 10: Full TPC-C performance on x86 with high contention

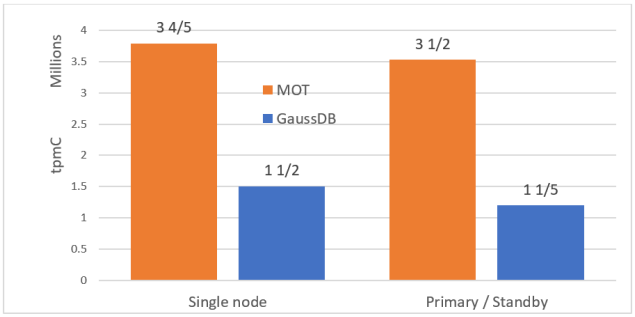


Figure 11: Replication overhead for TPC-C with 1000 warehouses and 812 clients on ARM(128)

All transactions are showing higher speedup with 300 connections than with 1 connection. One reason is that the lock-free index and OCC allow all transactions to make progress. Another reason is that according to the OCC algorithm readers are invisible, i.e. never write, which saves many, potentially NUMA, cache misses.

Figures 6a and 6b show that the overhead of logging is very low. The reason is that modern SSD and NVM hardware are very fast and have enough bandwidth and log insertion is non blocking. However, we do see performance degradation for the x86 server (Figure 6a), over 250 connections. We verified this is due to waiting for flush to complete, that is built into the parallel logging of GaussDB. In the ARM server (Figure 6b) we do not see the degradation as the SSD bandwidth there is higher.

Another insight is that GaussDB is scalable to the amount of cores in the hardware, and over subscription of multiple threads per core is not improving its performance while MOT keeps scaling. MOT better scalability comes again from the lock-free index and OCC which implies transactions can always make progress, and never wait for locks of slower or swapped concurrent transactions. To sharpen this insight, Figure 7 presents performance on a large ARM server with 256 cores, where MOT scales up to 768 connections.

In an OCC algorithm transactions never wait, but if there is contention, some work is aborted at commit time. This fact is demonstrated in Figure 10, where the scale factor is 12 warehouses, and parallelism grows to 120 threads, i.e. ten threads per warehouse. Contention of ten threads per warehouse is the upper limit of contention allowed by BenchmarkSQL. As the amount of serializable work is limited by the number of warehouses, the amount of aborted work grows with contention. However, the MOT keeps 2x speedup through the entire workload.

As demonstrated by Figure 11, in MOT the replication overhead of Primary/Standby high availability scenario is 7% on ARM(128) while for disk-based tables it is 20%.

5. RELATED WORK

We mentioned key related works about concurrency control[1, 38, 36, 2, 7, 8, 9, 23], indexing [4, 21, 25, 18] and multi-index tables [32], logging and recovery [17] and checkpointing [28], and JIT [10] in their relevant context in previous sections. In this section we add a few explicit related works that were not mentioned earlier.

Prototypes and Frameworks: The research on main memory OLTP engines produced many algorithms, as well as lightweight and extendable open source frameworks to compare them. In concurrency control a useful framework is DBX [37] which compares a wide range of algorithms, including 2PL, OCC, e.g. Silo, MVCC and simplified versions of H-Store and Hekaton in-memory databases. Although the micro benchmark is not realistic, it does stress the algorithms and shows the advantages and weaknesses of them. Silo type of OCC demonstrated the best performance on our many cores hardware on the workloads which we targeted.

Microsoft's Hekaton: SQL Server Hekaton [6] employs a commit verification protocol similar to Silo. One difference is that Hekaton uses the Bw-tree [22] for range-accessed indexes, which has a read-lock to prevent writers from accessing the row until the transaction that read the row commits. This protocol can reduce aborts, but instead even a read incurs an atomic write to shared memory. In other words, although both Silo and Hekaton employ OCC, Silo is more optimistic.

PostgreSQL Storage Engines: In [19] they describe a prototype of in-memory OLTP storage engine, implemented using FDW. However, their implementation is missing secondary indexes, DDL support and other essential components, which make it not production ready.

MOT: Fully optimistic, in-memory, production level storage engine, and integrated with PostgreSQL-based GaussDB, which adds to its industrial strength.

6. CONCLUSION

MOT is a new database storage engine designed for excellent OLTP throughput and scalability on large multicore machines with practically unbounded memory. All aspects of the MOT are optimized, including memory management, concurrency control and garbage collection. MOT is integrated into the fully-featured GaussDB SQL engine, to give users a seamless performance acceleration.

7. REFERENCES

[1] R. Appuswamy, A. Anadiotis, D. Porobic, M. Iman, and A. Ailamaki. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *PVLDB*, 11(2):121–134, 2017.

[2] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[3] L. Cai, J. Chen, J. Chen, Y. Chen, K. Chiang, M. A. Dimitrijevic, Y. Ding, Y. Dong, A. Ghazal, J. Hebert, K. Jagtiani, S. Lin, Y. Liu, D. Ni, C. Pei, J. Sun, L. Zhang, M. Zhang, and C. Zhu. Fusioninsight libra: Huawei's enterprise cloud data analytics platform. *PVLDB*, 11(12):1822–1834, 2018.

[4] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 181–190. Morgan Kaufmann, 2001.

[5] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[6] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254, 2013.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, pages 194–208, 2006.

[8] F. Faerber, A. Kemper, P. Larson, J. J. Levandoski, T. Neumann, and A. Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[9] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 237–246, 2008.

[10] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[11] Huawei. openGauss MOT Documentation. `https://opengauss.org/en/docs/1.0.0/docs/Developerguide/introducing-mot.html`.

[12] Huawei. TaiShan 2480 v2. `https://e.huawei.com/en/products/servers/taishan-server/taishan-2480-v2`.

[13] Huawei. GaussDB Distributed Database. `https://e.huawei.com/en/solutions/cloud-computing/big-data/gaussdb-distributed-database`, 2019.

[14] Huawei. TaiShan 5280 Server. `https://e.huawei.com/en/products/servers/taishan-server/taishan-5280`, 2019.

[15] Huawei. openGauss MOT open source. `https://gitee.com/opengauss/openGauss-server/tree/master/src/gausskernel/storage/mot`, 2020.

[16] Huawei. openGauss open source. `https://gitee.com/opengauss/openGauss-server`, 2020.

[17] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A scalable approach to logging. *Proc. VLDB Endow.*, 3(1):681–692, 2010.

[18] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 691–706, 2015.

[19] A. Korotkov. In-memory OLTP storage with persistence and transaction support. `https://www.postgresql.eu/events/pgconfeu2017`, 2017.

[20] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, and M. Grund. High-performance transaction processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[21] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

[22] J. J. Levandoski and S. Sengupta. The Bw-Tree: A Latch-Free B-Tree for Log-Structured Flash Storage. *IEEE Data Eng. Bull.*, 36(2):56–62, 2013.

[23] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 21–35, New York, NY, USA, 2017. ACM.

[24] D. Lussier. Benchmarksql 5.0, 2019.

[25] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196, 2012.

[26] open source. PostgreSQL. `https://www.postgresql.org/download/`.

[27] Oracle. Oracle timesten products and technologies, technical report., 2007.

[28] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1539–1551, 2016.

[29] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core OLTP under high contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1583–1598, 2016.

[30] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*, pages 221–228, 2007.

[31] N. Shamgunov. The MemSQL In-Memory Database System. In J. J. Levandoski and A. Pavlo, editors, *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics, IMDM 2014, Hangzhou, China, September 1, 2014*, 2014.

[32] G. Sheffi, G. Golan-Gueta, and E. Petrank. A scalable linearizable multi-index table. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 200–211, 2018.

[33] R. Sherkat, C. Florendo, M. Andrei, R. Blanco, A. Dragusanu, A. Pathak, P. Khadilkar, N. Kulkarni, C. Lemke, S. Seifert, S. Iyer, S. Gottapu, R. Schulze, C. Gottipati, N. Basak, Y. Wang, V. Kandiyanallur, S. Pendap, D. Gala, R. Almeida, and P. Ghosh. Native store extension for SAP HANA. *PVLDB*, 12(12):2047–2058, 2019.

[34] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, page 7, 2013.

[35] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[36] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[37] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[38] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1629–1642, 2016.

[39] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 465–477, 2014.