2017 IEEE 33rd International Conference on Data Engineering

Optimizing UNION ALL Join Queries in Teradata

Mohammed Al-Kateb, Paul Sinclair, Alain Crolotte, Lu Ma, Grace Au, and Sanjay Nair
Teradata Labs
El Segundo, CA, USA
Email: firstname.lastname@teradata.com

***Abstract*—The UNION ALL set operator is useful for combining data from multiple sources. With the emergence of big data ecosystems in which data is typically stored on multiple systems, UNION ALL has become even more important. In this paper, we present optimization techniques implemented in Teradata Database for join queries with UNION ALL. Instead of spooling all branches of UNION ALL before performing join operations, we propose cost-based pushing of joins into branches. Join pushing not only addresses the prohibitive cost of spooling all branches, but also helps in exposing more efficient join methods (e.g., direct hash-based joins) which, otherwise, would not be considered by the query optimizer. The geography of relations being pushed to UNION ALL branches is also adjusted to avoid unnecessary redistributions and duplications of data. We conclude the paper with a performance study that demonstrates the impact of the proposed optimization techniques on query performance.**

I. Introduction

UNION ALL set operations are a means of combining data from multiples sources. In traditional relational databases, a UNION ALL query resembles a logical table that combines rows from multiple physical tables. One example is when a fact table in an existing data warehouse grows too big and a new fact table is defined as an extension of it. When a query is issued against the fact data, it needs to access both tables as one single relation. This can be accomplished by combining the two tables as a view or derived table using UNION ALL.

In the context of big data ecosystems, data may need to be combined from multiple, and possibly, heterogeneous data sources [1]. An example is with the Teradata **U**nified **D**ata **A**rchitecture (UDA) [2], in which current data of an application may be stored in a table on Teradata Database and history data is stored in HDFS [3] on a remote Hadoop server. Data from local and remote systems can be combined using UNION ALL for data analytics through Teradata's QueryGrid - a processing engine that allows users to use SQL to query local and remote systems seamlessly, transparently, and efficiently [4].

When a UNION ALL view or derived table joins to another relation, a naive plan is to write all UNION ALL branches to a common spool[1] and then join the spool to the other relation. However, the cost of spooling all branches can be prohibitively expensive. This can be the case, for instance, if the branches are very large fact tables with millions/billions of rows. Spooling all branches may also lead to out-of-spool scenarios. Joining the common spool to other relations (e.g., dimension tables) can be excessively costly as well. In **M**assively **P**arallel **P**rocessing (MPP) systems like Teradata Database, the spool may need to be redistributed for subsequent joins, which can add significant overhead to query execution.

[1]Spools are intermediate/buffer tables.

In this paper, we address the problem of optimizing join queries over UNION ALL views and derived tables. First, we propose a cost-based optimization for pushing joins to UNION ALL branches. Second, we introduce a complementary optimization for adjusting the geography of the relation pushed into a UNION ALL view or derived table. Finally, we present a performance study on the proposed optimizations.

The key idea of join pushing is to avoid spooling all UNION ALL branches prior to joins - if and when applicable. The proposed optimization is cost based. The base plan of spooling all branches is costed and compared to the cost of pushing joins. The better plan is picked. Join pushing is seamlessly integrated in the lookahead mechanism of the Teradata join planner [5]. As a result, the query optimizer can choose to push a single table or a join relation of multiple tables. Some UNION ALL branches are handled individually, while others are grouped together prior to join pushing. Branch grouping can help reduce the total number of join steps. In addition, join pushing exposes more join methods. For example, pushing a table to join with another table in a UNION ALL branch on the **P**rimary **I**ndex (PI) of both tables can yield a direct merge join where neither tables need to be redistributed or duplicated. Join pushing also allows iterative decomposition of a join into multiple joins. A UNION ALL join is planned as multiple joins. Each join can itself be planned as multiple joins, for instance, if it qualifies for **P**artial **R**edistribution **P**artial **D**uplication (PRPD) joins [6]. PRPD joins can also be further planned as multiple joins, for instance, if the join is with a **C**olumn-**P**artitioned (CP) table [7].

Join pushing adjusts the geography of the relation being pushed to UNION ALL branches. Geography adjustments help reduce the total cost of a query plan by avoiding unnecessary redistributions and duplications. Adjustments take place while evaluating different join pushing plans and the change in cost for all adjustments is factored into the final cost of each plan. Consequently, the query optimizer has accurate estimates upon which it can decide on the most optimal plan to choose.

We present a performance study conducted on a variation of the TPC-DS benchmark [8]. The purpose of the experiments is to examine the impact of join pushing on queries with UNION ALL. Performance metrics include I/O count, CPU time, and elapsed time. Measurements are reported for queries using product join, merge join, and hash join methods. The results of the experiments show that join pushing improves the performance of queries considerably.

The rest of this paper is as follows. Section II is an overview of Teradata Database. Section III describes the new optimizations. Section IV presents performance study. Section V outlines related work. Section VI concludes the paper.

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.165

II. Teradata Database

Teradata Database is a parallel database [9] with an advanced shared-nothing architecture that inherently enables horizontal scalability and fault-tolerance. The software building blocks of parallelism in Teradata Database are **P**arsing **E**ngine (PE) and **A**ccess **M**odule **P**rocessor (AMP).

A PE is the unit of parallelism at the session level. It handles multiple sessions concurrently. It is responsible for optimizing queries, generating execution plans, dispatching steps to AMPs, and receiving response back from AMPs and returning it the client system.

An AMP is the unit of parallelism for data processing. It is not associated to a specific session but is rather associated with its portion of the data. An AMP is a collection of worker tasks, which are threads that process database requests for the AMP's data. A worker task performs the actual work requested by a particular step such as sorting, aggregation, and joins.

A table in Teradata Database can have a **P**rimary **I**ndex (PI), **P**rimary **A**MP Index (PA), or **no P**rimary **I**ndex (NoPI). Rows of PI and PA tables are hash-distributed to AMPs. Rows of NoPI tables are randomly distributed to AMPs. Once rows land on an AMP, they are always kept on the AMP in the order of their *rowid* by the Teradata file system. A *rowid* is a 16-byte key value that uniquely identifies a row in a table. It is used by the Teradata file system to position to a specific row on the AMP using an optimized 2-level B* tree based on the fixed-length *rowid*.

III. Optimization Techniques

In this section, we present cost-based techniques for optimizing join queries on UNION ALL views and derived tables. The main technique is pushing joins to UNION ALL branches. The second optimization is adjusting the geography of the relation pushed to branches of UNION ALL.

A. Join Pushing

We explain the join pushing optimization technique using a running example. Consider a query that joins two tables; $t_1(a_1, b_1, c_1, d_1)$ and $t_2(a_2, b_2, c_2, d_2)$ with a UNION ALL derived table that has two branches; one branch retrieving from $t_3(a_3, b_3, c_3, d_3)$ and the other branch retrieving from $t_4(a_4, b_4, c_4, d_4)$ as follows:

SELECT a_1, a_2, a
FROM t_1,t_2, (SELECT a_3 FROM t_3 WHERE b_3=3
UNION ALL
SELECT a_4 FROM t_4 WHERE b_4=4) $dt(a)$
WHERE c_1=c_2 and d_1=a;

The base plan for this query is to execute the branches of the *d*erived *t*able *dt*, spool results in a common spool, and join the common spool to t_1 and t_2 using the best join method/order chosen by the query optimizer. This base plan, however, can perform poorly. If t_3 and t_4 are very large tables, the cost of writing the common spool can be very expensive. This plan can also result in out-of-spool situations. The cost of joining the large common spool to t_1 and t_2 can be costly as well, especially if the common spool needs to be redistributed or duplicated for the next join.

Our proposed join pushing optimization addresses these issues. The idea of the optimization is to break down a join with a UNION ALL view or derived table into multiple join operations. Each operation joins one or more UNION ALL branch to other relations (i.e., a base table or join relation). The result of joins is written to a common spool.

For our running example, the join pushing optimization gives the query optimizer the following additional options to execute the query. Option 1 is to join t_1 to each of the two branches in two separate join operations, write the result of these two joins to a common spool, and join the common spool to t_2 to produce query result. Option 2 is similar to Option 1 except that t_2 is the table pushed to join with branches. Option 3 joins t_1 and t_2 first. Then the result of joining t_1 and t_2 is joined to each of the branches of the UNION ALL derived table. Results of the two joins are written to a common spool.

Join pushing is cost based. It is implemented inside Teradata Database's lookahead join planner [5], which looks multiple joins forward to estimate the cost of joining relations in different orders. Therefore, the query optimizer can choose to push a single table or a join between tables (as demonstrated in the running example). The base plan is costed first. Following that, various pushing options are costed. Then, the best overall plan is picked. While planning and costing a pushing option, the total cost of joining to branches is accumulated. If the accumulated cost exceeds the cost of the base plan, join planning terminates and discards that option. Early termination helps reduce parsing time and optimize resource usage.

The optimization is flexible enough to apply when UNION ALL branches are all base tables or if some are base tables and others are not. A non-base table branch is a branch that has to spool its result first before being consumed. This can be the case, for instance, when a branch performs an aggregation on a table. Multiple non-base branches are combined together. They all write their results to a common spool. Then the common spool is handled by the optimization transparently as one single branch. As part of this process, the demographics (e.g., statistics) of the combined branches are collected and propagated so the query optimizer can cost properly.

One key advantage of join pushing is in exposing more efficient join methods. A base plan, for instance, may end up choosing product join. This is mainly because the UNION ALL derived table is handled as a black box by the join planner. Join pushing, on the other hand, opens up this black box to use efficient access path (e.g., single-AMP and index-based access path) and join methods (e.g., merge joins).

Join pushing allows iterative decomposition of a join into multiple joins. Firstly, a join between a UNION ALL view or derived table and other relations is planned as multiple joins. Then, each of these joins can itself be planned as a multistep join in the case it qualifies for PRPD joins [6], two-step CP joins [7], etc. PRPD and CP joins can further use multistep joins such as two-step nested joins [7]. This process can continue to decompose joins as long as such a plan is deemed more optimal by the query optimizer.

B. Geography Adjustment

In Teradata, the geography of a relation specifies how the relation is prepared for a join operation. This is an important

concept in Teradata Database's shared-nothing parallel architecture because rows need to be co-located on AMPs for join processing. The geography of a relation can be direct (i.e., rows are accessed directly), local (i.e., rows are spooled locally on AMPs), hashed (i.e., rows are redistributed across AMPs), or duplicate (i.e., rows are duplicated to all AMPs).

An important element of the join pushing optimization is the geography adjustment for the relation (which can be a base table or a join relation) that is being pushed to UNION ALL branches. Consider, for example, Option 3 of join pushing mentioned above. The result of $t_1 \bowtie t_2$ is pushed to join with the two UNION ALL branches. In other words, the result of $t_1 \bowtie t_2$ is needed twice; once to join with t_3 and another to join with t_4. The naive approach to do that is to perform $t_1 \bowtie t_2$ twice - each time producing the result in the geography appropriate for joining with the corresponding branch. But doing the join twice is neither necessary nor optimal.

Join pushing optimization is equipped with a technique to adjust the *geography* of the relation pushed to UNION ALL branches. The essence of geography adjustment is as follows. If geographies are identical, then the relation can be accessed with direct geography without any further redistribution or duplication. If they are not identical but one geography can reuse the other, then the source of joins is adjusted to avoid accessing base tables (or generating join results) multiple times. The rule of reusing a geography is that local geography can be used to produce hashed geography, and local and hashed geographies can be used to produce duplicate geography.

To explain further, consider an example with a table joining to a UNION ALL derived table with three branches. Assume that the pushed relation needs to be in local, duplicate, and local geographies when joining with the first, second, and third branches, respectively. Without geography adjustment, the base table needs to be scanned three times to produce three spools in the target geographies. But with adjustment, geographies change as follows. The geography for joining with the first branch remains local and its source remains the base table. The geography for joining with the second branch remains duplicate but its source becomes the local spool used in the first join. The geography for joining with the third branch becomes direct and its source becomes the local spool as well.

The algorithm also adjusts the *order* of joining branches with a pushed relation. Recall Option 3 of the pushed plan in the running example. Assume that in order to join with the first branch (i.e., $(t_1 \bowtie t_2) \bowtie t_3$), the result of t1 $t_1 \bowtie t_2$ needs to be duplicated. And to join with the second branch (i.e., $(t_1 \bowtie t_2) \bowtie t_4$), the result of $t_1 \bowtie t_2$ needs to be redistributed. In this case, the geography adjustment technique decides to do $(t_1 \bowtie t_2) \bowtie t_4$ first. Then it duplicates the hashed spool of $t_1 \bowtie t_2$ to prepare for the join of $t_1 \bowtie t_2$ with t_3. Note that if a pushed relation is a base table, single table conditions are applied only once when the base table is spooled for the first join. The spool with the reduced number of rows is used for subsequent joins which can improve the performance further.

These geography and order adjustments take place as part of generating pushed join plans and the cost of all adjustments is factored into the final plan cost. Therefore, the query optimizer has an accurate estimation upon which it can decide on the most optimal plan to choose.

IV. Experiments

This section presents a performance study conducted to examine the impact of the optimizations described above on query performance. The results of the experiments are reported for I/O counts, CPU time, and elapsed times. I/O count is the total number of logical I/Os. CPU time is the total (i.e., sum) CPU time of all AMPs and is measured in seconds. Elapsed time is the duration between the start time of a query and its first response time and is also measured in seconds.

A. Setup

The experimental apparatus consists of a 2-node Teradata 2800 appliance running Teradata Database 16.0. Each node has 512GB of memory, 2 14-core CPUs, and 44 AMPs.

We performed a set of experiments using the TPC-DS benchmark [8] with a scale factor of 1000 (i.e., 1 TB). TPC-DS represents a scalable and full-fledged retail model with a large number of tables. This benchmark is a good fit for our experiments because its schema already contains three large fact tables with similar structure for sales information: *store sales*, *web sales*, and *catalog sales*. We used the three tables in a UNION ALL derived table that joins to the *promotion* table.

We developed four variations of join queries between the UNION ALL derived table and the *promotion* table. Q_1 uses a PI-PI join condition. Q_2 uses a PI-NonPI join condition. Q_3 uses a NonPI-PI join condition. And Q_4 uses a NonPI-NonPI join condition. For fair comparisons, we used the same join condition on *item key* for the four queries and varied the physical design of tables. We developed two versions of each of the three fact tables. One version uses *customer key* as the PI and the other uses *item key* as the PI. Both keys are common in all three fact tables. Similarly, we developed two versions of the *promotion* table. One version has *promotion key* as the PI and the other has *item key* as the PI. Queries were executed independently with a cache flush between runs.

B. Results

In this discussion, BP refers to the **B**ase **P**lan that spools all UNION ALL branches before joins and OP refers to the **O**ptimized **P**lan that applies join pushing optimizations. Results of BP and OP are reported for **P**roduct **J**oin (PJ), **M**erge **J**oin (MJ), and **H**ash **J**oin (HJ) methods. For OP, we also present and discuss the query optimizer's cost-based choice of optimal plans.

1) Product Join: PJ scans one relation row by row and, for each row, it scans all rows of the other relation to find matching rows. PJ often duplicates the smaller relation to all AMPs before the join to co-locate its rows with the other (i.e., larger) relation. The larger relation is then accessed directly.

Table I shows the PJ results, which show that OP outperforms BP in all performance metrics. Whether the join is on the PI of the tables in the UNION ALL branches (Q_1 and Q_3) or not (Q_2 and Q_4), BP spools all branches locally in a common spool. For Q_1 and Q_3 that join on the PI of branches, BP keeps the common spool locally and accesses the pushed table with direct (Q_1) or duplicate (Q_3) geography. But for Q_2 and Q_4 that do not join on the PI of branches, BP needs to redistribute the common spool (Q_2) or duplicate the pushed table (Q_4). OP

outperforms BP because it avoids spooling branches locally and avoids extra redistribution or duplication. In some cases, performance improvements appear marginal; this is because the PJ cost itself is the dominating factor.

TABLE I: Product Join

Query	Elapsed Time		I/O Count		CPU Time	
	BP	OP	BP	OP	BP	OP
Q_1	186.26	175.18	1886326	1327779	10813	10216
Q_2	350.85	279.79	2228138	1674719	13284	13563
Q_3	202.1	167.09	1887143	1328079	10707	9978
Q_4	365.65	274.77	2230504	1677864	13692	13262

2) Merge Join: MJ requires rows of both relations to be sorted on the row hash of the join columns. Then it performs a row-hash match between rows of the two relations. MJ can access a table with a direct geography if the join condition is on the PI columns of the table.

Table II shows the MJ results. For Q_1 and Q_3 when the join is on the PI of tables in UNION ALL branches, OP provides significant performance improvements over BP because join pushing is able to expose the branches and access tables directly for MJ. That is to say, there is no spooling at all for the branches. On the contrary, BP has to spool all branches and sort the spool even though they are being joined on the PI column. Q_2 and Q_4 also perform better with OP because it avoids extra redistribution.

TABLE II: Merge Join

Query	Elapsed Time		I/O Count		CPU Time	
	BP	OP	BP	OP	BP	OP
Q_1	126.35	5.25	4249026	22690	8257	40
Q_2	286.15	192.42	4142719	3408441	11538	11329
Q_3	128.67	5.5	4252605	29020	8379	40
Q_4	290.57	222.85	4146524	3415243	11781	10751

3) Hash Join: Unlike MJ, HJ does not require relations to be sorted on the row hash of join columns. It builds a memory-resident hash table for the smaller relation, scans the larger relation, and looks up the hash table for matching rows. If the smaller relation itself is too large, it is fanned out into hash partitions. The other relation is fanned out symmetrically.

HJ results shown in Table III follow a similar pattern as merge join. But we can observe that the overall performance of both BP and OP is better when compared to merge join. This is attributed to the fact that the *promotion* table is small enough for its hash table to fit in memory with no fan out. This reduces the I/O count, which, in turn, reduces the elapsed time.

TABLE III: Hash Join

Query	Elapsed Time		I/O Count		CPU Time	
	BP	OP	BP	OP	BP	OP
Q_1	106.79	5.65	2282052	27348	6670	40
Q_2	219.56	159.59	2229591	1677816	5662	4916
Q_3	107.32	5.37	2282014	27370	6629	40
Q_4	223.29	161.88	2231173	1677626	5671	4914

4) Cost-based Plans: We also examined the query optimizer's choice of join method for OP to validate the cost-based aspect of the optimizations. MJ and HJ perform much better than PJ and PJ is never picked by the query optimizer. MJ and HJ perform almost the same for Q_1 and Q_3 and MJ is chosen for both. HJ is chosen for Q_2 and Q_4, which is evidently the right decision since it performs better than MJ for both queries.

V. Related Work

Optimizing UNION ALL join queries has been addressed in research and industry. Herodotou et al. [10] propose techniques for multiway joins over partitioned tables and suggest that they are applicable to UNION ALL queries as well. But the emphasis of the work in [10] is on partition (or equivalently branch) elimination, which is outside the scope of our paper.

IBM DB2 [11] has several optimizations for UNION ALL queries, including join pushdown. These optimizations, however, are rule based and are done at the query rewrite level. Teradata Database already has similar optimizations [12], but rule-based optimizations are limited to specific scenarios covered by the rewrite rules. In contrast, the optimizations we present in this paper are cost based and are integrated in the lookahead framework of the Teradata join planner. All reasonable join plans that can be generated by the query optimizer are considered and compared to choose the best plan. Su et al. [13] present join factorization in Oracle to pull out common tables from UNION ALL branches. Join factorization is cost based. But it does the exact opposite of join pushing and does not directly overlap with our proposed optimizations.

VI. Conclusions

In this paper, we discussed join queries over UNION ALL views and derived tables. We introduced a cost-based optimization for pushing joins and devised a technique for adjusting the geography of the relation pushed to UNION ALL branches to avoid unnecessary redistributions and duplications of data. We also presented the results of a performance study conducted to examine and evaluate the impact of these optimizations.

References

[1] A. Labrinidis and H. V. Jagadish, "Challenges and Opportunities with Big Data," *PVLDB*, vol. 5, no. 12, pp. 2032–2033, Aug. 2012.

[2] Teradata Unified Data Architecture. www.teradata.com/solutions-and-industries/unified-data-architecture.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE MSST*, 2010, pp. 1–10.

[4] QueryGrid. www.teradata.com/products-and-services/query-grid.

[5] Teradata, "SQL Request and Transaction Processing." [Online]. Available: www.info.teradata.com/index.cfm.

[6] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling Data Skew in Parallel Joins in Shared-nothing Systems," in *ACM SIGMOD*, 2008, pp. 1043–1052.

[7] M. Al-Kateb, P. Sinclair, G. Au, and C. Ballinger, "Hybrid Row-Column Partitioning in Teradata," *PVLDB*, vol. 9, no. 13, pp. 1353–1364, 2016.

[8] TPC-DS. www.tpc.org/tpcds.

[9] C. Ballinger and R. Fryer, "Born To Be Parallel: Why Parallel Origins Give Teradata an Enduring Performance Edge," *IEEE Data Eng. Bull.*, vol. 20, no. 2, pp. 3–12, 1997.

[10] H. Herodotou, N. Borisov, and S. Babu, "Query Optimization Techniques for Partitioned Tables," in *ACM SIGMOD*, 2011, pp. 49–60.

[11] C. Zuzarte, R. Neugebauer, N. Sutyanyong, X. Qian, and B. Rick, "Partitioning in DB2 Using the UNION ALL View," Jul. 11 2005. [Online]. Available: www.ibm.com/developerworks/data/library/techarticle/dm-0202zuzarte.

[12] A. Ghazal and W. McKenna, "Pushing Joins across a Union," Dec. 17 2009, US Patent App. 12/140,852. [Online]. Available: www.google.dj/patents/US20090313211.

[13] H. Su, R. Ahmed, A. Lee, M. Zait, and T. Cruanes, "Join Factorization of Union/Union All Queries," Jan. 5 2010, US Patent 7,644,062. [Online]. Available: www.google.dj/patents/US7644062.