Joins over UNION ALL Queries in Teradata®: Demonstration of Optimized Execution

Mohammed Al-Kateb, Paul Sinclair, Grace Au, Sanjay Nair,
Mark Sirek, Lu Ma, Mohamed Y. Eltabakh*
Teradata Labs
El Segundo, CA
{mohammed.al-kateb, paul.sinclair, grace.au, sanjay.nair, mark.sirek, lu.ma, mohamed.eltabakh}@teradata.com

ABSTRACT

The UNION ALL set operator is useful for combining data from multiple sources. With the emergence and prevalence of big data ecosystems in which data is typically stored on multiple systems, UNION ALL has become even more important in many analytical queries. In this project, we demonstrate novel cost-based optimization techniques implemented in Teradata Database for join queries involving UNION ALL views and derived tables. Instead of the naive and traditional way of spooling each UNION ALL branch to a common spool prior to performing join operations, which can be prohibitively expensive, we demonstrate new techniques developed in Teradata Database including: 1) Cost-based pushing of joins into UNION ALL branches, 2) Branch grouping strategy prior to join pushing, 3) Geography adjustment of the pushed relations to avoid unnecessary redistribution or duplication, 4) Iterative join decomposition of a pushed join to multiple joins, and 5) Combining multiple join steps into a single multisource join step. In the demonstration, we use the Teradata Visual Explain tool, which offers a rich set of visual rendering capabilities of query plans, the display of various metadata information for each plan step, and several interactive UGI options for end-users.

KEYWORDS

Joins over Union All, Query Optimization, Cost-Based Optimization

1 INTRODUCTION

The UNION ALL set operation is a mean of combining data from multiple sources. In traditional relational databases, a UNION ALL query resembles a logical table that combines rows of multiple physical tables. The UNION ALL operator is used in modern applications more frequently than ever, and a few emerging examples include: 1) ***Large Fact Tables in Data Warehousing:*** 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 is accomplished by combining the two tables as a view or derived table using UNION ALL. And 2) ***Heterogeneous Access in Big Data Applications:*** In the context of big data ecosystems, data of a single application may need to be integrated from multiple, and possibly, heterogeneous data sources. In the Teradata **U**nified **D**ata **A**rchitecture (UDA) [2], the current data of an application may be stored in a table in Teradata Database and the history data is stored in HDFS on a remote Apache Hadoop server. Data from both 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 [7].

Example 1: The following is our running example used throughout the paper. *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* $t_3(a_3, b_3, c_3, d_3)$ *with a single-table condition on* b_3*, and the other branch retrieving from* $t_4(a_4, b_4, c_4, d_4)$ *with a single-table condition on* b_4*. The query is as follows:*

SELECT a_1, a_2, a, c
FROM t_1, t_2, (SELECT a_3, c_3 FROM t_3 WHERE b_3=3
UNION ALL
SELECT a_4, c_4 FROM t_4 WHERE b_4=4) $dt(a, c)$
WHERE c_1=c_2 AND d_1=a;

When the UNION ALL derived table (the output from the inner queries over t_3 and t_4) joins to the outer relations in the query, a naive and straightforward plan is to write all UNION ALL branches to a common spool[1] and then join the common spool to the relations t_1 and t_2. However, spooling all branches can be costly. This can be the case, for instance, if the branches are very large fact tables with millions or billions of rows. Spooling all branches may also lead to out-of-spool scenarios and failure of execution. Joining the common spool to the other relations can be costly as well. In **M**assively **P**arallel **P**rocessing (MPP) systems like Teradata Database [3], the common spool may need to be redistributed for subsequent joins, which can add significant overhead to query execution. In short, queries involving joins over UNION ALL derived tables can become prohibitively expensive if not carefully optimized.

In Teradata, we have introduced and developed new optimizations to overcome the drawbacks of the naive execution plan highlighted above [1]. These optimizations are the core of this demonstration. More specifically, we introduced: 1) A cost-based optimization for pushing joins into the UNION ALL branches, 2) A UNION

*The author (meltabakh@cs.wpi.edu) is a visiting faculty at Teradata Labs.

[1]Spools are intermediate/buffer tables.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3193565

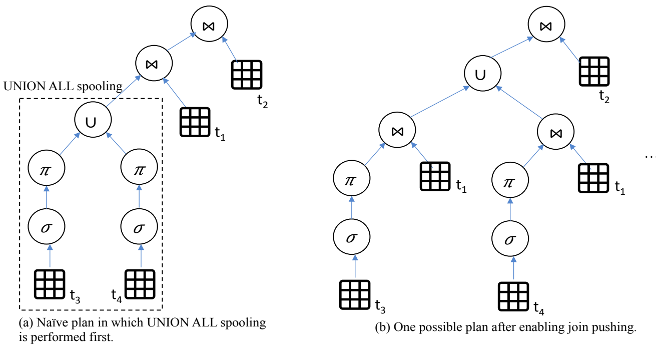


Figure 1: Join Pushing: The Generation of Alternative Plans.

ALL branch grouping mechanism prior to join pushing to reduce the number of branches, 3) Geography adjustment [2] of the pushed relations to avoid unnecessary redistribution or duplication, and 4) Iterative join decomposition of a single pushed join to multiple joins. The demo will also include a newly introduced optimization called *"multisource join"* that combines multiple join steps into a single multisource step.

In a nutshell, the introduced optimizations are complementary to each other, and combined altogether they open up opportunities for significant speedup for this type of Join-UnionAll queries. A key difference compared to the existing rule-based solutions is that the introduced optimizations are fully integrated within a cost-based optimizer, and thus these optimizations are picked only if they are effective.

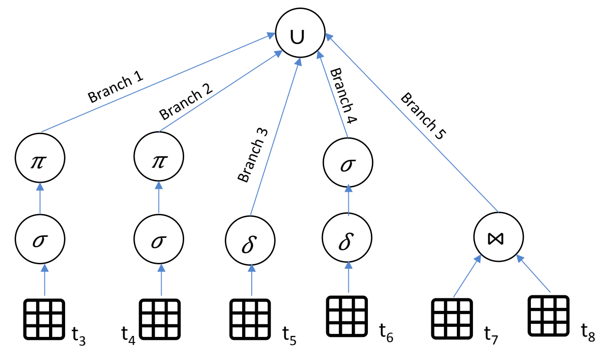
We believe the demo will be attractive to the audience since the addressed problem is applicable to a wide range of big data applications, and the introduced optimizations are beyond the standard textbook optimizations in database systems. Moreover, we plan to use the Teradata Visual Explain tool as the demo interface, which offers various high-end capabilities for plan rendering, detailed metadata display, and interactive exploration.

2 SYSTEM OVERVIEW

Teradata Architecture: Teradata Database is a highly parallel database engine with an advanced shared-nothing architecture that inherently enables horizontal scalability and fault tolerance. The building blocks of parallelism in Teradata Database are **P**arsing **E**ngine (PE) and **A**ccess **M**odule **P**rocessor (AMP). An AMP is associated with a portion of the data, and it consists of a collection of worker tasks running on one machine—which are threads—that perform operations on the data, e.g., sorting, aggregation, join.

A table in Teradata Database can have a **P**rimary **I**ndex (PI), **P**rimary **AMP** Index (PA), or **no P**rimary **I**ndex (NoPI). Rows of PI and PA tables are hash-distributed to AMPs while Rows of NoPI tables are randomly distributed to AMPs. Teradata Database offers advanced partitioning and store schemes for a given table across AMPs, which can be row, column, or a hybrid row-column stores that can seamlessly work together in a single system.

[2]The term "geography" refers to the layout of a relation on disk, e.g., a relation has specific storage ordering, partitioning, or distribution.



Branches 1 and 2 inherit the geography of their base tables t_1 and t_2, respectively. Branches 3, 4, and 5 will have their geography decided on the fly. Possible branch groupings include {Branch 3, Branch 4}, {Branch 3, Branch 5}, {Branch 4, Branch 5}, {Branch 3, Branch 4, Branch 5}.

Figure 2: Branch Grouping Optimization.

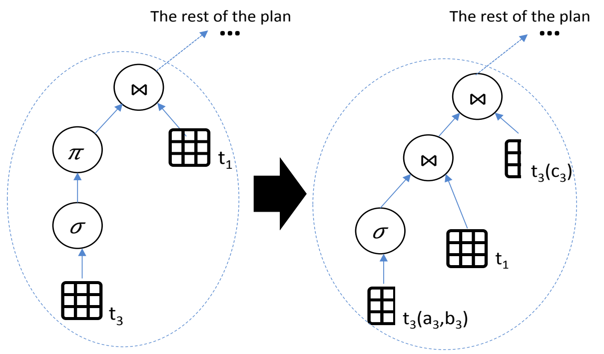
UNION All Optimizations: We briefly describe each of the developed optimizations. Assume $R \bowtie S$ denotes a join between two relations R and S, where S represents a UNION ALL view or derived table with n branches S_1, S_2, ..., S_n. The objective of the optimization techniques is to find a sequence of m joins where $m \leq n$ that is semantically equivalent to $R \bowtie S$ such that:

$$\left(\sum_{i=1}^{m} Cost\ (R \bowtie S_i) \right) < Cost\ (R \bowtie S)$$

where $Cost\ (R \bowtie S_i)$ is the cost of the plan that joins the relation R to the branch S_i, and $Cost\ (R \bowtie S)$ is the cost of the plan that joins the relation R to the entire UNION ALL relation S. The constraint $m \leq n$ reflects the fact that some of the branches of S can be grouped together before join pushing and handled as one branch.

• **Join Pushing:** This optimization aims at avoiding spooling all UNION ALL branches prior to joins if and when possible. Referring to Example 1, the naive base plan would execute the UNION ALL branches first and spool (materialize) the derived table *dt* into a common spool. This derived table then joins with the other tables, e.g., t_1 followed by t_2, as illustrated in Figure 1(a). As highlighted in Section 1, this plan can be very expensive and the UNION ALL spooling can be the bottleneck. With the *join pushing* optimization, the query optimizer is able to create many other alternatives and equivalent plans. One example is depicted in Figure 1(b), where table t_1 is pushed inside the UNION ALL branches. All other alternatives are also considered, e.g., pushing t_2 instead of t_1 and also joining t_1 and t_2 first and pushing their output inside the UNION ALL branches. The efficiency comes from the fact that join pushing is cost based, and the alternative plans are each costed and compared to the base plan to pick the least expensive plan.

• **Branch Grouping:** This optimization enables some UNION ALL branches to be handled individually, while others are grouped together prior to join pushing. This optimization takes place when some of the UNION ALL branches do not yet have a defined geography, i.e., layout and partitioning information has not yet been decided. This is the case when the table in a given branch is a non-base (computed-on-the-fly) table. For example, referring to



(a) One join between t_3 and t_1 (b) Decomposed join between t_3 and t_1

Assume Table $t3(a_3,b_3,c_3,d_3)$ is columnar with column groups $\{(a_3,b_3), (c_3), (d_3)\}$.
(a) All columns of t_3 are materialized together in one spool and then joined with t_1.
(b) Join is decomposed into two steps with access to the appropriate column groups.

Figure 3: Join Decomposition Optimization.

Figure 2, where a UNION ALL operator has five branches, branches 1 and 2 already have their geography set, which is the geography of the base tables t_1 and t_2, since the selection (σ) and projection (π) operators do not change the geography. In contrast, branches 3, 4, and 5 involve either grouping and aggregation (δ) or join ($\bowtie$) operators, which would require a new geography for their outputs. As illustrated in Figure 2, the *branch grouping* optimization kicks in under these situations. There are various grouping possibilities between branches 3, 4, and 5 to reduce the number of joins and minimize the overall cost. However, to avoid the exponential cost of trying all possible groupings, Teradata applies some heuristics to decide on the final grouping prior to the join pushing.

• **Iterative Join Decomposition:** Pushing the join into the UNION ALL branches is a kind of *join decomposition*. For example, referring to Example 1, a join with the derived table *dt* can now be implemented as multiple joins as in Figure 1 (b). The *iterative join decomposition* optimization can perform further decompositions after the push down. For example, consider the left branch of the UNION ALL operator ($t_3 \bowtie t_1$) in Figure 1(b). If t_3 happens to be a columnar table, then based on its column groups, the join with t_1 can be decomposed into multiple joins as illustrated in Figure 3. Join decomposition can be also activated if, for instance, table t_3 is a row-oriented table but with a secondary index on the joining column. In Teradata, secondary indexes are stored as separately from the base table and it is possible to have a full join with them before joining with the base table.

• **Geography Adjustment:** In Teradata, the geography of a relation specifies how the relation's rows are distributed in preparation for a join operation. The geography is determined based on several factors including the join type (predicate), relation cardinality, the join method picked by the query optimizer, etc. A relation can have one of four geographies, namely: `Direct` (i.e., rows are accessed directly from an AMP), `Local` (i.e., rows are accessed directly from an AMP after some pre-processing), `Hash` (i.e., rows are redistributed across machines), and `Duplicate` (i.e., rows are broadcasted to all machines).

Geography adjustment plays an important rule in UNION ALL join pushing to avoid unnecessary overheads, e.g., avoid unnecessary redistributions and duplications of the pushed relation. For example, considering the relations in Example 1, assume tables t_1 and t_2 join first to a specific spool, and then their results are pushed to each of the two UNION ALL branches. The first branch needs a `Duplicate` geography for the output of $t_1 \bowtie t_2$, while the second branch needs `Local` geography. Under *geography adjustment*, the system may alter the second-branch desired geography to either `Direct` and its source comes from broadcasted version, or `Duplicate` since the data is broadcasted anyway for the first branch.

• **Multisource Join:** This is a newly introduced optimization (not included in [1]). The main idea is that the pushed joins can be performed as one or more multisource steps. A multisource step reads rows from multiple sources (tables or spools) simultaneously. This means that instead of dispatching a separate join step for each join (and possibly incurring the same overhead multiple times), multiple joins can be grouped together as one multisource join step. A multisource join avoids multiple reads of the outer table, repeated building of a hash table for hash joins, and additional spooling and sorts. Distinct from the other introduced optimizations, multisource join is a rule-based optimization. It is applied as a heuristic rule whenever applicable. The optimization kicks in if the same join method is picked for the individual joins and the other relation has the same geography.

3 DEMONSTRATION PLAN

The demonstration plan includes the following items:

• **Learning Lessons to Audience:** A key learning lesson that we plan to emphasize from this demo to both database researchers and practitioners is that despite the decades of research in query optimization, still big data applications trigger the need for non-trivial optimizations. The demo will show examples of these novel optimizations fully integrated within the standard cost-based optimizers, and highlight the significant gain that can be achieved.

• **System Features:** The core system features to be demonstrated are the UNION ALL optimizations highlighted in Sections 1 and 2. The features are compared against both the naive plans, in which UNION ALL branches are evaluated first and spooled to a common spool, and the rule-based optimizations, in which the join pushdown blindly follows some rules instead of being cost based.

• **Datasets and Queries:** We plan to use datasets from the TPC-DS benchmark. This benchmark is a perfect fit for our experiments since its schema already contains three large fact tables with similar structure for sales information, namely *Store_Sales*, *Web_Sales*, and *Catalog_Sales*. Moreover, it has three other large tables for return information, namely *Store_Returns*, *Web_Returns*, and *Catalog_Returns*. Many of the benchmark queries use UNION ALL over these fact tables along with join operations, e.g., queries Q2, Q4, Q14, Q33, Q49, and Q76 [5]. We plan to also add some variations of these queries to create various scenarios serving our demo.

• **Advanced Visual Interface for Query Plans:** To visualize the query plans with and without the optimizations, we use the

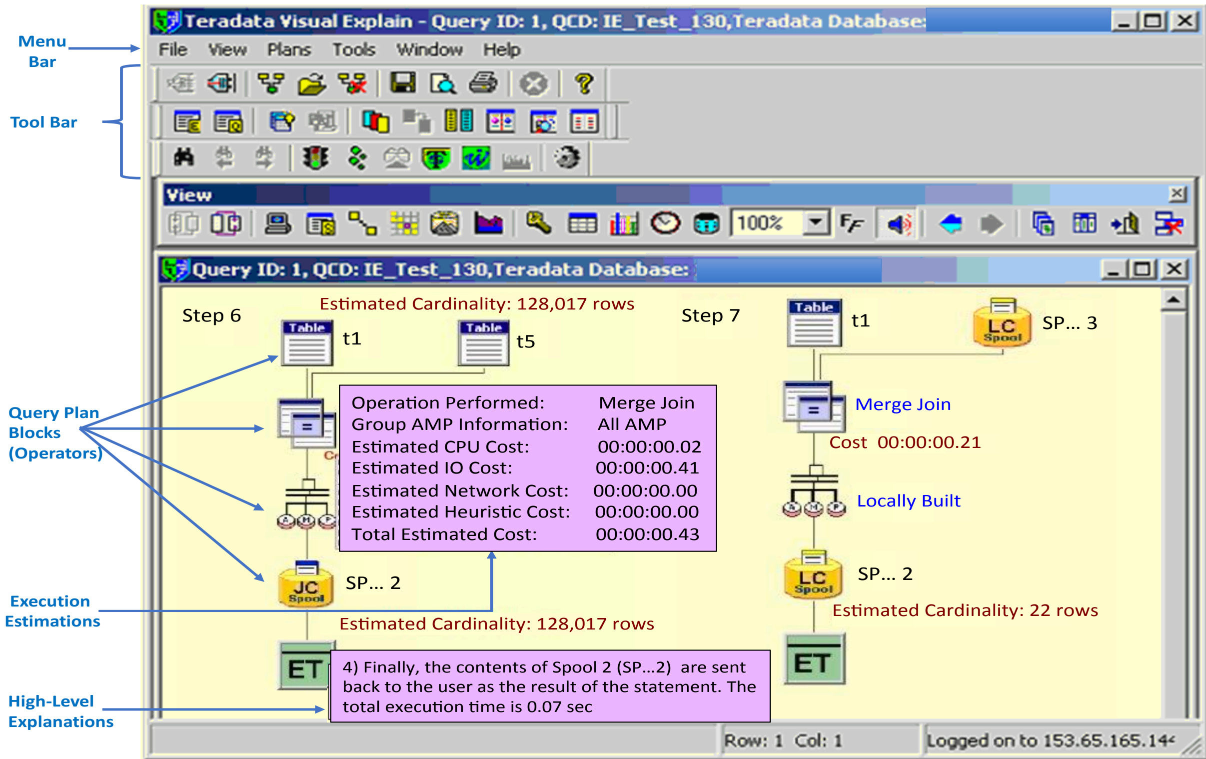


Figure 4: Teradata Visual Explain Tool: The Interactive Interface of our Demonstration.

Teradata Visual Explain tool (see Figure 4). It is an advanced tool that makes query plan analysis easier. The tool has a rich set of capabilities in the *Menu Bar* and *Tool Bar* with various functionalities and display options. The plan steps are captured and represented graphically in a form of a tree. For each step, pop-up windows can be displayed to show high-level text explanation, estimated execution statistics and cardinalities, underlying tables demographics, available indexes, and many others (see Figure 4). The tool can also visualize multiple windows at the same time to facilitate the comparison between different query plans.

• **Visual Performance Analysis:** The Teradata Visual Explain tool also enables monitoring and retrieving information about a query execution in real-time. Statistics, e.g., comparing the actual cost vs. the estimated cost, can be collected at the plan step level within a query. We use this capability to show the bottlenecks during execution; e.g., in a naive UNION ALL plan where the branches are very large tables, the bottleneck is the UNION ALL step.

4 RELATED WORK

Optimizing UNION ALL join queries has been addressed in both research and industry. Herodotou et al. [6] propose techniques for multiway joins over partitioned tables and suggest that they are applicable to UNION ALL queries as well. However, the emphasis of [6] is on partition (branch) elimination, which is different from the scope of our work. IBM DB2 [9] has several optimizations for UNION ALL queries, including join pushdown; however, these are rule based and are done at the query rewrite level. Teradata Database already has similar optimizations [4], but rule-based optimizations are limited to specific scenarios covered by the rewrite rules. In contrast, the optimizations we present in this demonstration are cost based and are integrated in the lookahead framework of the Teradata join planner.

On the other hand, Su et al. [8] present join factorization in Oracle to pull out common tables from UNION ALL branches. Join factorization does the exact opposite of join pushing and does not overlap with our optimizations.

In short, combining the five introduced UNION ALL optimizations and integrating them within a cost-based optimizer is unique to Teradata.

REFERENCES

[1] Mohammed Al-Kateb, Paul Sinclair, Alain Crolotte, Lu Ma, Grace Au, and Sanjay Nair. 2017. Optimizing UNION ALL Join Queries in Teradata. *ICDE* (2017).
[2] Teradata Unified Data Architecture. 2016. "www.teradata.com/solutions-and-industries/unified-data-architecture". (2016).
[3] Carrie Ballinger. 1994. Evolving Teradata Decision Support for Massively Parallel Processing with UNIX. In *SIGMOD*. 490.
[4] A.S. Ghazal and W.J. McKenna. 2009. Pushing Joins across a Union. (2009). www.google.dj/patents/US20090313211.
[5] GitHub: TPC-DS Queries. 2000. https://github.com/Agirish/tpcds. (2000).
[6] Herodotos Herodotou, Nedyalko Borisov, and Shivnath Babu. 2011. Query Optimization Techniques for Partitioned Tables. In *ACM SIGMOD*. 49–60.
[7] QueryGrid. 2014. www.teradata.com/products-and-services/query-grid. (2014).
[8] H. Su, R. Ahmed, A. Lee, M. Zait, and T. Cruanes. 2010. Join Factorization of Union/Union All Queries. (2010). US Patent 7,644,062.
[9] Calisto Zuzarte, Robert Neugebauer, Natt Sutyanyong, Xiaoyan Qian, and Berger Rick. 2005. Partitioning in DB2 Using the UNION ALL View. (July 11 2005). www.ibm.com/developerworks/data/library/techarticle/dm-0202zuzarte.