Improving performance by avoiding transaction logging on Load Isolated tables in Teradata

Chandrasekhar Tekur
Teradata India Pvt Ltd
Hyderabad, India
t.chandrasekhar@yahoo.com

Arnab Phani
Teradata India Pvt Ltd
Hyderabad, India
phaniarnab@gmail.com

R.K.N Sai Krishna
Teradata India Pvt Ltd
Hyderabad, India
rkn.sai@gmail.com

***Abstract*—In Teradata database, the transient journal (TJ) is a system-maintained log that provides a way to protect transactions from various system failures. Each transaction processed by the database records a before image of rows that are touched by the transaction. Then if a transaction fails for some reason, the before image of any modified rows can be retrieved from the transient journal and written over the modifications, returning the row to the same state it was in, before the transaction started. Teradata Database removes all before images from the transient journal after a transaction commits [1]. In addition, Teradata provides Load Isolation feature, a table isolation property (MVCC variant), that enables concurrent read of committed rows while the tables are being modified. To enable committed reads in concurrent sessions, rows that are being modified are logically deleted from load-isolated tables and new rows with the modified values are inserted when rows are updated. If a row is not yet committed, then the data that is read is the data that was true and final in the table prior to the subsequent data change [2].**

TJ maintenance is not an exception even for load-isolated tables in Teradata. TJ, though unavoidable, incurs extra costs to the system in the form of IO and CPU. Additionally the log needs to be purged periodically to cleanup records no longer needed causing additional overhead. This paper discusses an approach to avoid logging, with no compromise in data consistency, using Teradata Load Isolation, resulting in (a) significant improvement in the performance and transaction throughput, and (b) constant time (O(1)) transaction rollback, unlike the traditional approach(O(n)), where n is the number of rows.

Keywords—Transient journal, MVCC, database, rollback, snapshot isolation

I. INTRODUCTION

Transient journaling in Teradata is an ARIES based WAL logging[4]. ARIES based WAL logging is widely adopted in traditional database systems. This logging strategy originally demands constructing log records for each modified row. To reduce the overhead of WAL logging, Salem et al [5] proposed recording only incoming statements.

ARIES based log contains the before and after images of rows. Dewitt et al [6] proposed to reduce the log size by logging only the new values. However, log records without before images cannot enable the undo operation. So it needs large enough stable memory which can hold the complete log records for active transactions. They also proposed to write log records in batches to optimize the disk I/O performance. Similar techniques such as group commit [7] have also been explored in modern database systems.

An asynchronous commit strategy [8] is also proposed, which allows transactions to complete without waiting for the log writing to finish. This proposal can reduce the overhead of log flush to some extent. However, it sacrifices the durability guarantee, since the states of the committed transactions may lose at the time of failure. Lomet et al [9] proposed a logical logging system. The recovery phase of ARIES logging combines physiological redo and logical undo. This extends ARIES to work in a logical setting, and has been used to make ARIES logging more suitable for in-memory database system. Systems such as [10, 11] adopt this logical strategy.

Efficient checkpointing is another research effort, primarily devoted to in-memory databases. Salem et al [12] surveyed many checkpointing techniques, which cover both inconsistent and consistent checkpointing with different logging strategies.

Johnson et al [13] worked on logging-related impediments to database system scalability. The overhead of log related locking contention decreases the performance of the database systems, since each transaction needs to hold the locks while waiting for its log to be materialized. Works such as [13, 14, 15] attempted to make logging more efficient by reducing the effects of locking contention.

In this paper, a novel approach is proposed to completely avoid transaction logging on Load Isolated (LDI) tables [2] in Teradata database, by taking advantage of multiple versions maintained in the system as part of LDI.

The paper is organized as follows: Section 2 describes the background concepts required for the proposed approach. It also introduces the needed notation and terminology. Section 3 summarizes the proposed architecture. Section 4 illustrates the approach using an example. Section 5 demonstrates performance results and section 6 summarizes the contributions of this paper.

II. BACKGROUND

Table 1: Notation

Notation	Description
LDI	*Load Isolation*
MVCC	*Multi Version Concurrency Control*
TJ	*Transient Journal*
ILID	*Insert Load Id*
DLID	*Delete Load Id*
RLID	*Row Load Id*
CI	*Commit Indicator*
LC	*Load Committed read*
LU	*Load Uncommitted read*
CLID	*Current Load Id*

In Teradata, snapshot isolation is implemented using MVCC concept at a table level. Tables eligible for snapshot

978-1-5386-8158-9/19/$31.00©2019IEEE

isolation are called Load Isolated (LDI) tables. In a typical Teradata table(non LDI), say t_1 comprised of columns c_1, c_2, … c_n, each row consists of a row identifier (rowid) followed by column values(v_1, v_2, …v_n) as shown below:

Rowid	v_1, v_2, …v_n

In an LDI table, each row maintains a new 8 byte Row Load ID (RLID) value apart from the rowid. The RLID comprises of two parts: Insert-Load ID (ILID) and Delete-Load ID (DLID), 4 bytes each. For readability, RLID is denoted as a pair (ILID, DLID).

Rowid	RLID(ILID, DLID)	v_1, v_2, …v_n

A new 32-bit value called as Current Load ID (CLID) is maintained in the table header to record the last committed load id value for the table. CLID is initially 0 and is incremented by 1 for each load commit. When a row is inserted, the RLID value is populated as (CLID+1, 0), i.e., ILID is CLID+1 and DLID is 0.

For example, consider a row with rowid r1, CLID 0, columns values v_1, v_2, … v_n following would be the row structure after an insert operation:

r_1	(1,0)	v_1, v_2, …v_n

Once the insert operation is committed the CLID is incremented by 1.

In an LDI table, rows are only logically deleted by just marking the DLID part of the RLID as CLID+1. For example, considering the inserted row in the above example, following would be the row structure after a delete operation:

r_1	(1,2)	v_1, v_2, …v_n

Updating a row involves two steps,

a) Logical deletion of existing row
b) Insertion of a new row with updated values.

An update operation on a single row results in two rows, one corresponding to logical deletion and the other for insertion.

The logically deleted rows are termed as garbage rows, and these rows are physically removed from the table periodically as per user's choice. If DLID <> 0, then the corresponding row is eligible for garbage collection (physical removal). For example, consider the following row in an LDI table:

r_1	(1,0)	v_1, v_2, …v_n

After updating column c_1 (value v_1 changed to v_1^1), following are the resulting rows in the table:

RowID	RLID	Column values
r_1	(1,2)	v_1, v_2, …v_n
r_1^1	(2,0)	v_1^1, v_2, …v_n

Here, the row with rowid r_1 is logically deleted and the row with rowid r_1^1 is newly inserted with updated value. As part of garbage collection, the row with DLID value of 2 (r_1) would be physically removed.

For a normal read operation, the logically deleted rows are filtered out using a Load Commit (LC) condition on the RLID and CLID.

```
// LC condition
if ((ILID <= CLID) &&
   ((DLID == 0 ) || (DLID > CLID)))
    // valid row
else
    // garbage row
```

- ILID <= CLID condition ensures that the row under consideration is a committed row
- DLID = 0 ensures that the row is not logically deleted, and thus the row is qualified for read operation.
- DLID > CLID condition ensures that rows that are logically deleted as part of a parallel load operation (uncommitted) are qualified for read operation.

While a load operation is in progress on an LDI table, a concurrent reader cannot see the uncommitted modifications. In order to read uncommitted rows, a condition of DLID = 0 is applied on the rows.

```
// LU condition
if (DLID == 0)
    // valid row
else
    // garbage row
```

When the 32-bit CLID overflows, a garbage cleanup operation needs to be performed on the table before any subsequent load operations. This would reset the CLID value to 0 and the RLID value pair of each row to (0,0).

As part of journaling, before Teradata changes a data row (update, insert or delete statement), it takes a backup. The purpose of the transient journal is to allow rollback of failed transactions. In the case of a rollback, the base table row can be quickly replaced with the backup row from the transient journal.

The transient journal cannot be turned off manually to ensure database integrity. Often it is desirable to avoid its usage for performance reasons. Especially when doing a lot of changes on a table (such as updating billions of rows), the transient journal can become huge resulting in database running out of space. This would have significant implications on the existing workload (failing sessions, etc.). Another nasty side effect of the transient journal shows up when updating many rows in an enormous and skewed table. The rollback usually consumes a lot of resources and might have an adverse impact on the overall system performance [3]. Several customer experiences indicate that rollback operation can be very expensive, sometimes taking hours to days depending on the transaction size. There are even tools

(for example Recovery Manager in Teradata) to manage rollback operations.

The proposed approach aims to avoid transaction journaling altogether for the data modifications on tables defined as LDI, thereby avoiding the overhead needed for the maintenance of TJ.

III. Proposed Approach

In the proposed approach, a new commit indicator (CI), a 64 bit array to capture commit status of all the transactions on an LDI table (one bit for each transaction), is stored in the table header, in addition to the CLID. For each transaction commit/rollback on the table, corresponding bit in CI is set to either 0 or 1 respectively. The 64 bit array is initialized to 0 as part of table creation. If a transaction on the table is rolled back, the corresponding bit is set to 1. A subsequent transaction on that table uses the next bit in the array. Consequently, upto 64 load operations can be performed, after which garbage cleanup must be initiated on the table. The CI bit array can be extended in future if needed.

Currently the CLID is incremented by 1 after a load is committed. This approach proposes to increment the CLID by 2 at the end of each transaction irrespective of whether transaction is committed or rolled back.

During transaction rollback, no rows from the table are touched, the uncommitted rows (rows modified as part of an aborted transaction) remain in the table and are considered as garbage rows. Only the table header is modified by setting the corresponding bit in CI to 1. This radically reduces the rollback time from an order of transaction size to a constant time.

The garbage rows are not visible to the readers. An extra row filter as shown below (in addition to the existing condition) is applied on each row during read to filter out garbage rows:

```
if (DLID == 0)
{
    if ((ILID == 0) ||
        (CI & (1 << ((ILID-1)/2)) == 0))
        // Row is valid
    else
        // Row is filtered out
        // Garbage row inserted by an aborted transaction
}
else
{
    if (CI & (1 << ((DLID-1)/2)) == 1)
        // Row is valid
    else
        // Row is filtered out
        // Garbage row logically deleted by an aborted transaction
}
```

- If DLID=0 and ILID=0 in a row, it indicates the row was in the table before a recorded load operation and is valid.
- If DLID=0, ILID<>0 and (CI & (1 <<((ILID-1)/2)) =0), the row is valid, and it was not inserted as part of an aborted transaction. The value (ILID-1/2) indicates the bit position in CI corresponding to the transaction in which this row was inserted. If the bit value in CI is 0, the row is valid, else it was inserted as part of an aborted transaction and should be filtered out.
- If DLID<>0, the row is valid even if it was logically deleted by an aborted transaction. The corresponding bit value of 1 in CI at a position indicated by (DLID-1/2) implies the row is valid.

Garbage cleanup uses the above filter in the reverse method to identify the garbage rows to physically remove from the table.

Following is the new LC condition on an LDI table.

```
// New LC condition
if (( (DLID == 0 ) &&
     (ILID != 0 && ((CI & (1 << ((ILID-1)/2)) == 0)))) ||
    ((DLID > CLID) ||
    (DLID != 0 && ((CI & (1 << ((DLID-1)/2))) == 0))))
     // valid row
else
     // garbage row
```

Following is the new LU condition on an LDI table.

```
// New LU condition
if (((DLID == 0) && (ILID == 0 ||
      ((CI & (1 << ((ILID-1)/2)) == 0)))) ||
      ((DLID != 0) && ((CI & (1 << ((DLID-1)/2))) == 0)))
      // valid row
else
      // garbage row
```

With the proposed approach, there is an additional overhead of garbage rows taking extra space in the table. In addition, for read operations, an additional check needs to be performed to filter out the gargabe rows. To minimize the overhead incurred, garbage cleanup can be scheduled at regular intervals.

IV. Illustration

Following is the initial state of an LDI table in Teradata database.

Table header: (CLID = 0, CI=0x000…0)
Rows: none

Transaction 1 (insert two rows)
After transaction commit:
Table Header(CLID = 2, CI=0x000…0)

RowID	RLID	Column values
r_1	(1,0)	$v_1, v_2, \ldots v_n$
r_2	(1,0)	$v_1, v_2, \ldots v_n$

Transaction 2(insert row r_3, update r_1)
After transaction commit:
Table Header(CLID = 4, CI=0x000…00)

RowID	RLID	Column values
r_1	(1,3)	$v_1, v_2, \ldots v_n$
r_1^1	(3,0)	$v_1^1, v_2 \ldots v_n$
r_2	(1,0)	$v_1, v_2, \ldots v_n$
r_3	(3,0)	$v_1, v_2, \ldots v_n$

Transaction 3(insert row r_4, update r_2)
After transaction Rollback:
Table Header(CLID = 6, CI=0x000…100)

RowID	RLID	Column values
r_1	(1,3)	$v_1, v_2, \ldots v_n$
r_1^1	(3,0)	$v_1^1, v_2 \ldots v_n$
r_2	(1,5)	$v_1, v_2, \ldots v_n$
r_2^1	(5,0)	$v_1^1, v_2 \ldots v_n$
r_3	(3,0)	$v_1, v_2, \ldots v_n$
r_4	(5,0)	$v_1, v_2, \ldots v_n$

Note that r_2^1 and r_4 are garbage rows since the above transaction is rolled back.

Transaction 4(insert row r_5, update r_3)
After transaction commit:
Table Header(CLID = 8, CI=0x000…0100)

RowID	RLID	Column values
r_1	(1,3)	$v_1, v_2, \ldots v_n$
r_1^1	(3,0)	$v_1^1, v_2 \ldots v_n$
r_2	(1,5)	$v_1, v_2, \ldots v_n$
r_2^1	(5,0)	$v_1^1, v_2 \ldots v_n$
r_3	(3,7)	$v_1, v_2, \ldots v_n$
r_3^1	(7,0)	$v_1^1, v_2 \ldots v_n$
r_4	(5,0)	$v_1, v_2, \ldots v_n$
r_5	(7,0)	$v_1, v_2, \ldots v_n$

Read from table returns following rows by applying the row filter:
r_1^1, r_2, r_3^1, r_5.

Rows r_2^1 & r_4 are filtered out even though their DLID is zero because these rows are inserted by an aborted transaction and are considered garbage. Similarly r_3 is qualified even though its DLID is non-zero because it was deleted by an aborted transaction.

Garbage cleanup will pick rows r_1, r_2^1, r_3, r_4 for physical removal.

Table after garbage cleanup with reset.
Table header (CLID = 0, CI=0x000…0)
Rows: {r_1^1(0,0), r_2(0,0), r_3^1(0,0), r_5(0,0)}

V. Performance Results

All experiments were run on a Teradata system with 24 parallel Intel x86 processors, and 128GB of memory. For experimental setup, 100 different tables with various kinds of schema and row sizes are considered. Inserts, deletes and updates are performed on an LDI table with a large number of rows (>100 million) and measured the CPU, IO and Elapsed time differences with and without TJ and observed significant performance improvements with the proposed approach.

Figure 1 and Figure 2 demonstrate ~40% and ~50% improvement in CPU time and IO reduction respectively by avoiding TJ writes as part of transactions performing various kinds of write operations.

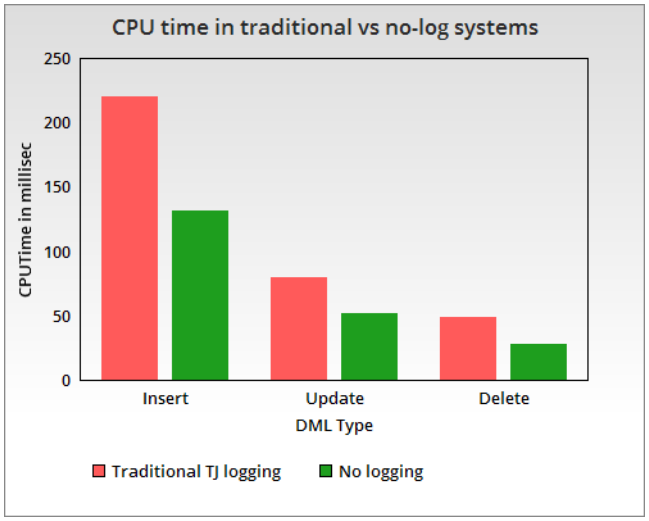

Figure 1: Performance comparison in terms of CPU

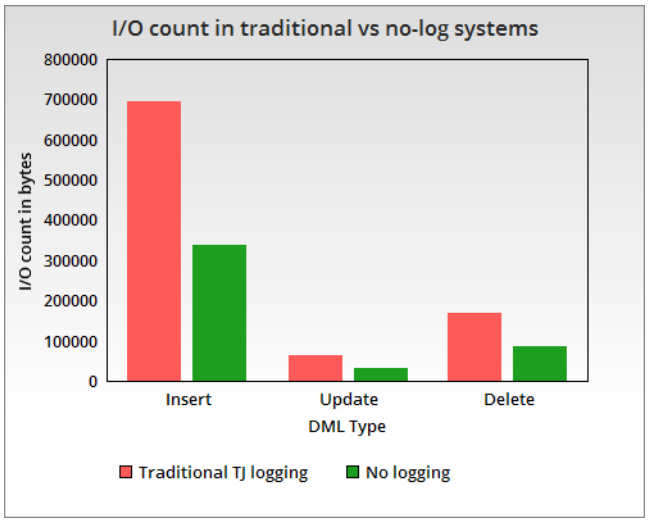

Figure 2: Performance comparison in terms of IO

For rollback of a load operation (involving more than a million rows), it took ~3 hours with the existing approach. With the proposed approach, the rollback operation is completed in less than a second.

VI. Summary

In this paper, a novel approach is proposed that aims to avoid transaction journaling altogether for the data modifications on tables defined as LDI, thereby avoiding the overhead needed for the maintenance of TJ. The approach avoids maintainance of a separate transaction journal while modifying the database records. During transaction rollback, no recovery processing is required for the modified records. In addition, the size of transaction journal is greatly reduced which makes the periodic TJ purge faster.

Acknowledgment

The authors would like to thank Manjula Koppuravuri, Engineering Director, Teradata India Pvt Ltd, for providing architectural guidance.

VII. REFERENCES

[1] https://docs.teradata.com/reader/Fs1l1bqzqbnO0oVqjSVP5g/tXf6b4JILf3wQHr9vkznmw.

[2] https://docs.teradata.com/reader/Ws7YT1jvRK2vEr1LpVURug/OIMLq7F0gPiqeJOQT2DyZQ

[3] https://www.dwhpro.com/teradata-transient-journal/

[4] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17, 1 (March 1992), 94-162. DOI=http://dx.doi.org/10.1145/128765.128770

[5] Kenneth Salem and Hector Garcia-Molina. 1989. Checkpointing Memory-Resident Databases. In Proceedings of the Fifth International Conference on Data Engineering. IEEE Computer Society, Washington, DC, USA, 452-462.

[6] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In Proceedings of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD '84). ACM, New York, NY, USA, 1-8. DOI=http://dx.doi.org/10.1145/602259.602261.

[7] R. Hagmann. 1987. Reimplementing the Cedar file system using logging and group commit. In Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87). ACM, New York, NY, USA, 155-162. DOI: https://doi.org/10.1145/41457.37518.

[8] Postgresql 8.3.23 documentation,chapter 28. reliability and the write-ahead log. https://www.postgresql.org/docs/8.3/static/wal-async-commit.html. Accessed: 2015-6-06.

[9] David Lomet, Kostas Tzoumas, and Michael Zwilling. 2011. Implementing performance competitive logical recovery. Proc. VLDB Endow. 4, 7 (April 2011), 430-439. DOI=http://dx.doi.org/10.14778/1988776.1988779.

[10] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. Proc. VLDB Endow. 1, 2 (August 2008), 1496-1499. DOI=http://dx.doi.org/10.14778/1454159.1454211.

[11] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," 2011 IEEE 27th International Conference on Data Engineering, Hannover, 2011, pp. 195-206. doi: 10.1109/ICDE.2011.5767867.

[12] K. Salem and H. Garcia-Molina, "System M: a transaction processing testbed for memory resident data," in IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, pp. 161-172, March 1990. doi: 10.1109/69.50911.

[13] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. Proc. VLDB Endow. 3, 1-2 (September 2010), 681-692. DOI=http://dx.doi.org/10.14778/1920841.1920928.

[14] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented transaction execution. Proc. VLDB Endow. 3, 1-2 (September 2010), 928-939. DOI=10.14778/1920841.1920959 http://dx.doi.org/10.14778/1920841.1920959.

[15] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: page latch-free shared-everything OLTP. Proc. VLDB Endow. 4, 10 (July 2011), 610-621. DOI=http://dx.doi.org/10.14778/2021017.2021019.