Commit Time Materialized View Maintenance for Bulk Load Operations in Teradata

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

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

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

***Abstract*—Materialized view is called Join Index (JI) in Teradata. JIs can store tables pre-joined, solely change a table to a different schema or aggregation results. JIs are stored permanently on the disks and cannot be peeped through. Also, their usage or evasion isn't subject to user choice. Teradata optimizer decides when to use a JI, or the underlying table(s) directly from a cost point of view. While the advantage of having JIs is to increase read performance, its downside could be their maintenance. In case of row insertion or modification of an indexed column in base table, JI must be updated. This can create a big performance impact if multiple DMLs within same transaction result in same JI(s) maintenance, bulk load operations for example.**

On tables with Multiversion Concurrency Control (MVCC), readers and writers work concurrently (Load Isolation in Teradata for example). Readers continue to access the last committed rows while concurrent modifications are happening on the base table(s). JIs defined on multiversioned tables also follow the MVCC principle and continue allowing readers to access the last committed rows. Currently, JIs are maintained immediately with every DML request on base table(s). The proposed approach is to defer the JI maintenance till commit time for MVCC tables, resulting in performance improvement for write operations, as all the JI related operations are performed at the end.

***Keywords*—join index, load isolation, MVCC, optimizer, transaction**

I. INTRODUCTION

Materialized views and its maintenance are crucial for data warehouses, banking, billing and retailing applications. Creating materialized view can be a very good practice in many cases. It can store tables pre-joined, solely change a table to a different schema or aggregation results. If some specific joins are frequently required, it is always good and recommended to create join indexes for those joins [20]. JIs can give great cost benefits from optimization point of view.

Usually, a view is updated/maintained immediately, as part of the SQL request that updates/modifies the base table. The overhead of having JIs is that Teradata Database updates the JI in case of update or deletion of an indexed (referenced in the JI) column of the base table, or a new row insertion. Maintenance overhead of a table is directly proportional to the number of join indexes defined for it. This can create a big performance impact if multiple DMLs modify the JI within same transaction. Majority of the commercial database systems available today including IBM DB2 [4, 5], Oracle [3] and Microsoft SQL Server [8], PostgresSQL[23] came up with their own way of view maintenance which better suite their underlying architecture.

The proposed approach addresses the performance degradation incurred in maintaining JIs defined on one or more multi-versioned tables. Immediate maintenance imposes a significant overhead in terms of execution time, CPU usage and I/O that cannot be accepted in many applications. Deferred maintenance, in contrast results in JI being inconsistent with its definition. The proposed approach chooses to implement this only for multi-versioned tables where the committed data is still accessible for readers while the table is being modified [18, 19]. Various applications of MVCC are becoming increasingly important with the advent of emerging application architectures like Hybrid transactional/analytical processing (HTAP[24]). The objective of this approach is to defer JI maintenance till commit time. For now, the research is limited to tables following MVCC principles to cope up with the industry requirements.

A. Notation

AMP [21]	*Acronym for Access Module Processor, execution unit in Teradata database.*
JI [20]	*Acronym for Join Index, materialized view in Teradata database.*
LDI [18, 19]	*Acronym for Load Isolation, a variant of snapshot isolation in Teradata database.*
Execution Plan [22]	*Execution flow of a query.*

B. Load Isolation

Teradata provides a table isolation property that enables reading committed rows from tables, even though the data is changing while the table is being loaded. Load isolation ensures that the most recently committed data is retrieved. 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. To enable concurrent index-based reads, indexes (including join indexes) also maintain the commit property of the row. A

978-1-5386-8158-9/19/$31.00©2019IEEE

join index table is marked as a load-isolated table if any of the referenced base tables is a load-isolated table.

Example 1: LDI table

```
CREATE TABLE DB2.t1 ,FALLBACK ,
        NO BEFORE JOURNAL,
        NO AFTER JOURNAL,
        DEFAULT MERGEBLOCKRATIO,
        WITH CONCURRENT ISOLATED LOADING FOR
    ALL
        (
        a INT,
        b INT,
        c INT)
    PRIMARY INDEX ( a );

CREATE JOIN INDEX DB2.j1 ,FALLBACK ,
AS
    SELECT DB2.t1.b
     FROM DB2.t1
    WHERE DB2.t1.b >  0
    PRIMARY INDEX ( b );

INSERT INTO t1(101, 102, 103);
```

Session 1 (writer)	**Session 2 (Reader)**
*SELECT * FROM DB2.t1;* *a b c* *--- --- ---* *101 102 103* *SELECT * FROM j1;* *b* *---* *102* */* Selecting from a join index is not possible normally, above is done through a diagnostic */*	*SELECT * FROM DB2.t1;* *a b c* *--- --- ---* *101 102 103* *SELECT * FROM j1;* *b* *---* *102* */* Selecting from a join index is not possible normally, above is done through a diagnostic */*
BT; / Begin transaction */*	
INSERT WITH ISOLATED LOADING DB2.t1(110, 220, 330);	*LOCKING DB2.t1 FOR LOAD COMMITTED* *SELECT * FROM DB2.t1;* *a b c* *--- --- ---* *101 102 103* */* New rows are not visible as the writer transaction is yet to commit*/*
ET; / End transaction */*	
	LOCKING DB2.t1 FOR LOAD COMMITTED *SELECT * FROM DB2.t1;* *a b c* *--- --- ---* *101 102 103* *110 220 330*

Teradata's Load Isolation feature improves concurrency and system throughput by allowing "Readers to not block Writers and Writers to not block Readers," adding to the existing "Read Uncommitted" and "Serialize" isolation levels.

Example 1 shows how reader reads only the last committed rows while another transaction is writing on a table.

C. Join index on LDI tables

A join index table is like a regular LDI table if any of the base tables is LDI. Join indexes on LDI tables are maintained with every DML on base table like any other regular JIs. LDI JIs' maintenance is unnecessary for every DML as no reader (who wants to read committed rows) will be able to read the newly inserted/updated rows in the JI. Example 2 shows that the new rows are only visible to reader only after writer commits the transaction.

Example 2: LDI table with JI

Session 1 (writer)	**Session 2 (Reader)**
BT; / Begin transaction */*	
INSERT WITH ISOLATED LOADING DB2.t1(111,222,333);	*LOCKING DB2.j1 FOR LOAD COMMITED* *SELECT * FROM DB2.j1;* *b* *---* *102* *220* */* New rows are visible now as the writer transaction is committed*/*
ET; / End transaction */*	
	LOCKING DB2.j1 FOR LOAD COMMITED *SELECT * FROM DB2.j1;* *b* *---* *102* *220* *222*

II. PROPOSED APPROACH

This approach enables JIs on LDI table to be maintained at the end of a transaction. As a result, JI will not to be in sync with the base tables during the lifetime of the transaction. However, JI is still usable since it contains committed rows. Even though base tables have updated rows and JIs still have not got those updates, same set of rows are returned to the readers irrespective of whether JI qualifies or not.

A. Optimizer Enhancements

Teradata Optimizer would be the most affected module with this approach. Existing approach generates JI update steps with each DML on base tables. With the proposed approach, optimizer skips JI processing for DMLs, if the base table(s) is LDI. At the end transaction phase, optimizer would check all the tables updated in this transaction and generates update steps for the JIs. Optimizer also must skip taking JI path for read queries in load transaction as JIs will have dirty rows till commit time. This may look like a limitation but the case is unlikely, as DBAs don't usually mix up load and read queries in same transaction.

Following is the plan for INSERT DML into a base table with the existing approach. The bold part of the plan is for eager JI maintenance.

EXPLAIN INSERT with ISOLATED LOADING t1(4,5,6);

Explanation

---

1) First, we lock db.J1 for write on a reserved RowHash to prevent global deadlock.

2) Next, we lock DB.t1 for write on a reserved RowHash to prevent global deadlock.

3) We lock db.J1 for write, and we lock DB.t1 for write.

4) We Begin Isolated Load on db.T1, db.J1.

5) We execute the following steps in parallel.

- *1) **We do an INSERT into (concurrent load isolated) DB.t1**. The estimated time for this step is 0.02 seconds.*
- *2) **We do an INSERT into (concurrent load isolated) db.J1**. The estimated time for this step is 0.01 seconds.*
- *3) We lock DBC.TVM for write on a RowHash, and then we do a single-AMP UPDATE from DBC.TVM by way of the unique primary index "Field_1 = '00000204'XB, Field_2 = 'T1'" with no residual conditions.*
- *4) We lock DBC.TVM for write on a RowHash, and then we do a single-AMP UPDATE from DBC.TVM by way of the unique primary index "Field_1 = '00000204'XB, Field_2 = 'J1'" with no residual conditions.*

6) We End Isolated Load on db.T1, db.J1.

7) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

Following is the plan for INSERT DML into a base table with the proposed approach. The bold part of the plan is for deferred JI maintenance.

EXPLAIN INSERT with ISOLATED LOADING t1(4,5,6);

Explanation

---

1) First, we lock DB.t1 for write on a reserved RowHash to prevent global deadlock.

2) Next, we lock DB.t1 for write.

3) We Begin Isolated Load on db.T1.

4) We execute the following steps in parallel.

- *1) We do an INSERT into (concurrent load isolated) DB.t1. The estimated time for this step is 0.02 seconds.*

Note: No JI Insert

- *2) We lock DBC.TVM for write on a RowHash, and then we do a single-AMP UPDATE from DBC.TVM by way of the unique primary index "Field_1 = '00000204'XB, Field_2 = 'T1'" with no residual conditions.*
- *3) We lock DBC.TVM for write on a RowHash, and then we do a single-AMP UPDATE from DBC.TVM by way of the unique primary index "Field_1 = '00000204'XB, Field_2 = 'J1'" with no residual conditions.*

5) We do an all-AMPs RETRIEVE step from DB.T1 (Load Uncommitted) by way of an all-rows scan into Spool 1 (all_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 1 row (25 bytes). The estimated time for this step is 0.03 seconds.

6) We do an all-AMPs RETRIEVE step from Spool 1 by way of an all-rows scan into Spool 2 (all_amps), which is redistributed by the hash code of (DB.T1.b) to all AMPs. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with no confidence to be 1 row (17 bytes). The estimated time for this step is 0.03 seconds.

7) We do an all-AMPs MERGE into db.J1 (concurrent load isolated) from Spool 2. The size is estimated with no confidence to be 1 row. The estimated time for this step is 1 second.

8) We End Isolated Load on db.T1, db.J1.

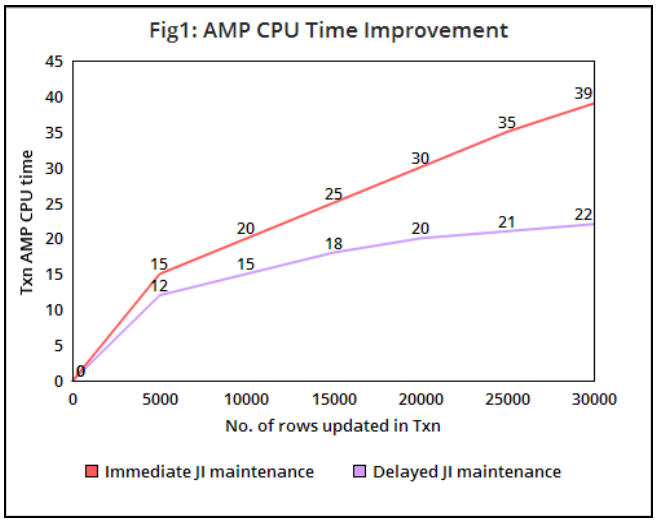
9) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

III. EXPERIMENTS

All experiments were run on a setup with eight nodes where each node is a Teradata machine with 24 parallel Intel x86 processors, and 128GB of memory. For experimental setup, we considered:

- Two base tables with JIs defined on them, which had 20 million rows initially.
- Six transactions with multiple load DMLs, each loading multiple rows.

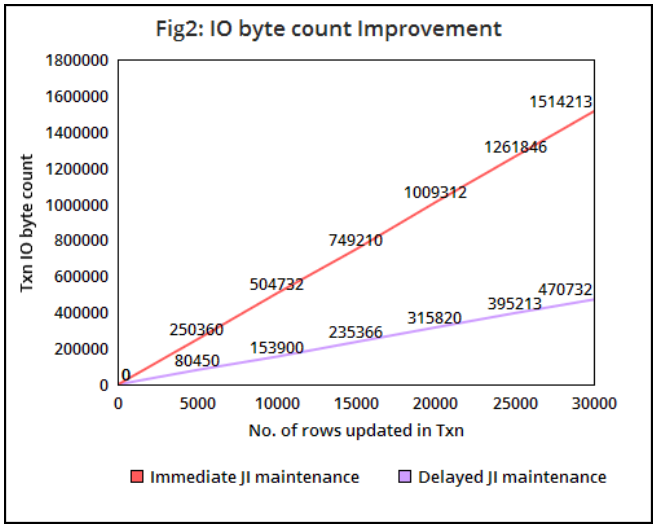
A. *Experimental Results*



25 to 80 percent of performance improvement was observed in terms of execution time and more than 100 percent in terms of IO byte count. Figure 1 summarizes the performance improvement observed in CPU time spent in AMP in millisecond. Similar improvement is observed in elapsed time too. Figure 2 compares IO byte count between eager and deferred JI maintenance.

Performance improvement with this approach depends on the following factors:

- Number of rows present in base tables and JIs: Inserting rows into an empty table/JI is less costly than inserting into a loaded table/JI. So, transactions processing big tables tend to give better performance results.
- Number of JIs defined on the base table(s): More the number of JIs defined, higher the IO and CPU performance benefit.
- Complicacy level of the JI definitions: Inclusion of aggregates, multiple WHERE clauses on different columns, etc. in JI definition can increase the JI maintenance cost with higher margin.



IV. RELATED WORK

Work related to materialized view maintenance grabbed considerable recognition in the research community over the last two decades. Majority of the commercial database systems available today including IBM DB2 [4, 5], Oracle [3] and Microsoft SQL Server [8], PostgresSQL[23] invented their own way of view maintenance which better suite their underlying architecture.

Many incremental view maintenance algorithms are studied in the context of eager maintenance [7, 8, 9, 10, 5, 11, 12, 23]. Nguyen Tran Quoc Vinh [23] proposed an incremental update algorithm. It is an improvisation of the one published prior to this. In addition, in PostgreSQL DBMS, for synchronous incremental updates of a materialized view, he proposed an algorithm that generates source code of triggers automatically in PL/pgSQL language. Eager compensation as a technique for view maintenance is put forward in [15, 16, 17] for the case when base tables are spread across multiple systems. A multi-versioning based concurrency control is studied in [13] to reduce contention of resources.

Maintenance of deferred view is proposed in [1] and [14], though the goals are distinct. Algorithmic issues are the main focus in both the papers and the experimental results provided are very minimal. Salem et al. [14] proposed a view maintenance algorithm which is incremental and works as a series of small steps. Dividing view maintenance into small parts can be quite inefficient – combining those can bring in improved efficiency. J. Zhou et al. [2] proposed an algorithm of lazy view maintenance with overhead of maintenance manager and execution time view update, while our approach adds no overhead in query execution.

V. CONCLUSIONS AND FUTURE WORK

Materialized views are proven to accelerate query execution time but must be kept up-to-date with base tables. Usually views are needed to be maintained as part of transaction to remove the possibilities of dirty JI read. This enables readers always to get updated JIs, but updaters pay for view maintenance. Eager maintenance slows down DMLs, especially when multiple views are affected. Moreover, if the transaction follows snapshot isolation principle, the effort to update views with every DML goes complete waste as readers don't need the updated views till transaction end.

This approach brought a considerable amount of performance improvement by deferring view maintenance till transaction end even though it came with some side effects like longer End Transaction request, unusable JI path in load transaction.

For now, this approach is limited to tables following MVCC principles as the customers who are using LDI tend to require this approach. As a future work same approach can be researched for applying over any table.

ACKNOWLEDGMENT

The authors thank Manjula Koppuravuri, Engineering Director, Teradata India Pvt Ltd for the guidance and support.

VI. REFERENCES

[1] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. 1996. Algorithms for deferred view maintenance. In Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD '96), Jennifer Widom (Ed.). ACM, New York, NY, USA, 469-480. DOI=http://dx.doi.org/10.1145/233269.233364.

[2] Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. 2007. Lazy maintenance of materialized views. In Proceedings of the 33rd international conference on Very large data bases (VLDB '07). VLDB Endowment 231-242.

[3] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan, Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. 1998. Materialized Views in Oracle. In Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98), Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 659-664.

[4] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. 2000. Answering complex SQL queries using automatic summary tables. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD '00). ACM, New York, NY, USA, 105-116. DOI=http://dx.doi.org/10.1145/342009.335390.

[5] Wolfgang Lehner, Richard Sidle, Hamid Pirahesh, and Roberta Wolfgang Cochrane. 2000. Maintenance of cube automatic summary tables. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD '00). ACM, New York, NY, USA, 512-513. DOI=http://dx.doi.org/10.1145/342009.335454.

[6] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. In Proceedings of the 2001 ACM SIGMOD international conference on Management of data (SIGMOD '01), Timos Sellis and Sharad Mehrotra (Eds.). ACM, New York, NY, USA, 331-342. DOI=http://dx.doi.org/10.1145/375663.375706.

[7] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. In Proceedings of the 1986 ACM SIGMOD international conference on Management of data (SIGMOD '86), Carlo Zaniolo (Ed.). ACM, New York, NY, USA, 61-71. DOI=http://dx.doi.org/10.1145/16894.16861.

[8] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining views incrementally. In Proceedings of the 1993 ACM SIGMOD international conference on Management of data (SIGMOD '93), Peter Buneman and Sushil Jajodia (Eds.). ACM, New York, NY, USA, 157-166. DOI=http://dx.doi.org/10.1145/170035.170066.

[9] Timothy Griffin and Leonid Libkin. 1995. Incremental maintenance of views with duplicates. In Proceedings of the 1995 ACM SIGMOD international conference on Management of data (SIGMOD '95), Michael Carey and Donovan Schneider (Eds.). ACM, New York, NY, USA, 328-339. DOI=http://dx.doi.org/10.1145/223784.223849.

[10] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. 1997. Maintenance of data cubes and summary tables in a warehouse. SIGMOD Rec. 26, 2 (June 1997), 100-111. DOI: https://doi.org/10.1145/253262.253277.

[11] S. Chen, E.A. Rundensteiner, "GPIVOT: Efficient Incremental Maintenance of Complex ROLAP Views", 21st International Conference on Data Engineering (ICDE 05), pp. 552-563, 2005.

[12] P.-Å. Larson, J. Zhou, "Efficient maintenance of materialized outer-join views", ICDE, pp. 56-65, 2007.

[13] Dallan Quass and Jennifer Widom. 1997. On-line warehouse view maintenance. SIGMOD Rec. 26, 2 (June 1997), 393-404. DOI: https://doi.org/10.1145/253262.253352.

[14] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. 2000. How to roll a join: asynchronous incremental view maintenance. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD '00). ACM, New York, NY, USA, 129-140. DOI=http://dx.doi.org/10.1145/342009.335393.

[15] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. 1995. View maintenance in a warehousing environment. In Proceedings of the 1995 ACM SIGMOD international conference on Management of data (SIGMOD '95), Michael Carey and Donovan Schneider (Eds.). ACM, New York, NY, USA, 316-327. DOI=http://dx.doi.org/10.1145/223784.223848.

[16] Y. Zhuge, H. Garcia-Molina, J. Wiener, "The Strobe Algorithms for Multi-Source Warehouse Consistency", Proceedings of International Conference on Parallel and Distributed Information Systems, 1996-December.

[17] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. 1997. Efficient view maintenance at data warehouses. In Proceedings of the 1997 ACM SIGMOD international conference on Management of data (SIGMOD '97), Joan M. Peckman, Sudha Ram, and Michael Franklin (Eds.). ACM, New York, NY, USA, 417-427. DOI: https://doi.org/10.1145/253260.253355.

[18] https://www.info.teradata.com/HTMLPubs/DB_TTU_16_00/index.html#page/SQL_Reference/B035-1142-160K/mgn1472241599813.html

[19] https://www.competence-site.de/techbytes-teradata-database-15-10-load-isolation/

[20] https://info.teradata.com/htmlpubs/DB_TTU_16_00/index.html#page/SQL_Reference/B035-1141-160K/oax1472241394652.h

[21] https://www.info.teradata.com/HTMLPubs/DB_TTU_16_00/index.html#page/SQL_Reference/B035-1141-160K/lkb1472241404075.html

[22] http://www.teradatapoint.com/teradata/explain-plan-interadata.htm

[23] Quoc Vinh, N.T., "Synchronous incremental update of materialized views for PostgreSQL", Program Comput Soft (2016) 42: 307. https://doi.org/10.1134/S0361768816050066

[24] https://en.wikipedia.org/wiki/Hybrid_transactional/analytical_processing_(HTAP)