Industrial-Strength OLTP Using Main Memory and Many Cores

Conference Paper *in* Proceedings of the VLDB Endowment · August 2020

DOI: 10.14778/3415478.3415537

19 authors, including:

Hillel Avni
Huawei Technologies
26 PUBLICATIONS **182** CITATIONS
SEE PROFILE

Eliezer Levy
Huawei Technologies
31 PUBLICATIONS **2,091** CITATIONS
SEE PROFILE

Vladi Vexler
Huawei Technologies
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Industrial-Strength OLTP Using Main Memory and Many Cores

Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, Liran Mishali, Yeqin Mo, Nir Pachter, Dima Sivov, Vinoth Veeraraghavan, Vladi Vexler, Lei Wang, Peng Wang

firstname.lastname@huawei.com
Huawei Tel Aviv Research Center, Israel

ABSTRACT

GaussDB, and its open source version named openGauss, are Huawei's relational database management systems (RDBMS), featuring a primary disk-based storage engine. This paper presents a new storage engine for GaussDB that is optimized for main memory and many cores. We started from a research prototype which exploits the power of the hardware but is not useful for customers. This paper describes the details of turning this prototype to an industrial storage engine, including integration with GaussDB. Standard benchmarks show that the new engine provides more than 2.5x performance improvement to GaussDB for full TPC-C on Intel's x86 many-cores servers, as well as on Huawei TaiShan servers powered by ARM64-based Kunpeng CPUs.

PVLDB Reference Format:
Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, Liran Mishali, Yeqin Mo, Nir Pachter, Dima Sivov, Vinoth Veeraraghavan, Vladi Vexler, Lei Wang, Peng Wang. Industrial-Strength OLTP Using Main Memory and Many Cores. *PVLDB*, 13(12): 3099-3111, 2020.
DOI: https://doi.org/10.14778/3415478.3415537

1. INTRODUCTION

GaussDB [13] was announced in 2019 as a distributed relational database management system, followed by July 2020 open-source release of openGauss [16] as a community version of the closed-source GaussDB. It is a mixed OLAP and OLTP enterprise database system developed by Huawei database group. Starting from 2012 the first 5 years of research and development focused on a massively parallel OLAP database system. Since the launch in 2015 under the brand of FusionInsight MPPDB, LibrA [3] and GaussDB 200, the petabyte-scale OLAP platform has been adopted by many customers over the globe, including some of the world‘s largest financial institutes in China, and by 2016 reached the Gartner magic quadrant. In 2017 the development of GaussDB OLTP began, driven by demand of top customers for a next-generation OLTP DB system. Five key requirements were defined as follows: (1) transactional scale-out with a share-nothing architecture, (2) high efficiency of scale-up on multi-socket many-cores servers, (3) high-performance benefiting large memories; (4) high SQL coverage and interoperability; and (5) high availability. Driven by the above and influenced by emergence of new memory-optimized databases, such as Microsoft Hekaton [6], MemSQL [31], SAP HANA [20] and works in the art such as [36, 37, 38], we formalized a key research direction as follows: develop a high-performance memory-optimized storage engine, pluggable into GaussDB envelope, benefiting of its SQL and high availability (HA) capabilities. In the following our references to GaussDB refer only to GaussDB OLTP capabilities.

GaussDB was originally adapted from Postgres 9.2[26]. One of its main features, inherited already from OLAP MPPDB is thread based parallelism. This feature provided significant system performance. However, legacy synchronization and page-based data and index stayed in place.

Our goal was to remove the overheads and complexities involved in disk-based (e.g. pages awareness) and process-based (e.g. semaphore management) data base and exploit state of the art concurrent data structures and contemporary research on in-memory transaction management to improve GaussDB performance, while keeping the simple ACID semantics and the full SQL functionality.

This paper is about a storage engine that was created to harness the increasingly larger amount of main memory and processing cores for online transaction processing (OLTP). This new storage engine is integrated seamlessly in GaussDB. Our challenges were: (1) Adjusting state of the art in-memory concurrency-control research, which is usually incarnated in the shape of a minimal PoC prototype, to industrial workloads. (2) Integrating it into a full-featured SQL engine.

In the sequel, we use the term Memory-Optimized Tables (MOT) to denote both the new storage engine component, as well as the new feature and functionality provided by this engine. MOT is part of the open source openGauss server code [15] and documentation [11].

1.1 System Architecture

GaussDB is designed to scale linearly to hundreds of physical machines. Figure 1 illustrates the high level logical system architecture. Database data are partitioned and stored in many data nodes which fulfill local ACID properties. Cross-partition consistency is maintained by using two

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415537

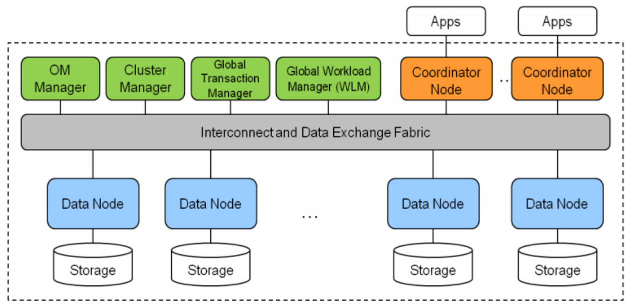


Figure 1: GaussDB System High-level Architecture

phase commit and global transaction management. If a table was created as a MOT, the SQL query processor will direct all access to that table to the MOT storage engine, and then the table will redirect all it's logging back to the data-node envelope. In recovery, the log records are taken back by the MOT table which replays them. Section 3 explains how MOT is incorporated into GaussDB, which does not have a modular storage engine. This way a transaction can use both MOT and disk-based tables, unlike other in-memory implementations, e.g. [6, 35] which replace the SQL engine and make it complicated to reuse the distributed transactions framework and perform transactions that mix in-memory and disk-based tables.

MySQL has a well-defined API for plugging new storage engines. However, it's logging mechanism is outside the storage engine and therefore its pluggable storage engines perform only ACI transactions without durability.

The performance of MOT is coming from choosing the most optimized indexing method and concurrency control, and bringing them to industrial level.

Another challenge we had is memory management and garbage collection in complex parallel and NUMA environments. To facilitate fast operation we allocate a designated memory pool for rows per table and for nodes per index. Each memory pool is composed from chunks of 2MB. We have API to allocate these chunks from local NUMA node, from pages coming from all nodes, or in round robin where every chunk is allocated on the next node. By default, pools of shared data are allocated in round robin to balance access while not splitting rows between different NUMA nodes. However, thread private memory is allocated from local node, as we verify a thread always operates in the same NUMA node.

1.2 Internal Architecture

One of our objectives was to build an OLTP system that would excel in using current and future many-core CPU architectures. In particular, linear scale-up with the number of cores was an explicit target. Based on our empirical experiments, the combination of the mature Masstree [25] lock-free implementation and our robust improvements to Silo [36], provided us exactly what we needed in that regard.

For index we compared state of the art solutions, e.g. [21, 25] and chose Masstree [25] as it demonstrated the best overall performance for point queries, iteration and modifications. Masstree is a combination of a Trie and a B+ tree, implemented to carefully exploit caching, prefetching, optimistic navigation, and fine-grained locking. It is optimized for high contention and adds various optimizations to its predecessors e.g. OLFIT [4]. However, Masstree index downside is higher memory consumption. While rows data consume the same memory size, the memory per row per each index, primary or secondary, is higher on average by 16 bytes: 29 bytes in the lock-based B-Tree used in disk-based tables vs. 45 bytes in MOT's Masstree.

The next step was to choose a concurrency control algorithm. To gain advantage from in-memory architecture we desired to maximize the speed of OLTP transactions. Although there have been recent suggestions for efficient in-memory MVCC [23], we chose to avoid the rapid garbage collection, and maintain only actual data. Another design choice of MOT is not to partition the data, as done in H-Store [8], because in realistic workloads transactions cross partition boundaries and performance rapidly degrades [36]. Some new designs use static and dynamic analysis to regulate parallelism [29], but this approach can introduce high latency and impractical constraints.

We split the single-version, shared-everything category of concurrency control algorithms, which we chose for MOT, to the following subcategories:

- **Optimistic concurrency control (OCC):** An OCC algorithm, e.g Silo [36] and TicToc [38], has three phases: The transaction reads records from the shared memory and performs all writes to a local, private copy of the records (the *read phase*). Later, the transaction performs a series of checks (the *validation phase*) to ensure consistency. After successful validation, the OCC system commits the transaction by making the changes usable by other transactions (the *write phase*). If the validation fails, the transaction is aborted and nothing is written. If two OCC transaction execute concurrently, they never wait for each other.

- **Encounter time locking (ETL):** In ETL, readers are optimistic, but writers lock the data which they access. As a result, writers from different ETL transactions see each other, and can decide to abort. It was verified empirically in [9] that ETL improve performance of OCC in two ways. First they detect conflicts early and often increase the transaction throughput because transactions do not perform useless work, as conflicts discovered at commit time, in general, cannot be solved without aborting at least one transaction. Second, encounter-time locking allows us to efficiently handle reads-after-writes (RAW) without requiring expensive or complex mechanisms.

- **Pessimistic concurrency control (2PL):** Lock a row at access time for read or for write, and release the lock at commit time. These algorithms require some deadlock avoidance scheme. The deadlock can be detected by calculating cycles in a wait-for graph or avoided by keeping time ordering in TSO [2] or by some back-off scheme. In 2PL algorithms, if one transaction is writing a row, no other transaction can access it, and if a row is being read, no transaction is allowed to write it.

As shown in [37, 1] OCC is the fastest option for most workloads. One reason is that when every core executes multiple threads a lock is likely to be held by a swapped thread, especially in interactive mode. In addition pessimistic algorithms involve deadlock detection which introduce overhead and usually use read-write locks which are less efficient than standard spin-locks. We chose Silo [36], as it was simpler than other existing options, e.g. TicToc [38], while maintaining the same performance for most workloads. ETL is sometimes faster than OCC, but it introduces spurious aborts which can confuse the user, in contrast to OCC which aborts only in commit.

For now, similarly to other leading main memory DBMSs in the industry [27] and in the art [39], the MOT table data capacity is either limited to a maximum available memory, or can exceed it by help of the memory page swapping by the operating system, though in such case performance may deteriorate. Recently several directions have been proposed to mitigate this issue by data reorganization [34], anti-caching [5] and tiering [33]. This is an important topic and is a future work for MOT as well.

Contributions: First we present and motivate MOT adjustments for industrial workloads. These adjustments include optimistic inserts and deletes to tables with multiple indexes, garbage collection in the presence of DDL SQL commands such as DROP and TRUNCATE table, and more. These enhancements do not improve performance, however, we verified with micro benchmarks that they do not downgrade the performance of the original OCC algorithm. Second we describe how MOT is integrated into the GaussDB and show its impact on GaussDB performance. After the integration of MOT to GaussDB side by side the disk-based storage engine, the SQL processing engine became the next performance bottleneck. To overcome this we added a specialized JIT compilation mechanism.

The rest of this paper is organized as follows. Section 2 presents the enhancements we added to the MOT research prototype to make it fit industrial workloads. Section 3 explains how we integrated MOT with GaussDB and how MOT is reusing GaussDB services for durability, recovery, checkpointing and SQL query processing. We show the performance of GaussDB with MOT in Section 4. Finally, we survey related work in Section 5 and conclude in Section 6.

2. ADJUSTING THE ENGINE

We describe the ways we extended MOT with support for optimistic insert to multi-index tables, non-unique indices, and important refinements in the concurrency control and commit protocol, to make it suitable as the underlying concurrency control mechanism for industrial use.

2.1 Indexing and Storage

The structure of a MOT table T with three rows and two indexes is depicted in Figure 2. The rectangles are data rows, and the indexes point to *sentinels* (the elliptic shapes) which point to the rows. The sentinels are inserted into unique indexes with a key and into non-unique indexes with a key + suffix, as explained in Section 2.1.2. The sentinels facilitate maintenance operations where the rows can be replaced without touching the index data structure. In addition, there are some flags and a reference count embedded in the sentinel to facilitate optimistic inserts as explained in Section 2.1.1.

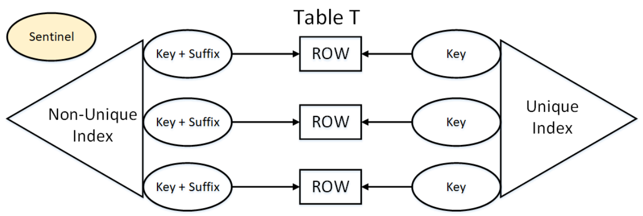


Figure 2: Structure of a Memory-Optimized Table

2.1.1 Optimistic inserts

The following is our solution to enable fully optimistic insertion to a table with multiple indexes. In section 2.2 we also discuss how a transaction reads its own updates and inserts, both in point, and range-queries.

2.1.1.1 Motivation.

Index contention may occur in the OCC designs that write to global indexes before entering the write phase. Upon creation of a new record, several OCC designs [18, 36] insert a new index entry to the table's indexes as well; the new record is locked to prevent concurrent transactions from reading it before it is committed. This early index update ensures that the new record satisfies unique-key invariants [36] and also simplifies making the index change visible to the current transaction. However, early index updates can create contention at indexes by frequently modifying their internal data structure even for the transactions that are eventually aborted. Furthermore, concurrent transactions that attempt to read the new record may be blocked for an extended period of time if the transaction that created the new record has a long read phase. A contention on inserts can also confuse an application as in OCC it expects to fail on inserts only for duplicate insert. This application may fail to retry the transaction that aborted due to a deadlock on inserts, and lose important data.

Another challenge is to make an optimistic insertion to a table with multiple indexes. Recently, [32] also offered a scalable insertion to a table with multiple indexes. However, their solution is working only on linked lists and does not support join or database transactional isolation, so it does not fit industrial use. In addition they use a theoretical extension to CAS which atomically modifies a reference and an associated boolean flag. As this extension does not exist in hardware it is not a practical solution for us.

2.1.1.2 Implementation Details.

In the original `Silo` an insert operation on key K works as follows. If K already mapped to a committed record, the insert fails and the transaction is aborted. Otherwise, a new record R is constructed in the `ABSENT` state, a mapping from K to R is added to the tree, and R is added to both the read-set and the write-set as if a regular put occurred. If an insert finds a row in the `ABSENT` state, it reuses it as if the row was inserted successfully by itself. Silo inserts the primary key as the record of the secondary indexes, which implies a significant overhead. With multiple unique indexes, as often the case in industrial workloads, pointing directly to the absent data row from the secondary index will produce unpredictable behavior.

Consider a table T with two unique indexes, I_1 and I_2, and a transaction X which tries to insert a row R into T. There are several scenarios which are not handled by the original **Silo** insert, e.g.:

- **Double Row:** X successfully inserted R in the primary index I_1, but when it tried to insert R in the unique secondary index I_2, it finds the key of R exists in I_2 in the `ABSENT` state, so X takes over the row which was linked from I_2. Now X has two different physical rows for R, one inserted in I_1 and another in I_2. When X will commit it will have different rows in each index for the same committed row which is unacceptable.

- **Self Inserted:** X needs to recognize a situation the insertion of R, or a scan, finds R in the `ABSENT` state in the index, but R was inserted by X itself. In this case the insert is duplicate and X needs to be aborted, or X needs to access the data of R in the scans.

To solve the above problems and allow direct reference from secondary indexes to data rows we developed the optimistic insertion algorithm depicted in Figure 3. At the beginning, a transaction X inserts a sentinel that is not connected to a row. The sentinel is inserted in the `ABSENT` state with reference count equal one. If the insert was successful, X maps the private row which it needs to insert in the transactional private insert-set part of the access-set. The access-set is implemented by a key-value map, and as a key for the row we use the pointer to the sentinel. Note that if a table has multiple indexes, the row will be mapped multiple times in the access-set with different keys, which are pointers to sentinels in different indexes. In the commit phase, after all sentinels of X insertions are locked, the sentinels are linked to the private inserted row, which becomes public. This way the same row is linked from all indexes and the **Double Row** problem is resolved.

If the insert failed, and there is a non-absent sentinel in the index, we got a duplicate insert and we abort the transaction. However, if the sentinel is in the `ABSENT` state we increment its reference count and try to map it in the access-set. If mapping failed, it means we already have this sentinel as a key in the access set. The only way this could happen is if the current transaction already tried to insert the same unique-key, so again, we abort the transaction due to a (self) duplicate insertion. During a scan, if an `ABSENT` sentinel is encountered but it maps a row in X access-set, the mapped row is used, as it was inserted previously by X. This solves the **Self Inserted** problem.

Another problem that is created by the optimistic inserts with multiple unique indexes is the cleanup after aborts. If transaction X aborts, it needs to decide from which indexes it can remove R, i.e., check if there are concurrent uncommitted transactions that reference this index entry. To address the problem we added a reference count to the sentinel which is incremented by every concurrent insert, and decremented by the cleanup after abort of an inserting transaction. Note that at most one insert will survive and the others will abort due to a conflict with the successful transaction. However, there is also a chance that all concurrent transactions inserting

Algorithm 1 Sentinels reference count update

```
1: function REFCOUNTUPDATE($sentinels$, $operationop$)
2:     status ← false
3:     rc ← done
4:     while $success = false \wedge s.refcount \neq 0$ do
5:         c ← s.refcount
6:         if op = INSERT then
7:             $next\_c = c + 1$
8:         else
9:             ▷ Cleanup after insert abort
10:            $next\_c = c - 1$
11:        end if
12:        success ← CAS(&$s.refcount$, $c$, $next\_c$)
13:    end while
14:    if success = false then
15:        rc ← retry_insert
16:        ▷ insert operation: s is removed, retry
17:    else
18:        if $next\_c = 0$ then
19:            ▷ cleanup after abort: remove s
20:            rc ← index_delete
21:        end if
22:    end if
23:    return rc
24: end function
```

R will abort, and then the last one will need to delete the sentinels of R from the indexes.

As seen in Figure 3, the transaction tries to insert the sentinel with reference count equal one. If the insert was not successful and sentinel is not committed, i.e. the sentinel is in the `ABSENT` state, we call the `refcountUpdate()` function from Algorithm 1 with the acquired sentinel and the `INSERT` operation. The reference count incrementation fails only if a concurrent cleanup is deleting the sentinel from the index, and in this case the function returns *retry_insert* and the insert is retried. Later, if the transaction aborts, the cleanup after abort calls `refcountUpdate()` with the same sentinel to try to decrement the reference count. If the reference count reached 0, the function returns *index_delete* and the cleanup deletes the sentinel from the index. This is the way we make sure an abort is deleting all the sentinels it acquired that are unused.

2.1.2 Non-unique indexes

A non-unique index may contain multiple rows with the same key. Non-unique indexes are used solely to improve query performance by maintaining a sorted order of data values that are used frequently. For example, a database may use a non-unique index to group all people from the same family. However, the Masstree data structure implementation does not allow mapping multiple objects to the same key. Our solution to enable the creation of non-unique indexes, as shown in Figure 2, is to add a symmetry breaking suffix to the key which maps the row. The suffix we use is the pointer to the row itself, which has a constant size of 8 bytes and a value that is unique to the row. When inserting to a non-unique index, the insertion of the sentinel always succeeds, so we can use the row which the executing transaction allocated.

When searching the non-unique secondary index we use the required key, e.g. the family name. The full concatenated

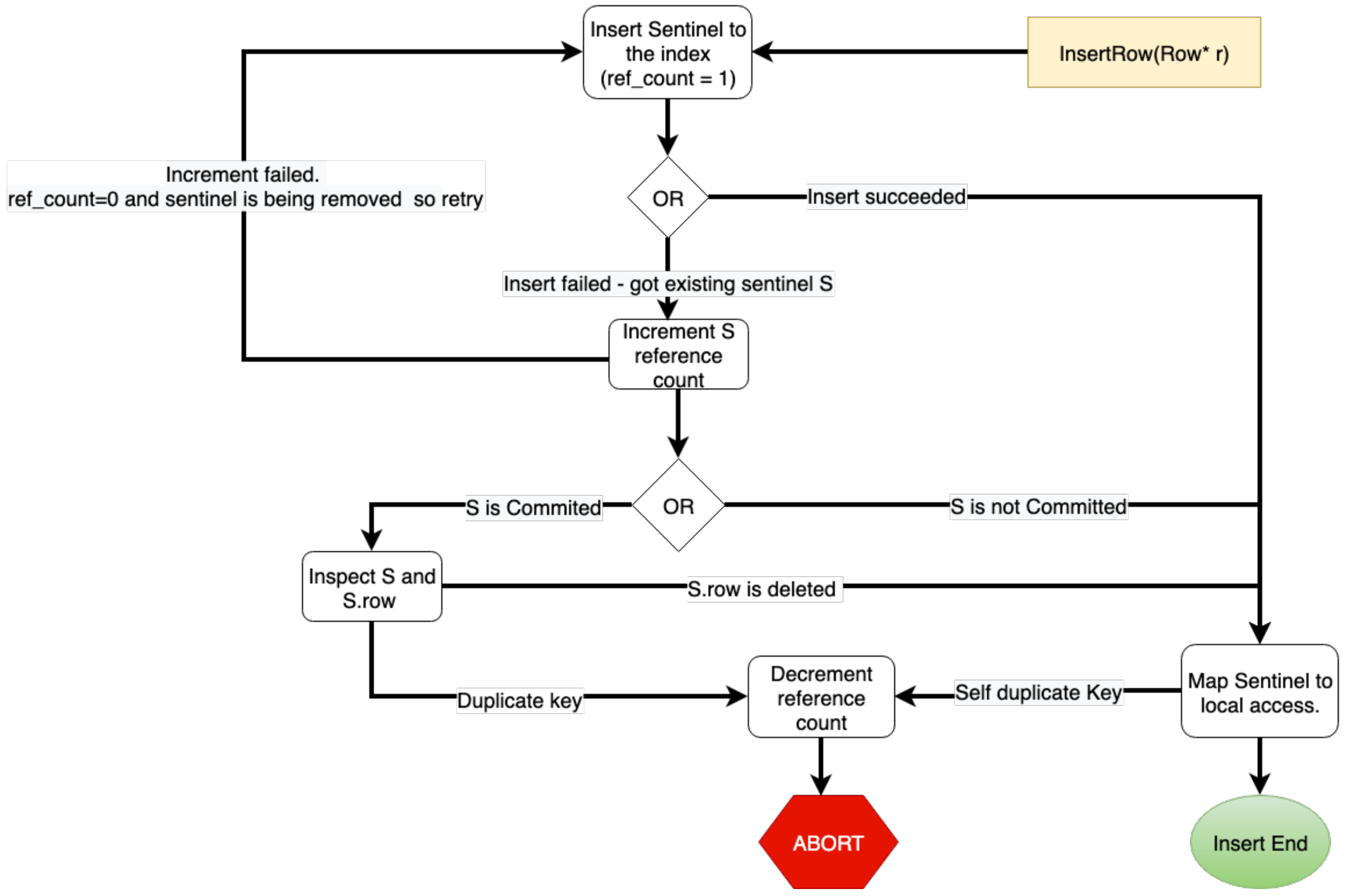

Insert Sentinel to the index (ref_count = 1)
InsertRow(Row* r)
Increment failed. ref_count=0 and sentinel is being removed so retry
OR
Insert succeeded
Insert failed - got existing sentinel S
Increment S reference count
S is Commited
OR
S is not Committed
Inspect S and S.row
S.row is deleted
Duplicate key
Decrement reference count
Self duplicate Key
Map Sentinel to local access.
ABORT
Insert End

Figure 3: Optimistic insert flow chart

key is used only for inserts and deletes. Insert and delete operations are always getting a row as a parameter, so it is possible to create the full key and use it in the execution of the deletion or the insertion of the specific row to the index.

2.2 Concurrency Control

In OCC, an update is buffered until commit, and if a scan encounters a row it will find there the last committed value, and not the one written by the current transaction. In the open source implementations of `Silo` [36], [37], if a row R is updated during a transaction T, and later read by T, T will not see its own update. This is a known problem from optimistic software transactional memory [7], which is called the read after write hazard. The known solutions for this problem is to either scan the write-set per read, which can cripple the performance of the OCC, or to use encounter time locking [30], i.e. lock and update rows at encounter and abort a reader that sees a location locked by another transaction.

Our solution was to create the state machine, and keep one access-set of all read, updated, inserted and deleted rows. The set is keyed by the pointer to the original row or sentinel, so once it is encountered it can be found in the access-set. The value for the key is the private copy of the row and the state of the row in the transaction. If a row is found to be in the *updated* (`WR`), *inserted*(`INS`) or *deleted* (`DEL`) states the relevant value, i.e. the local copy of the row is returned to the user. The states are maintained according to the state machine, where a row starts in the invalid (`INV`) state, and can go to read (`RD`), updated (`WR`), inserted (`INS`) and (`DEL`) states with the relative operations. The default and common isolation level in GaussDB is read-committed. Putting it in simple words, read-committed is an isolation level that guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data; data is free to change after it is read. When working in this mode, the access-set does not keep rows in `RD` state.

The *inserted* (`INS`) state is a special case. As explained in Section 2.1.1 only the sentinels are inserted in the index, and the row is connected only upon a successful commit. As a result, when a scan encounters an uncommitted insert the row pointer is not available. This is the reason we map the inserted rows with the pointer to the sentinel as the key, and the access itself includes the private inserted row. For inserts the private copy will be connected to the sentinels at commit and become public version, so during the transaction there is no "original" row in the access-set. If an uncommitted sentinel is encountered, i.e. a sentinel with an absent, `NULL` row, the sentinel is used as a key for a search in the access set.

This search either returns the private row, if it was inserted by the current transaction or skips this row, as it was inserted by a concurrent uncommitted transaction.

The commit protocol with the optimistic inserts is shown in Algorithm 2. As the access-set is organized in an ordered map such as `std:map`, when we extract the set of rows which need to be updated in lines 5 and 12 into an array the array is sorted. It is sorted by pointer values, which implies a global order on all rows and sentinels. Later we use this order to prevent deadlocks in locking. In line 5 we extract the rows which were deleted or updated and exist in the database, while in line 12 we extract the private inserted rows. The private rows are pointed from possibly more than one index and might be stored in the access set multiple times. However, we extract the row only from the primary sentinel which always exists exactly once in every table. The data of this row is not overwritten, as in line 55 they become public. In line 17 the read-set is collected. It is an empty operation in the read-committed isolation level.

In line 22 we lock the write-set and in line 26 we lock all the sentinels in the inserts-set. We remind the reader that the arrays of the write-set and the insert-set were extracted from the access-set which is an ordered map, and therefore they are sorted. This is important as the sorting avoids deadlocks and therefore we can wait until the location is locked. After all insert-set and write-set are locked, in line 30, we validate the write-set and the read-set (empty in read-committed) are consistent, and then in line 35 we validate the sentinels are still absent, and were not committed. If they were committed we abort the transaction.

After validation is successful, in line 42 we write the updated data. Then in line 46 we connect the private inserted rows to the inserted sentinels, and in line 47 we set the row to not absent, i.e. committed. Right after this we release all the locks from the sentinels in line 51 and then release the locks from the rows in line 55 and set the new version for both old and inserted rows.

For readability we simplified the commit pseudo code, and omitted logging and aborts. In the real code, logging is flushed after locking and locks are released also after an abort.

Algorithm 2 Commit Protocol for MOT

```
 1: function MOTValidateCommit(T)
 2:     for $ac \in access\_set$ do
 3:         if $ac.state = \mathtt{WR} \lor ac.state = \mathtt{DEL}$ then
 4:             $write\_set[ws\_size++] = ac$
 5:             ▷ collect the rows needed to be updated
 6:         end if
 7:         if $ac.state = \mathtt{INS}$ then
 8:             $insert\_set[is\_size++] = ac$
 9:             if $ac.type = primary\_sentinel$ then
10:                 $write\_set[ws\_size++] = ac$
11:                 ▷ An inserted row is added to $write\_set$
12:                 ▷ Once, per primary sentinel
13:             end if
14:         end if
15:         if $ac.state = \mathtt{RD}$ then
16:             $read\_set[rs\_size++] = ac$
17:             ▷ No read-set for read-committed
18:         end if
19:     end for
20:     for $e \in write\_set$ do
21:         $ac.row.lock()$
22:         ▷ Lock updated rows
23:     end for
24:     for $e \in insert\_set$ do
25:         $ac.sentinel.lock()$
26:         ▷ Lock inserted sentinels
27:     end for
28:     for $ac \in write\_set \bigcup read\_set$ do
29:         if not $valid(ac.row)$ then
30:             ▷ If isolation level is above read-committed
31:             abort
32:         end if
33:     end for
34:     for $ac \in insert\_set$ do
35:         if not $absent(ac.sentinel)$ then
36:             abort
37:             ▷ If the sentinel is already committed - abort
38:         end if
39:     end for
40:     for $ac \in write\_set$ do
41:         $ac.write()$
42:         ▷ Write updated and inserted rows
43:     end for
44:     for $ac \in insert\_set$ do
45:         $ac.sentinel.row = ac.row$
46:         ▷ connect inserted rows
47:         $ac.sentinel.absent = false$
48:     end for
49:     for $e \in insert\_set$ do
50:         $ac.sentinel.unlock()$
51:         ▷ Release locks of inserted sentinels
52:     end for
53:     for $e \in write\_set$ do
54:         $ac.row.unlock()$
55:         ▷ Free locks and set the version of updated rows
56:     end for
57: end function
```

3. INTEGRATION WITH GAUSSDB

GaussDB is based on PostgreSQL, which does not have a built in storage engine adapter such as MySQL handlerton. GaussDB contains more than two million lines of code, written during three decades, and often lacks in modular structure. To make the integration feasible we decided to try to extend GaussDB foreign data wrapper (FDW), and use it for our MOT storage engine. In Section 3.1 we present how tables and indexes meta data and data are created and used with FDW. Section 3.2 describes how MOT acquires high availability capabilities from the GaussDB infrastructure through FDW, and Section 3.3 explains how we perform efficient memory reclamation.

The history of FDW began when SQL/MED came out as part of the ANSI SQL standard specification in 2003. MED stands for ”Management of External Data”. By definition, ”external data” is the data that the DBMS is able to access but does not manage. Foreign tables are external data sources, presented as relations. FDW for foreign tables was introduced in PostgreSQL 9.1 and has been receiving improvements ever since.

GaussDB FDW, as taken from PostgreSQL, is intended to be used to access data which is stored in external database

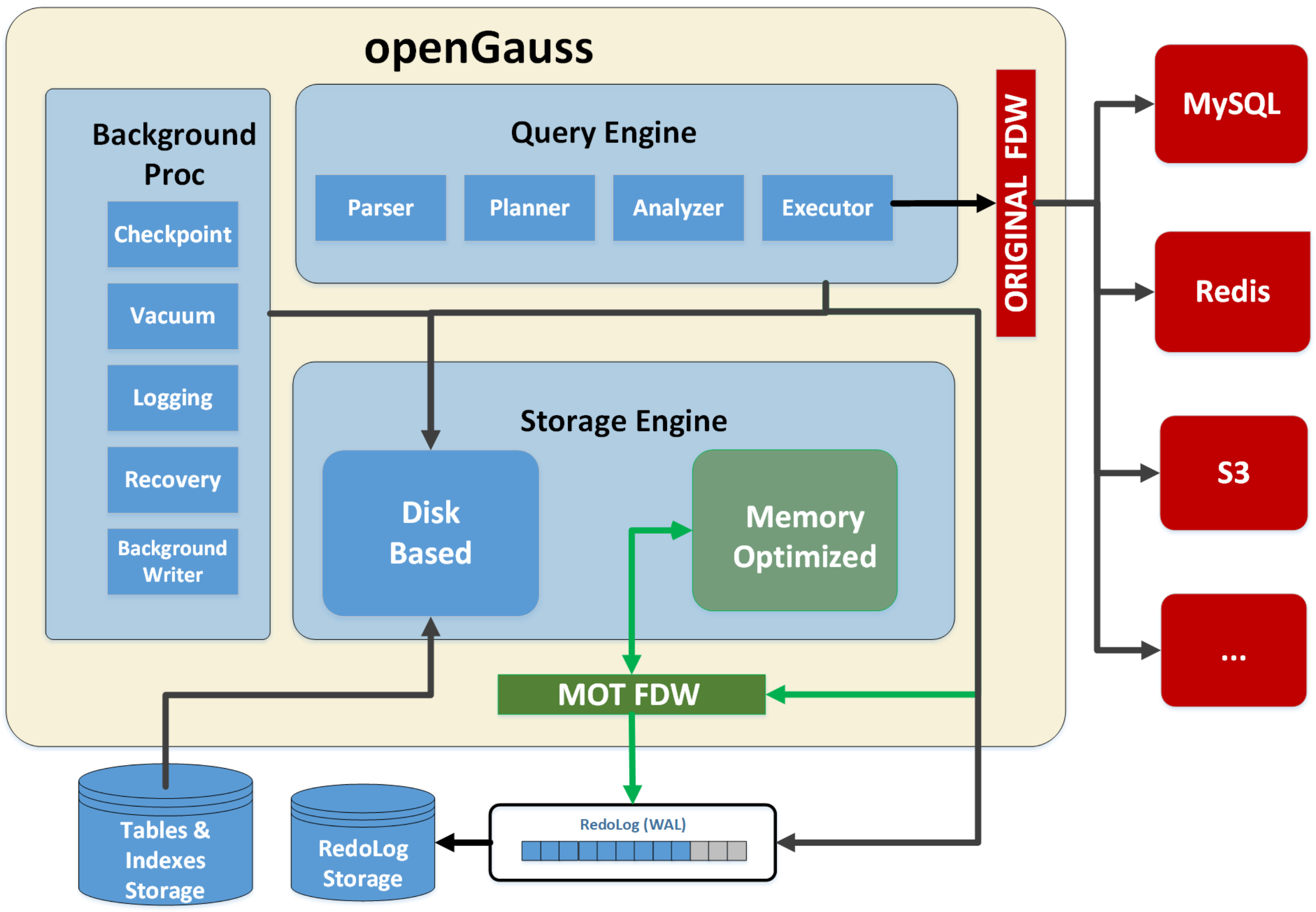


Figure 4: MOT in GaussDB

servers, and not managed by the local database. However, MOT storage engine is embedded in GaussDB, and managed by it. The MOT module is not independent, it is controlled by the GaussDB planner and executor, gets logging and checkpointing services from the GaussDB, and participates in the recovery process of the GaussDB. As shown in Figure 4, the MOT FDW is connected to more database functionality than standard FDW. Specifically, MOT reuses GaussDB index creation, and exploits logging, checkpointing and recovery from GaussDB.

To prepare for remote access using GaussDB FDW, a foreign table is created using CREATE FOREIGN TABLE, for each remote table needed to be accessed. For standard FDW the columns of the foreign table must match the referenced remote table. This is implied by the assumption that the foreign table is already created by the remote data source and the local table is only linking to that external entity. For MOT, the FDW needed the ability to trigger the creation of the table itself and its related indexes. Later, access to the indexes is also required for planning and execution. As we wanted MOT to work in coordination with GaussDB native tables, we reused GaussDB logging, replication and checkpointing mechanisms, to provide consistency for cross-table transactions through failures. In this case, the MOT sometimes initiates calls to GaussDB functionality through the FDW layer. Such calls were never required previous to MOT introduction.

3.1 Tables and Indexes Created and Used

The function which creates an FDW tables is named `ValidateTableDef`. As implied by its name, it was meant to only validate the foreign tables, which are created externally. In the previous FDW implementations, e.g. for external PostgreSQL and file servers, this function processed only table creation and, as suggested by its name, only validated the table references a foreign table. In MOT FDW, `ValidateTableDef` actually creates the specified table, and in addition it handles index creation on that relation and `DROP TABLE` and `DROP INDEX`, as well as `VACUUM` and `ALTER TABLE` which were not previously supported in FDW.

3.1.1 Index Usage for Planning and Execution

Query life breaks into two phases, the planning and the execution. During execution a query iterates on the relevant table rows, and performs some work, e.g., update or delete, per iteration. An insert is a special case where the table adds the row to all indexes and no scanning is required. In the planning phase, which may take place once per multiple executions, the best index for the scan is chosen.

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:

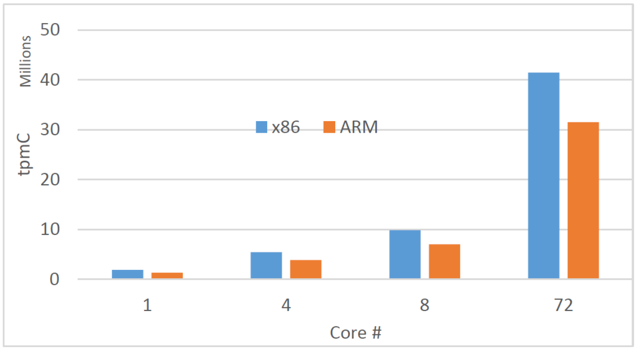


Figure 5: MOT standalone microbenchmark on ARM(96) vs. x86

- **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

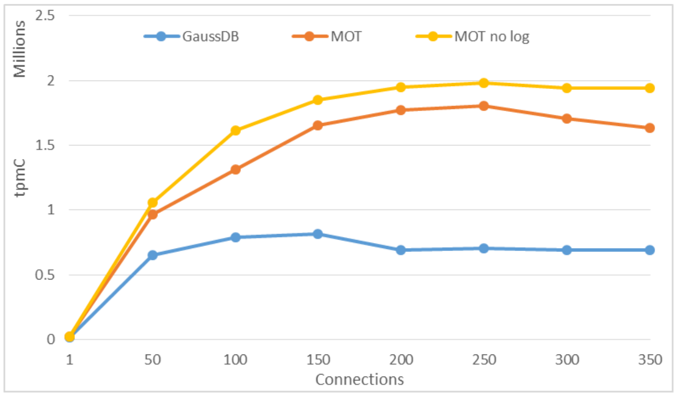
The clients in both cases are running on unloaded machines which are connected with 10Gb Ethernet to the server.

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.

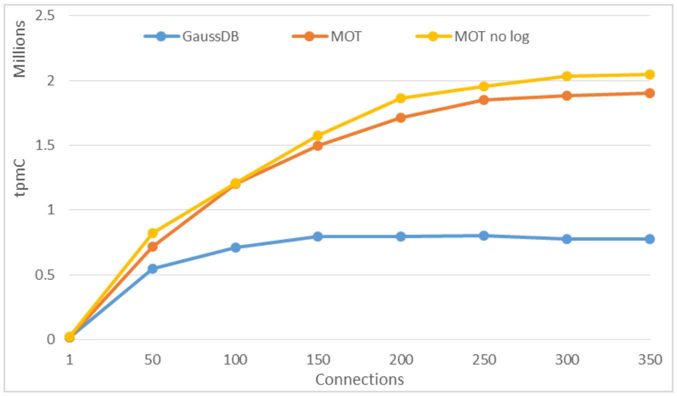
4.3 Results

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.

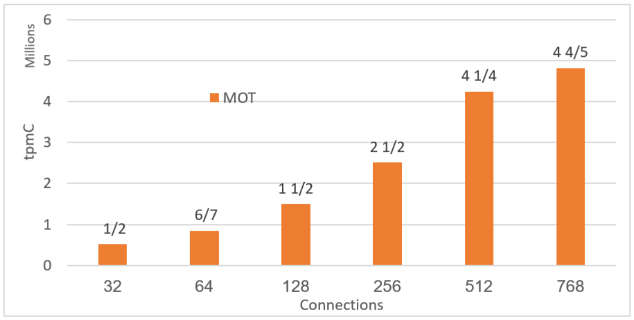


Figure 7: TPC-C with 1000 warehouses on ARM(256)

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

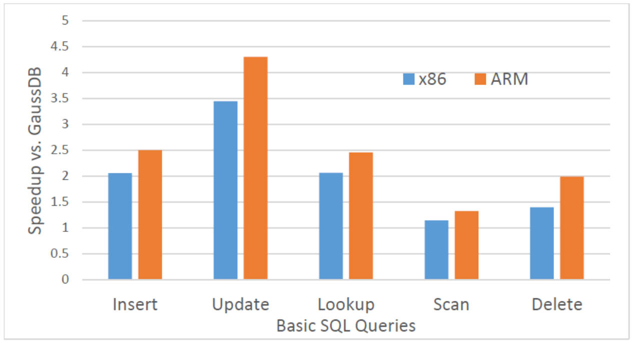


Figure 8: Speedup of Basic SQL queries

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.

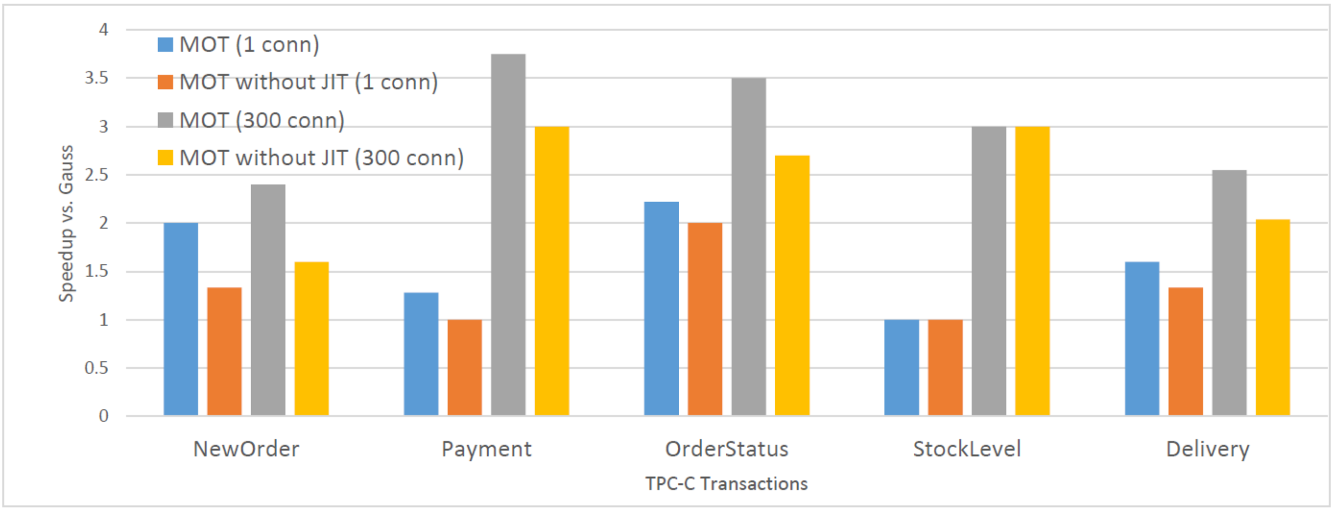


Figure 9: Speedup of TPC-C transactions (x86)

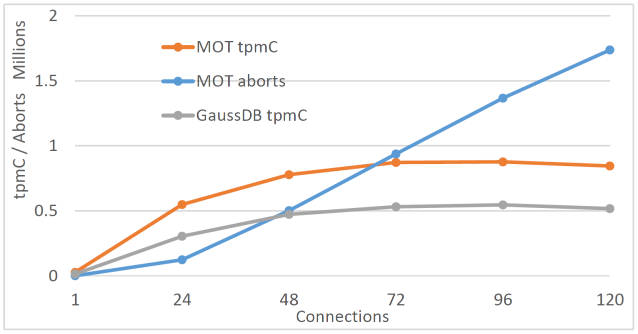


Figure 10: Full TPC-C performance on x86 with high contention

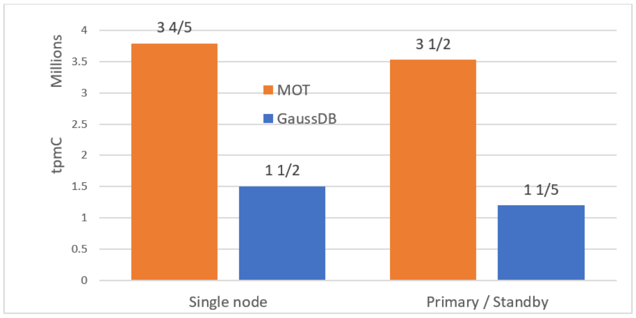


Figure 11: Replication overhead for TPC-C with 1000 warehouses and 812 clients on ARM(128)

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.