Grouping, Subsumption, and Disjunctive Join Optimizations in Oracle

Rafi Ahmed
Oracle America Inc.
Redwood City, CA, USA
rafi.ahmed@oracle.com

Krishna Kantikiran Pasupuleti
Oracle America Inc.
Redwood City, CA, USA
kanti.kiran@oracle.com

Sriram Tirupattur
Oracle America Inc.
Redwood City, CA, USA
sriram.tirupattur@oracle.com

Lei Sheng
Oracle America Inc.
Redwood City, CA, USA
lei.sheng@oracle.com

Hong Su
Oracle America Inc.
Redwood City, CA, USA
hong.su@oracle.com

Mohamed Ziauddin
Oracle America Inc.
Redwood City, CA, USA
mohamed.ziauddin@oracle.com

ABSTRACT

Query optimization must evolve with new workloads. As analytic and data warehouse workloads become more ubiquitous, optimization techniques that reduce the amount of data processed during query execution, enable shared computation and avoid expensive data access and joins must be rigorously explored. In this paper, we present aggregate-decomposition techniques as enhancements to an existing query transformation that performs grouping before joins. Consequently, the transformation generates more query rewrite candidates and can also be applied to a larger set of queries. Further, we introduce two new query transformations, i) subsumption of views and subqueries that explores opportunities for sharing computation and ii) union-all duplicator transformation for queries with disjunctive join predicates that removes the need for multiple data access and joins. These techniques are applicable to commonly noticed query patterns in customer workloads and provide significant performance benefit as indicated in our performance study. They have been implemented in Oracle RDBMS.

PVLDB Reference Format:
Rafi Ahmed, Krishna Kantikiran Pasupuleti, Sriram Tirupattur, Lei Sheng, Hong Su, and Mohamed Ziauddin. Grouping, Subsumption, and Disjunctive Join Optimizations in Oracle. PVLDB, 17(12): 4200 - 4212, 2024.
doi:10.14778/3685800.3685837

1 INTRODUCTION

Modern database workloads increasingly consist of analytics queries that require processing large amount of data. Over time, as more data is collected and uploaded to data warehouses and Hybrid Transactional and Analytical Processing (HTAP) systems, the queries written to analyze the data and gain insights need to be processed such that their performance remains optimal. Oracle query optimizer uses various query transformation techniques to rewrite queries into their equivalent but more optimal forms. Some of the common constructs that can be observed in such queries are subqueries, views and joins. Optimal forms of queries often require rewriting these constructs such that the new equivalent forms i) prune data during query execution as early as possible so that the overhead of subsequent computation is reduced, ii) avoid redundant computation through sharing of common parts, iii) open up access paths and operations that would otherwise not be possible, etc. In this paper we present some techniques built into the Oracle query optimizer that achieve the above stated objectives of query optimization.

Grouping rows before joins is a well-known technique [5, 22] used by query optimizers to improve the efficiency of joins by reducing the number of data rows that need to be subsequently joined. The reduction is achieved by placing a new group-by operation before the join. In Oracle RDBMS, the corresponding query transformation is called group-by placement (GBP), which uses a cost-based framework [2]. In prior work [5, 22] (and in Oracle RDBMS prior to our proposal), when aggregate expressions in queries refer to columns from multiple tables, there is lack of discussion on methods to split them further so that grouping can be done on joins of a subset of those tables. In this paper, we propose techniques that allow subsets of join graph to be grouped together leading to optimal plans that were hitherto not possible. When such grouping is done, the aggregates present in the original query must be rewritten to combine the partial aggregates (generated by grouping) to obtain the desired result (examples are shown in Section 2.1). However, when the aggregates contain complex arithmetic and/or conditional expressions, decomposing them to facilitate the GBP transformation is not straightforward. We show how this can achieved thus enabling additional combinations of grouping.Section 2 of this paper describes our techniques in detail.

Queries containing multiple subqueries and views are common in many benchmarks and customer workloads. Previous studies [17, 24] have suggested evaluating such common query sub-expressions once, materializing the results and re-using them. We propose a different method to rewrite queries that avoids materialization by unifying the common computation into a containing inline view. Section 3 discusses our technique that is applicable to specific forms of queries.

In customer workloads, we observed a class of queries that contain conjunctive join and disjunctive join predicates such that the conjunctive join alone produces many rows relative to the size of the input (i.e., it is an "expanding" join) but the disjunctive join later eliminates many of them. Expanding joins cause significant

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685837

degradation in performance when they generate huge number of intermediate rows. In Section 4 of this paper, we propose a new execution operator that facilitates simultaneous application of conjunctive and certain disjunctive join predicates by generating copies of rows on-the-fly, thus eliminating the expanding join in between.

The above-described transformation techniques have been implemented and released in Oracle RDBMS. To our knowledge, our work is the first that describes and utilizes these techniques driven by query rewrites during optimization.

1.1 Cost-Based Query Transformation Framework (CBQT) - Overview

A significant number of query transformations in Oracle are cost-based where logical transformation (also known as query rewrite) and physical optimization are combined to generate optimal execution plans. During cost-based transformation, a query is copied, logically transformed and its cost is calculated using the cost-based physical optimizer. This process is repeated multiple times applying a new set of transformations; and at the end, one or more transformations are selected and applied to the original query, if they result in a lower cost. The cost-based query transformation (CBQT) framework [2] provides a mechanism for the exploration of the state space generated by each transformation thus enabling the Oracle optimizer to select the optimal state in an efficient manner.

1.1.1 State Space Search Techniques. A fundamental question related to cost-based transformation is whether these transformations lead to a combinatorial explosion of alternatives that need to be evaluated and what the trade-off is between optimization cost and execution cost. The sources of multiple alternatives are the various transformations themselves as well as the set of objects (e.g., subquery blocks, view blocks, tables, table groups, join edges, predicates, etc.) on which each transformation may apply. If there are N independent objects on which a transformation T can apply, then 2^N possible alternative combinations can potentially be generated by the application of T. For simplicity, we denote a state as an array of bits, where the n^{th} bit represents whether the n^{th} object (e.g., subquery, view or table group, etc.) is transformed (a value of 1) or not transformed (a value of 0).

The complexity of a cost-based transformation is determined by the number of alternative combinations, the state space, which grows exponentially with the number of transformation objects. In order to limit the potential increase in optimization time, we use several techniques for searching the state space of various transformations. Some examples of search techniques are exhaustive, iterative, linear, two-pass, and perturbation walk. The CBQT framework automatically decides which search technique to use based on the number of elements to be transformed, the characteristics of the transformation, and the overall complexity of the query.

2 GROUP-BY PLACEMENT FOR QUERIES WITH COMPLEX AGGREGATES

In this section, we first give a brief overview of the existing group-by placement (GBP) transformation in Oracle in which all the tables inside aggregates are considered as one grouping unit, followed by describing novel decomposition techniques for complex aggregates that allow us to split these table groups further.

2.1 Background: Forms of GBP Transformation

Depending upon the columns of the tables that appear in aggregate expressions of a query and consequently the tables that early grouping is applied on, there are mainly three forms of GBP.

The first variant is called "coalesced grouping" (CG) in which early grouping is performed on the tables whose columns appear inside the aggregates. For example, consider query Q1 (and its variants) shown in Figure 1 in which tables T1 and T2 are joined on column 'x', and the aggregate is on column 'a' of T1. The query can be transformed using CG variant of GBP to obtain query Q1C. A new view V is created that groups the table T1 before it is joined with T2 in the outer query block. The group V generates all columns required by its outer query block (T1.x and T1.g) as well as the partial aggregate column 'Sa'. The original aggregate in the outer query block, SUM(T1.a), is modified to SUM(V.Sa) that adds the partial aggregates across groups and obtains the desired result.

The second variant is called "factored grouping" (FG) in which early grouping is performed on the tables whose columns do not appear in the aggregates. Such groups generate multiplicative "factors" that are used to scale the aggregates of the other tables. For query Q1, the transformed version is shown in Q1F of Figure 1. As grouping is now performed on table T2, the factors denoting the size of each group are gathered by the COUNT(*) aggregate of the view V. The original aggregate in the outer query is modified - SUM(T1.a * V.CNT) - to obtain the scaled sum for each group that matches on the join column 'x'.

The third variant combines the above ideas and creates both coalesced and factored grouping views in the outer query as shown in Q1CF of Figure 1.

Similar variants can be easily derived for other types of aggregates like MIN, MAX, COUNT in the original query and are not shown for brevity. It can be noted that the techniques are applicable even in the absence of a group-by clause or aggregate function in the original query (e.g., GROUP BY T1.g in Q1) and are trivially applicable to queries that contain a single table. Additionally, there are other specialized variants of GBP that are not discussed in this paper.

We introduce terminology that helps in formalizing the techniques later. The tables that appear in the aggregate expressions of a query (e.g., T1 in Q1) are called "Aggregating Tables" represented by set AT, and the rest of the tables (e.g., T2 in Q1) are called "Non-Aggregating Tables" represented by set NAT.

2.1.1 State Space of GBP Transformation. During cost-based GBP transformation, for each qualifying query block, multiple states are generated, costed and the cheapest among them is chosen to be applied on the query block. A join graph is constructed in which each table (or view) is a vertex and there is an edge between two tables if they appear in a join condition. For example, if the join condition is T1.x = T2.y, the join graph would carry an edge between vertices T1 and T2. The state space is defined over the set of tables present in a query block, the type (e.g., chain, star, cycle, clique, etc.) of the join graph and the applicability of one or more variants described earlier. For GBP, the join graph must be connected. The

```
Q1:
SELECT SUM(T1.a)
FROM  T1, T2
WHERE T1.x = T2.x and T2.k > 4
GROUP BY T1.g;

Q1C:
SELECT SUM(V.Sa)
FROM  T2, (SELECT T1.x, T1.g, SUM(T1.a) Sa
           FROM  T1
           GROUP BY T1.x, T1.g) V
WHERE V.x = T2.x and T2.k > 4
GROUP BY V.g;

Q1F:
SELECT SUM(T1.a * V.CNT)
FROM  T1, (SELECT T2.x, COUNT(*) CNT
           FROM  T2
           WHERE T2.k > 4
           GROUP BY T2.x) V
WHERE T1.x = V.x
GROUP BY T1.g;

Q1CF:
SELECT SUM(Vc.Sa * Vf.CNT)
FROM  (SELECT T1.x, T1.g, SUM(T1.a) Sa
       FROM  T1
       GROUP BY T1.x, T1.g) Vc,
      (SELECT T2.x, COUNT(*) CNT
       FROM T2
       WHERE T2.k > 4
       GROUP BY T2.x) Vf
WHERE Vc.x = Vf.x
GROUP BY Vc.g;
```

Figure 1: Query Q1 and its three variants generated by GBP transformation.

```
Q2:
SELECT T2.B2, T3.C3, SUM(T1.A1), COUNT(T1.D1)
FROM T1, T2, T3, T4
WHERE T1.B1 = T2.B2 and T2.D2 = T3.D3 and T3.C3 = T4.C4
GROUP BY T2.B2, T3.C3;
```

Figure 2: Query Q2 with table T1 in the aggregating table set AT.

variants form the substates of each state. If there are N tables in a query block and the join graph is a clique (i.e., strongly connected), there will be $3 * 2^N$ combinations generated (3 variants and 2^N states). Heuristics are used to limit the combinatorial explosion of the space; sometimes linear exploration strategy is used.

The set of states of the state space is generated by partitioning the tables into two sets C and F such that C necessarily includes all aggregating tables (set AT) i.e., C is a superset of AT and F contains the rest of the tables. Further, tables in each of the sets C and F must be connected to avoid introducing cartesian joins, if a view is to be generated for that set. The three variants (substates) of a state correspond to generating views only on C (coalesced view), only on F (factored view) and both C, F.

```
Q3:
SELECT SUM(T1.a * T2.b)
FROM  T1, T2
WHERE T1.x = T2.x
GROUP BY T2.g;

Q3C:
SELECT SUM(T1.a * V.Sb)
FROM  T1, (SELECT T2.x, T2.g, SUM(T2.b) Sb
           FROM T2
           GROUP BY T2.x, T2.g) V
WHERE T1.x = V.x
GROUP BY V.g;

Q3F:
SELECT SUM(T1.a * V.b * V.CNT)
FROM  T1, (SELECT T2.x, T2.g, T2.b, COUNT(*) CNT
           FROM T2
           GROUP BY T2.x, T2.g, T2.b) V
WHERE T1.x = V.x
GROUP BY V.g;
```

Figure 3: Coalesced and Factored grouping on table T2 of query Q3 that has a simple arithmetic expression in its aggregate.

Consider query Q2 shown in Figure 2. It has four tables one of which (T1) appears in the aggregates; so, AT = {T1}. The corresponding state space generated is shown in Table 1. Each state is represented by a set of bits; one bit for each table. A bit value "1" indicates the corresponding table will be included in C. Note that table T1 appears in C in all states as it belongs to AT. Hence, the state space is generated over the rest of the tables. Some states may have disconnected join graphs induced by either C or F; consequently, they have fewer substates.

2.2 Aggregates with Complex Arithmetic Expressions

When aggregate expressions contain columns from a subset of tables in the query, coalescing the aggregates involves placing the entire aggregate into the GBP view as shown in Q1C where $SUM(T1.a)$ along with the table T1 is moved into the view V. Consider query Q3 (Figure 3) whose aggregate expression contains columns from both the tables. Coalesced grouping is not straightforward because the entire aggregate $SUM(T1.a * T2.b)$ cannot be moved into a view along with any one table (and both tables cannot be moved as the outer query block must retain at least one table for the join to be valid). However, observing that the multiplication operation distributes over SUM, one of the tables (e.g., T2) can be moved to the coalesced view and the aggregate in the outer query block can be modified as shown in Q3C in Figure 3. Similarly, factored grouping can be applied to get the variant Q3F. (Q3C however, is likely to be more optimal than Q3F and is pruned.)

In the rest of this section, we describe an algorithm to use the idea of decomposition of expressions that contain a combination of arithmetic operators *, +, - and / to perform GBP transformation. The process consists of three steps.

(1) Normalization, in which the aggregate expressions of the query block are converted to a standard normal form.

Table 1: State Space of GBP for query Q2 (AT = {T1}, Join Graph = T1-T2-T3-T4).

T2 T3 T4	C	F	Connected Join Graph	No. of GBP substates
1 1 1	{T1, T2, T3, T4}	Ø	C	1
1 1 0	{T1, T2, T3}	{T4}	C, F	3
1 0 1	{T1, T2, T4}	{T3}	F	1
1 0 0	{T1, T2}	{T3, T4}	C, F	3
0 1 1	{T1, T3, T4}	{T2}	F	1
0 1 0	{T1, T3}	{T2, T4}	-	0
0 0 1	{T1, T4}	{T2, T3}	F	1
0 0 0	{T1}	{T2, T3, T4}	C, F	3

```
Q4:
SELECT SUM((T1.a + T2.b)*(T2.c - T1.d))
FROM  T1, T2
WHERE T1.x = T2.x
GROUP BY T2.g;
```

Figure 4: Query Q4 with a complex aggregate.

(2) Generation of GBP states conforming to the proposed new rules that identify tables that can be moved into the GBP view(s).
(3) Transformation of the normalized aggregates in the outer query block and generation of corresponding aggregates inside the view query block(s) using a set of new proposed rules (based on the context of each state of GBP).

2.2.1 Aggregate Expression Normalization. Consider query Q4 from Figure 4 that contains the following complex arithmetic expression in its aggregate.

$$SUM((T1.a + T2.b) * (T2.c - T1.d))$$

Applying coalesced grouping on such a query is not straightforward as the aggregate needs to be correctly decomposed in order to move one of the tables into a coalesced view. However, considering that the distributive property can be used in expressions of the form shown in Q3, if the above expression is converted into such form, a similar technique could be applied.

Given an arbitrary arithmetic expression inside an aggregate (using the arithmetic operators *, +, - and /), the GBP transformation first converts it into a "sum of products (SOP)" normal form. The normalized expression consists of "product" terms that are combined using the addition or subtraction operator. Each product term, in turn, consists of base terms that are combined using multiplication and/or division operators. Base terms can either be columns of a table or other constants (including bind variables etc.). For example, the following normalized expression consists of three product terms combined using "+" and "-" operators (multiplications are implicitly shown).

$$xyz + ab - pq/r$$

The normalized form of the SUM expression in query Q4 is as follows.

$$SUM(T1.a * T2.c - T1.a * T1.d + T2.b * T2.c - T2.b * T1.d)$$

2.2.2 Generation of Additional GBP States. Normalization facilitates generation of additional valid GBP states that expand the scope of GBP transformation. As described in Section 2.1.1 (and shown in Table 1), regular valid GBP states all have the property C ⊇ AT. However, no such state is possible for Q4 as it requires moving all tables from the outer block into the view. So, we introduce the following new rules.

- In a GBP state, C must contain at least one table from AT, i.e., C ∩ AT ≠ ∅.
- If the aggregation operator is SUM, for any table Tx ∈ C, if a column reference of Tx appears in the denominator of a division ("/") operator in any product term of the SOP expression, the state is invalid.

The first rule relaxes the condition C ⊇ AT so that new GBP states can be generated. The second rule is necessary because division is not distributive over addition for denominators (i.e., $x/(y+z) \neq x/y + x/z$). An example showing the application of these rules to generate additional states is given in Section 2.2.4.

We use other heuristics to limit the number of states that need to be costed, like checking if a state introduces grouping key columns contain high NDVs (number of distinct values) and discarding non-promising states etc. The details of all such heuristics are outside the scope of this paper.

2.2.3 Transformation of Aggregates. When views are created according to a GBP state and appropriate tables are moved into them, the aggregate expressions in the outer query block must be transformed accordingly to retain the correct semantics. The transformation depends on i) the GBP state generated (i.e., a specific partitioning of tables into C and F) and ii) the associated aggregation operation (SUM, MIN etc.) along with the SOP expression inside it. It is done as follows.

Each product term "P" in the SOP expression is examined and transformed independently based on the aggregation operator. (For brevity, we write "a table of P" to mean "a table whose columns appear in the term P".)

Aggregate function SUM():

- If no table of P is in C, the product P must be scaled by the cardinality of each group formed by the GBP view. Hence, a "count(*) CNT" aggregate is added to the GBP view (V) and P is replaced by "P * V.CNT".
  E.g., P = T1.a*T2.b is modified to T1.a*T2.b*V.CNT when T1, T2 ∉ C.

```
Original Query:
SELECT SUM(T1.a * T2.b - T3.x * T4.y * T5.z)
FROM T1, T2, T3, T4, T5
...

GBP State C = {T3, T4}:
SELECT SUM(T1.a * T2.b * V.CNT - V.S * T5.z)
FROM T1, T2, T5, (SELECT COUNT(*) CNT, SUM(T3.x * T4.y) S
                  FROM T3, T4
                  ..) V
...
```

Figure 5: Transformation of SUM aggregate for GBP state C = {T3, T4}. (Query shown partially)

- If at least one table of P is in C, P is split into two terms P1 and P2 such that P1 ∈ C and P2 ∉ C. Using the distributive property, an aggregate "SUM(P1) S" is created inside the GBP view and P1 in P is replaced by the column S from the GBP view.
  E.g., P = T3.x*T4.y*T5.z is modified to V.S*T5.z where V.S = SUM(T3.x*T4.y), when T3, T4 ∈ C and T5 ∉ C.

The complete aggregate transformation using both the rules is shown in Figure 5.

Aggregate function MIN() (similar for MAX()):

- If all tables of P are in C, an aggregate "MIN(P) M" is created inside the GBP view and P is replaced by column M from the view.
  E.g., T3.x*T4.y (T3, T4 ∈ C) is replaced by V.M where M = MIN(T3.x*T4.y).
- If at least one table of P is not in C, column references in P that belong to C are added to grouping keys of the GBP view.
  E.g., T3.x*T2.b (T3 ∈ C, T2 ∉ C) is modified to V.x*T2.b where V.x = T3.x.

It is possible to introduce MIN/MAX aggregates inside the GBP view (instead of grouping keys) when additional constraints are present on columns (e.g., T2.b above is known to be always positive). We don't describe them further. The complete aggregate transformation using both the rules is shown in Figure 6.

Aggregate function COUNT(): The aggregate COUNT in the outer query block is transformed into SUM as it needs to add the counts for each group that are produced by the GBP view. However, the aggregate COUNT added to the GBP view can compute the arithmetic expression using an addition operator because the computed value is not relevant for the result. For example $COUNT(T1.a * T2.b)$ is the same as $COUNT(T1.a + T2.b)$. We skip giving a specific example as it can be derived similar to the other aggregate functions.

2.2.4 New State Space of GBP Transformation. To illustrate the new state space of GBP, we take a query Q5 obtained by adding another table T3 to query Q4 (Figure 4). The query and its normalized aggregate expression are shown in Figure 7. Table 2 depicts the new state space generated. States {1 1 1} and {0 0 0} are not shown as they are not valid. Without decomposition, only state {1 1 0} is possible (due to the rule C ⊇ AT). The states shown in bold are the new states generated that conform to the relaxed rule C ∩ AT ≠ ∅.

```
Original Query:
SELECT MIN(T3.x * T2.b + T3.x * T4.y)
FROM T1, T2, T3, T4, T5
...

GBP State C = {T3, T4}:
SELECT MIN(V.x * T2.b + V.M)
FROM T1, T2, T5, (SELECT T3.x x, MIN(T3.x * T4.y) M
                  FROM T3, T4
                  ...
                  GROUP BY T3.x..) V
...
```

Figure 6: Transformation of MIN aggregate for GBP state C = {T3, T4}. (Query shown partially)

```
Q5:
SELECT SUM((T1.a + T2.b)*(T2.c - T1.d))
FROM  T1, T2, T3
WHERE T1.x = T2.x
  AND T2.y = T3.y
GROUP BY T2.g;

Normalized Aggregate:
SELECT SUM(T1.a*T2.c + T2.b*T2.c - T1.a*T1.d - T2.b*T1.d)
```

Figure 7: Query Q5 and its normalized aggregate expression.

Table 3 shows how the product terms in the aggregate of the outer query block are transformed for each state, according to the rules in Section 2.2.3. Factored views (not shown) are generated in the same way. In the third and the fourth columns of the table, each modified product term (P) in the outer query block and its corresponding aggregate term in the view are shown in the same line. For instance, in state {1 0 0}, the product term V.Sa * T2.c derives its value V.Sa, from the aggregate "SUM(T1.a) Sa" of the view V.

2.3 Aggregates with Conditional Expressions

In this section, we address another category of aggregate expressions that include conditional logic. These are commonly seen in customer workloads and benchmarks. Consider query QC1 from Figure 8 that has a CASE expression inside the aggregate. Originally, as both tables appear inside the aggregate, they are considered as aggregating tables. Thus, the set AT = {T1, T2}. This prevents GBP transformation from happening. However, it can be observed that the columns of table T1 only appear in conditional checks and are not aggregated. Hence, it is possible to perform the transformation and obtain query QC2. (Another variant can also be generated with a factored view on T1).

The proposed enhancement consists of 1) identifying tables in aggregate expressions that appear in conditions and 2) generating new GBP states and transforming the aggregates accordingly.

2.3.1 Pre-processing Conditional Expressions. The tables in an expression are partitioned into a set CT (for "Condition Tables") containing those that participate in conditional checks and another set AT containing those that participate in aggregation. In query QC1, CT = {T1} and AT = {T2}. Figure 9 indicates how the sets CT and AT are derived for a nested conditional aggregate. If a table appears in

Table 2: New State Space of GBP for Query Q5 (AT = {T1, T2}, Join Graph = T1-T2-T3). New states are in bold.

T1 T2 T3	C	F	Connected Join Graph	No. of GBP Substates
1 1 0	{T1, T2}	{T3}	C, F	3
1 0 1	{T1, T3}	{T2}	F	1
1 0 0	{T1}	{T2, T3}	C, F	3
0 1 1	{T2, T3}	{T1}	C, F	3
0 1 0	{T2}	{T1, T3}	C	1
0 0 1	{T3}	{T1, T2}	C, F	0 (C ∩ AT = ∅)

Table 3: Aggregate (SUM) Transformation for Coalesced View (V) in New GBP States of Table 2.

T1 T2 T3	Tables in V	Transformed Terms in Outer Query Block	Corresponding Aggregate Terms in V
1 0 0	T1	V.Sa * T2.c + T2.b * T2.c * V.CNT – V.Sad - T2.b * V.Sd	SUM(T1.a) Sa COUNT(*) CNT SUM(T1.a * T1.d) Sad SUM(T1.d) Sd
0 1 1	T2, T3	T1.a * V.Sc + V.Sbc – T1.a * T1.d * V.CNT – V.Sb * T1.d	SUM(T2.c) Sc SUM(T2.b * T2.c) Sbc COUNT(*) CNT SUM(T2.b) Sb
0 1 0	T2	T1.a * V.Sc + V.Sbc – T1.a * T1.d * V.CNT – V.Sb * T1.d	SUM(T2.c) Sc SUM(T2.b * T2.c) Sbc COUNT(*) CNT SUM(T2.b) Sb

```
QC1:
SELECT SUM(CASE WHEN T1.a = 2 THEN T2.b ELSE T2.c END)
FROM  T1, T2
WHERE T1.x = T2.x AND T2.y > 2
GROUP BY T1.g;

QC2:
SELECT SUM(CASE WHEN T1.a = 2 THEN V.Sb ELSE V.Sc END)
FROM T1, (SELECT SUM(T2.b) Sb, SUM(T2.c) Sc
          FROM T2
          GROUP BY T2.x) V
WHERE T1.x = V.x AND V.y > 2
GROUP BY T1.g;
```

Figure 8: Query QC1 transformed into QC2 using a coalesced GBP view.

```
SUM(CASE
     WHEN
       (CASE WHEN T1.b > 5     <== Condition column
             THEN T1.c         <== Condition column
             ELSE T1.a         <== Condition column
        END) = 5
     THEN
       (CASE WHEN T2.b < 5     <== Condition column
             THEN T3.x         <== Aggregate column
             ELSE T3.y         <== Aggregate column
        END)
     ELSE
       T3.y                    <== Aggregate column
   END)
```

Figure 9: Condition Table and Aggregating Table sets for nested conditional aggregate.

both the sets CT and AT, the decomposition is not performed as it adds more complexity to splitting the expression.

2.3.2 Generation of New GBP States. We introduce the following rule to expand the scope of GBP based on the identified sets, CT and AT. For every partitioning C, F generated (such that C ⊇ AT),

- If C contains *all* tables of CT (i.e., C ⊇ CT ∪ AT) or C contains no tables of CT (i.e., C ∩ CT = ∅), GBP substates are generated with group-by views on C, on F, and both C, F.
- If C contains *some* (but not all) tables of CT (i.e., (C ∩ CT) ⊂ CT), GBP state is generated with group-by view on F only.

The first rule opens the possibility of splitting the conditional expression and moving all tables of CT together. The second rule ensures coalesced views are not generated that would require complex splitting of the aggregate because a subset of tables in CT would need to be moved. So, we either move all the tables in CT into a coalesced view or move none of them. As stated in Section 2.2.2, there are other heuristics we use to discard non-promising states.

2.3.3 New State Space. Table 4 gives an illustration of the new state space produced by partitioning the tables in the conditional aggregate of an example query QC3 shown in Figure 10.

Table 4: New State Space of GBP for Query QC3 (AT = {T3}, CT = {T1}, Join Graph = T2-T1-T3). New states are in bold.

T1 T2	C	F	C vs CT	Connected Sets	No. of GBP Substates
1 0	{T1, T3}	{T2}	C has *all* tables of CT	C, F	3
0 1	{T2, T3}	{T1}	C has *no* tables of CT	F	1
0 0	{T3}	{T1, T2}	C has *no* tables of CT	C, F	3

```
QC3:
SELECT SUM(CASE WHEN T1.b=4 THEN T3.b+1 ELSE T3.a END)
FROM T1, T2, T3
WHERE T1.x = T2.x
  AND T1.y = T3.y
  AND T2.b > 2
GROUP BY T1.g;
```

Figure 10: Query QC3 with conditional expression in its aggregate.

The transformation of aggregates in the outer query block is relatively straightforward.

- When C has *all* tables of CT, the entire conditional aggregate is moved into the GBP view.
- When C has *no* tables of CT, simple aggregates are placed in the GBP view and the conditional aggregate is retained in the outer query block while replacing the appropriate columns with references to the view as shown in Figure 8.

To conclude, we proposed novel methods of "aggregate decomposition" and "state generation" that ensure more efficient grouping can be performed in queries.

3 SUBSUMPTION OF VIEWS AND SUBQUERIES

In Oracle terminology, a query block that appears in the FROM clause of another query block is called a 'view', whereas a query block that appears in the WHERE, SELECT, or HAVING clauses is called a 'subquery'. This section describes a technique for subsuming multiple views or subqueries into a single view that replaces them. Subsumption thus reduces multiple table accesses and joins thereby improving query performance. It is a heuristic query transformation that is used in the presence of the qualifying conditions.

Query blocks with the following patterns are commonly found in customer workloads (esp. app-generated queries) and benchmarks.

- All views/subqueries appear in the same outer query block.
- Each view/subquery has aggregation with no group-by clause, i.e., it produces exactly one row.
- All views/subqueries have identical tables and join predicates.

Consider query QV1 in Figure 11 that satisfies the above conditions. The views have the same join predicate "*F.a = D1.a*" and contain different filter predicates. The subsumption algorithm performs the following after creating a new subsuming view based on the given views.

- For every SELECT item of each subsumed view, a CASE statement is formed encapsulating its WHERE clause filter predicates and arguments of its aggregate function.

```
QV1:
SELECT *
FROM (SELECT AVG(F.x) AV
      FROM F, D1
      WHERE F.a = D1.a AND F.c = 5
        AND F.g = 69) V1,
     (SELECT SUM(F.y) SM
      FROM F, D1
      WHERE F.a = D1.a AND F.c = 5
        AND F.g = 40) V2,
     (SELECT COUNT(F.z) CN
      FROM F, D1
      WHERE F.a = D1.a AND F.c = 5
        AND F.g = 2
        AND D1.m > 8) V3;

QV2:
SELECT *
FROM (SELECT AVG(CASE WHEN F.g = 69
                 THEN F.x ELSE NULL END) AV,
             SUM(CASE WHEN F.g = 40
                 THEN F.y ELSE NULL END) SM,
           COUNT(CASE WHEN F.g = 2 and D1.m > 8
                 THEN F.z ELSE NULL END) CN
      FROM F, D1
      WHERE F.a = D1.a
        AND F.c = 5           /* Factorization */
        AND F.g IN (69, 40, 2)  /* Unification */
     ) SV;
```

Figure 11: Query QV1 transformed into QV2 using subsumption.

- The filter predicates that appear in the views are factored and unified into the subsuming view.

Factorization: The common filter predicates are removed from subsumed views and added to the WHERE clause of the subsuming view.

Unification: Different filter predicates originating from subsumed views are combined into a disjunctive or a range filter predicate and added to the subsuming view.

The objective of factorization and unification is to make joins and aggregations more efficient in the subsuming view. QV2 in Figure 11 is the corresponding transformed query obtained using the above techniques. The views V1, V2 and V3 have been subsumed into a single view SV. The filter predicate "*F.c = 5*" common to all three views has been factored out and placed in the subsuming view SV. The three predicates, "*F.g = 69*", "*F.g = 40*", and "*F.g = 2*", are unified as "*F.g IN (69, 40, 2)*" and placed in the subsuming view, but the individual filter predicates (e.g., "*F.g = 40*") appear in the

```
QS1:
SELECT (SELECT SUM(F.x)
        FROM F
        WHERE F.c between 1 and 5) SM1,
       (SELECT AVG(F.x)
        FROM F
        WHERE F.c between 6 and 10) AV,
       (SELECT COUNT(F.x)
        FROM F
        WHERE F.c between 11 and 15) CN,
       (SELECT SUM(F.x)
        FROM F
        WHERE F.c between 16 and 20) SM2
FROM T
WHERE T.k = 1;

QS2:
SELECT *
FROM (SELECT SUM(CASE WHEN F.c between 1 and 5
                      THEN F.x ELSE NULL END) SM1,
             AVG(CASE WHEN F.c between 6 and 10
                      THEN F.x ELSE NULL END) AV,
             COUNT(CASE WHEN F.c between 11 and 15
                        THEN F.x ELSE NULL END) CN,
             SUM(CASE WHEN F.c between 16 and 20
                      THEN F.x ELSE NULL END) SM2
      FROM F
      WHERE F.c between 1 and 20
                          /* Range Unification */
     ) SV, T
WHERE T.k = 1;
```

Figure 12: Query QS1 transformed into QS2 using subsumption.

corresponding CASE statement. Further, the non-factored and non-unified predicates (e.g., "*D1.m > 8*") also appear in the corresponding CASE statements. Note that SV returns a single row (same as QV1).

Figure 12 shows an example of subsuming subqueries where the filter predicates of the four SELECT-clause subqueries have been unified as a single range filter predicate into one view SV. Based on the similarities among various aggregate functions and filter predicates in the subsumed views, the efficiency can be further improved by introducing a group-by view that pre-computes aggregates, as shown in Figure 13, ensuring that the conditional aggregates which tend to be more expensive are performed on fewer rows (after grouping).

Grouping and pre-computation is applied when the following heuristic criteria are met. A configurable target value (typically ranging between 5 and 10) called "Grouping Reduction Factor" (GRF) is used to determine if grouping is likely to be beneficial (i.e., too many groups are not generated).

- The subsuming view refers to a single table (e.g., T).
- The number of aggregate functions in the group-by view is less than or equal to half of the number of aggregates in the subsuming view (i.e., at least 2X reduction).
- There are *unified* filter predicates in the WHERE clause of the subsuming view.
- If there are N columns in the table (T), the product of their scaled NDVs satisfy the following condition (assuming the columns are independent).

$$\prod_{i=1}^{N} NDV(C_i) < Cardinality(T)/GRF \quad (1)$$

  i.e., the estimated cardinality of group-by is less than the desired target.

```
QSG2:
SELECT *
FROM (SELECT SUM(CASE WHEN GV.c between 1 and 5
                      THEN GV.Vsm ELSE NULL END) SM1,
             SUM(CASE WHEN GV.c between 6 and 10
                      THEN GV.Vsm ELSE NULL END) /
             SUM(CASE WHEN GV.c between 6 and 10
                      THEN GV.Vcn ELSE NULL END) AV,
             SUM(CASE WHEN GV.c between 11 and 15
                      THEN GV.Vcn ELSE NULL END) CN,
             SUM(CASE WHEN GV.C between 16 and 20
                      THEN GV.Vsm ELSE NULL END) SM2
      FROM (SELECT SUM(F.x) Vsm, COUNT(F.x) Vcn, F.c
            FROM  F
            WHERE F.c between 1 and 20
                              /* Range Unification */
            GROUP BY F.c) GV          /* Grouping */
     ) SV, T
WHERE T.k = 1;
```

Figure 13: Query QS2 transformed into QSG2 after Grouping.

For example, if we assume a uniform distribution, scaled NDV of column T.b would be NDV(T.b) * ((B2 – B1)/(Max(T.b) – Min(T.b))) where B1 and B2 are the minimum and maximum values in the unified range of the filter predicates. If the unified predicate is an IN -list, it is the count of elements in the list. In query QS2 from Figure 12, the four aggregates in the CASE statements of SV are defined over the same column and there is a unified predicate; hence a group-by view GV has been created in SV (query QSG2 in Figure 13).

TPC-DS queries Q9, Q28 and Q88 are candidates for the Subsumption optimization.

4 UNION-ALL DUPLICATOR FOR DISJUNCTIVE QUERIES

In this section, we describe a novel technique that uses a "union-all duplicator" operator for optimizing query blocks containing expanding joins and a specific pattern of disjunctive predicates. Many customer queries contain the following type of join predicates:
T1.x = T2.x and (T1.y IS NULL OR T1.y = T2.y),
where T1 and T2 are large tables and the inner join on T1.x = T2.x is expanding. In such cases, the transformation of such queries using the *union-all duplicator* operator results in significant performance gain. We introduce relevant terminology before describing the transformation.

- *Expanding join*: When columns in a join predicate have very low NDVs and the column values have significant overlap, the cardinality of the join is much larger than the cardinality of either of the input tables. Such joins cause degradation in performance as the large intermediate result must be processed and may even spill to disk. Parallel plans may suffer even more as such intermediate rows tend to

```
QUEx:
SELECT prod, qtr, amount
FROM   sales
UNION_ALL DUPLICATOR
   (amount FOR qtr IN (Q1, Q2));
```

Table sales:

prod	Q1	Q2
Shoes	200	4000
Jeans	700	NULL

Result of QUEx:

prod	qtr	amount
Shoes	Q1	200
Shoes	Q2	4000
Jeans	Q1	700

Figure 14: Result of a query with Union-All Duplicator (UAD) operator.

exhibit large skew affecting the distribution of rows among parallel threads.

- *Post-join filter predicate*: This predicate must be applied to the result of a join. Post-join filter predicates cannot be used for forming join keys. For example, the disjunctive predicate, (T1.y IS NULL OR T1.y = T2.y), shown earlier must be applied as a post-join filter after the join between T1 and T2 has been performed on the conjunct T1.x = T2.x.
- *SYS_OP_MAP_NONNULL*: This is an operator in Oracle which maps values of different data types and NULLs into unique values of RAW data type. It allows matching on NULL values. Henceforth, we will use its shortened form *OP_MAP_NN*. *OP_MAP_NN(NULL)* is a fixed value *FF*.
- *NVL2*: This operator takes three arguments and returns its second argument if its first argument is non-NULL; otherwise it returns its third argument.

4.1 The Union-All Duplicator Operator

The union-all duplicator (UAD) is an operator that is applied to a table expression. This operator converts a portion of a row into column data and duplicates the rest of the row. Data in a set of user-specified column names are rotated as column values under a new user-specified column name. For example, consider query QUEx on table *sales* shown in Figure 14. For each row in *sales* the operator generates two new rows, each containing the original row's column values (e.g., 200, 4000 in the first row) placed in a new user-specified column, *amount*. However, new rows are not generated for NULL values in the original row's columns. Hence, there is only one row output for the product "Jeans". During execution, the UAD operator evaluates the query block, fetches one row, and produces multiple rows without needing to access the table again (or perform joins again if multiple tables are involved).

```
QU1:
SELECT T1.a, T2.b
FROM  T1, T2
WHERE T1.n = T2.n
  AND (T1.u is NULL OR T1.u = T2.u);

QU2:
SELECT T1.a, T2.b
FROM  T1, T2
WHERE T1.n = T2.n AND T1.u is NULL
UNION ALL
SELECT T1.a, T2.b
FROM  T1, T2
WHERE T1.n = T2.n AND T1.u = T2.u;
```

Figure 15: Query QU1 with an expanding join on *n*, and its OR-expanded version QU2.

```
QU3:
SELECT T1.a, T2.b
FROM T1,
      (SELECT b, n, K
       FROM (SELECT b, n
              NVL2(u, OP_MAP_NN(u), NULL) P1,
              OP_MAP_NN(NULL) P2
             FROM T2) V1
       UNION_ALL DUPLICATOR
        (K FOR KP IN (P1, P2))) V2
WHERE T1.n = V2.n AND OP_MAP_NN(T1.u) = V2.K;

QU4:
SELECT T1.a, V2.b
FROM T1,
      (SELECT b, n, OP_MAP_NN(u) K
       FROM T2
       WHERE u IS NOT NULL
       UNION ALL
       SELECT b, n, OP_MAP_NN(NULL) K
       FROM T2) V2
WHERE T1.n = V2.n AND OP_MAP_NN(T1.u) = V2.K;
```

Figure 16: QU1 transformed into QU3 using UAD. QU4 is a simulation of QU3.

4.2 Alternate Strategies

Consider query QU1 in Figure 15 that has an expanding join on *n*. There are three ways in which the query can be transformed: i) OR-expansion ii) UAD transformation and iii) simulation of UAD using table duplication in a union-all view.

The OR-expansion transformation first converts the predicates in the where clause into a disjunctive normal form (DNF) and places each conjunct of the DNF in a union-all branch where tables and joins are duplicated. In QU1, the DNF predicate is "*(T1.n = T2.n and T1.u IS NULL) OR (T1.n = T2.n and T1.u = T2.u)*". Each branch in the expanded query joins T1 and T2 and applies one conjunct of the DNF predicate as shown in QU2 of Figure 15.

Query QU1 can be transformed into a more efficient form QU3 using UAD operator as shown in Figure 16. It is easier to understand QU3 by observing QU4 which simulates the UAD operator

of QU3 using a union-all branch. Consider three sample rows of table T2. The corresponding rows produced by view V1 in QU3 are shown in Table 5. In a row, if T2.u is NULL its P1 value is NULL, otherwise it is OP_MAP_NN(T2.u). Every row of T2 produces a fixed value OP_MAP_NN(NULL) (FF) in column P2. The rows are then duplicated using UAD and the result is joined with T1 using the join predicate "*OP_MAP_NN(T1.u) = V2.K*". When UAD is applied on V1, the output (view V2) consists of two sets of rows as shown in Table 6. One set is used to match with T1's rows that have non-NULL values in T1.u and the other set is used to match with T1's rows that contain NULL values in T1.u.

(1) The rows shown in purple contain the mapping result OP_MAP_NN(T2.u), when T2.u is not NULL (first branch of union-all in QU4). They allow matching on the join predicate "*T1.u is NULL OR T1.u = T2.u*" only when T1.u is not NULL.
(2) The rows shown in blue contain mapping of NULL for every row of T2 (second branch of union-all in QU4) and therefore allow matching of T1.u when it is NULL; and thereby simulating the disjunct, "*T1.u is NULL*".

Note that the values of K in the two parts/branches are mutually exclusive for every row of T2.

Table 5: Sample rows in QU3's view V1. (Columns *b, n, u* show table T2's data. T2 has 3 rows. *OP_MAP_NN(NULL) = FF*)

b	n	u	P1	P2
3	2	1	C10200	FF
15	12	NULL	NULL	FF
2	14	5	C10600	FF

Table 6: Two sets of rows produced by view V2 in QU3 after UAD is applied on V1. (Column *u* is shown only for reference.)

b	n	u	K
3	2	1	C10200
3	2	1	FF
15	12	NULL	FF
2	14	5	C10600
2	14	5	FF

Under the CBQT framework (Section 1.1), the above alternatives can be costed and the cheapest among them selected. Generally, the most optimal strategy is the one that uses the union-all duplicator operator, as it does not require multiple table accesses and joins. For instance, QU3 is generally more efficient that QU4, especially when they contain complex nested views inside them instead of simple tables.

The union-all duplicator transformation can be applied to any general filter predicate on T1 as shown in Figure 17. The underlying idea remains the same, as the union-all duplicator operator generates two mutually exclusive sets of T2 rows. Here, a CASE

```
QU5:
SELECT T1.a, T2.b
FROM  T1, T2
WHERE T1.n = T2.n AND
      (T1.d > 313 OR T1.u = T2.u);

QU6:
SELECT T1.a, T2.b
FROM  T1,
       (SELECT b, n, K
        FROM (SELECT n,
                NVL2(u, OP_MAP_NN(u), NULL) P1,
                OP_MAP_NN(NULL) P2
              FROM G_4K T2) V1
        UNION-ALL DUPLICATOR
          (K FOR KP IN (V1.P1, V1.P2))) V2
WHERE T1.n = V2.n AND
      (CASE WHEN T1.d > 313
        THEN OP_MAP_NN(NULL)
        ELSE NVL2(T1.u, OP_MAP_NN(T1.u), NULL)
       END) = V2.K;
```

Figure 17: QU5 with a general filter predicate transformed into QU6 using UAD.

statement in the WHERE clause produces FF (i.e., mapped NULL), when the given filter predicate, "*T1.d > 313*", evaluates to TRUE. In Oracle, the execution plan for this transformed query can build hash-join or sort-merge-join keys on complex join predicates between T1 and V2 (including the CASE expression), since these join predicates are considered well-formed.

To conclude, UAD transformation significantly benefits queries which contain expanding joins and disjunctive predicates, a pattern that is often found in customer workloads.

5 PERFORMANCE STUDY

In this section, we present data that show the effectiveness of the optimization techniques described in this paper. We first discuss performance improvement in various queries of TPC-DS benchmark. Later, we present representative data from customer workloads.

5.1 TPC-DS Workload

The transformation techniques described each apply to different queries in the TPC-DS benchmark. We present our observations based on experiments performed on data set generated with 1TB scale factor. We refer readers to the TPC-DS specification for full query texts.

5.1.1 Queries with Conditional Aggregates. Conditional expressions inside aggregates are found in TPC-DS queries Q2, Q40, Q43, Q50, Q59, Q62, Q66 and Q99. Many of these queries use CASE expressions inside aggregates to check a column from the *date_dim* table while joining it with other fact tables like *store_sales*. For example, Q59 contains an aggregation, *sum(case WHEN (d_day_name = 'Sunday') THEN ss_sales_price ELSE null)*. Grouping can be performed on the *store_sales* table by splitting the aggregate. Sometimes when group-by placement is chosen by the optimizer, it opens up other downstream optimizations on top of it, like pushing the grouping to the individual branches of union-all set (Q2), pushing

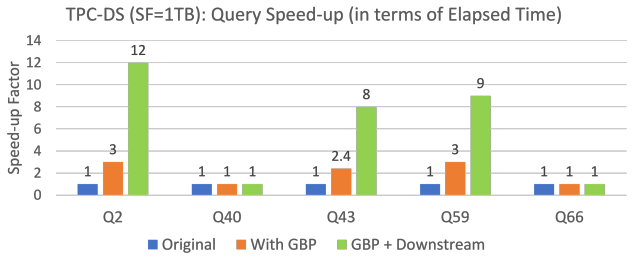

Figure 18: Effect of GBP enhancements on elapsed times of qualifying TPCDS queries.

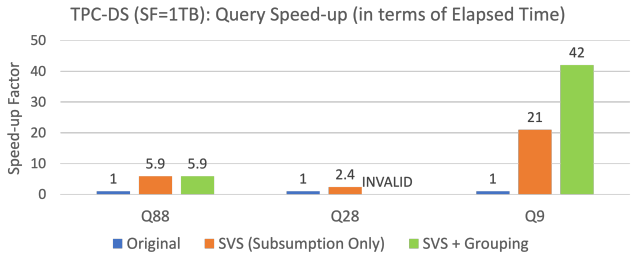

Figure 19: Effect of Subsumption and Grouping Transformation on elapsed times of qualifying TPCDS queries.

grouping and aggregations to table scans on disk (in Oracle Exadata) etc., leading to a large cumulative improvement in performance. Figure 18 shows the improvement in elapsed time as a speed-up ratio for each qualifying query when GBP is chosen in the final plan vs the plan without GBP.

Q2, Q43 and Q59 show the most improvement as they benefit from downstream optimizations in addition to grouping. In Q2 and Q59, splitting and grouping reduces the data of the "fact" tables by 1,000,000X and 4000X respectively which drives most of their improvement. Q40 and Q66 do not benefit from the additional states made possible by the enhancements. Q40 has relatively fewer conditional aggregates and multiple tables in addition to the ones in the aggregate and shows little improvement post-splitting. Although the grouping reduction that can be obtained by splitting is 16X, one of the joins in combination with old GBP states (prior to the enhancement) has a higher reduction of 30X. Similarly, although query Q66 has many conditional aggregates in it, the old GBP states are sufficient to benefit from the high reduction factors on tables *web_sales* and *catalog_sales* (65000X and 155,000X respectively). Hence, new GBP states are not chosen by the optimizer.

TPC-DS queries Q50, Q62 and Q99 contain case expressions that check columns from a single table and aggregate constant values. Hence, there is no case for splitting the expressions and generating additional GBP states. Therefore these queries are not affected by the enhancements.

Generally, execution of conditional expressions like CASE inside aggregates has more overhead and reducing the number of rows (through grouping) on which the conditional logic is evaluated improves performance significantly.

5.1.2 Subsumption of views and subqueries. TPC-DS queries Q9, Q28 and Q88 are candidates for subsumption transformation. Figure 19 shows the improvement in elapsed time of the queries in terms of the speed-up ratio before and after the transformation.

In Q88, there are eight views (each containing four tables) which satisfy the preconditions for subsumption as specified in Section 3. The views are subsumed into one view along with unifications and factorizations. After subsumption the query's performance (elapsed time) improves by a factor of ≈ 6X. The query doesn't qualify for pre-computation of aggregates via grouping as it contains more than one table. However, even if forced, it doesn't improve the performance.

```
SELECT f.c2, d1.c2,
       SUM(f.c3 * d1.c3), SUM(f.c4)
FROM  fact f, dim d1
WHERE f.c1 = d1.c1
  AND f.c4 < 120
  AND d1.c2 < 1000
GROUP BY f.c2, d1.c2;
```

Figure 20: A simplified representative customer query.

Query Q28 has six views that are subsumed by one view. Various unifications of filter predicates are applied but no factorization is possible. The transformed query's performance is ≈ 2.4X better than the original query. Grouping and aggregate pre-computation is not valid for this query because of the presence of COUNT DISTINCT.

In Q9, the equality filter predicate on a unique column in the outer query block ensures that the SELECT clause will be evaluated at most once. In this query, there are 15 subqueries referencing *store_sales*; they are subsumed by one query block and a group-by view that enables pre-computations of common aggregate functions is created in the final plan. The improvement is ≈ 21X with subsumption and a further ≈ 2X speed-up due to grouping.

5.2 Customer Workloads

In this section, we present our observations regarding the effectiveness of the described techniques in real-world workloads.

5.2.1 Queries with Arithmetic and Conditional Expressions in Aggregates. We encountered queries in Oracle Application Server, large enterprise ERP, finance, healthcare industry workloads, that perform joins between a large *fact* table and multiple *dimension* tables with aggregate expressions covering the *fact* and some *dimension* tables. The aggregates use arithmetic operations on columns from *fact* and *dimension* tables. In such queries, early grouping on the *fact* table alone (which is desirable) is infeasible without decomposing the aggregates. A simplified query example that fits the pattern is shown in Figure 20. Typically, around 10% of queries present such constructs.

While performance improvements vary across such diverse workloads, specifically, in two large customer applications in ERP and financial services domains using hundreds of terabytes of data, we observed a median speed-up factor of ≈5X across hundreds of eligible queries. The speed-up increases further when the base tables are loaded in-memory or stored on Oracle Exadata storage cells because

if grouping is introduced on single tables, the grouping operation is piggybacked on table scans, amplifying the benefit. Conditional expressions inside aggregates are also commonly present in queries of such workloads and we observed a median speed-up of ≈7X in them. This is because of the overhead of per-row evaluation of the condition, without grouping. For confidentiality reasons, we are not at liberty to publish the shape and nature of customer queries.

5.2.2 Disjunctive Queries with Union-all duplicator. In our experience, financial services workloads at large banks often contain queries with disjunctive join predicates (QU1 in Figure 15) where the non-disjunctive join is an expanding join. Moreover, the objects in the join are complex views that contain both tables and other nested views (nesting extending to many levels). A typical expanding join in such queries produces tens of millions of rows; before the disjunctive join predicate reduces their number to a few hundreds. Workloads contain hundreds of such queries. The median speed-up in elapsed time of the eligible queries, provided by union-all duplicator transformation is between 7X and 8X, with some queries improving by as much as 19X. The improvement comes from removal of the expanding join in the intermediate step. When joins return many rows on which the post-join filters need to be applied, the temporary results may even spill to disk that involves I/O. But due to UAD, when all the join predicates are simultaneously applied, the result is relatively small and is not materialized. This leads to savings in I/O in addition to improved join performance.

Parallel queries often suffer from skew in the presence of expanding joins because the large intermediate result tends to exhibit skew. Application of UAD operator reduces skew significantly by removing the intermediate step and hence parallel query plans tend to benefit even more due to UAD.

We now present the specifics of a class of queries that we encountered in an application at a large financial market exchange company. The representative class of queries have the following pattern. They contains tens of tables and views nested to more than five levels with many tables containing tens of terabytes of data. Among all joins within them, one join contains a disjunctive join predicate alongside an expanding join predicate. The expanding join produces over 200 million rows and the disjunctive predicate reduces them to 6 million. The queries run in parallel and use a degree of parallelism ranging from 64 to 128.
Before UAD optimization (all median values),

- The percentage of total query time taken by the entire join is 93.6%.
- The median start time of the join is 30 seconds after the start of query execution and end time is 440 seconds in a total query duration of 472 seconds.

After UAD transformation (all median values),

- The percentage of total query time taken by the entire join is 73.3%.
- The join starts 8 seconds after the start of query execution due to a different join order and takes about 21 seconds ending at 29 seconds.

The median speed-up for the above class of queries is ≈16.3X.

Optimization Overhead: The techniques described in this paper, especially the extensions to GBP, increase the time taken by the optimizer as they explore new states. In order to limit the overhead, linear exploration is used when there are many tables involved. In practice, we noticed the overhead to be less than 5% of the compilation time. The performance numbers presented in this section include this overhead.

5.3 Conclusion

The optimization techniques proposed in this paper can be applied to many queries in real-world customer workloads and benchmarks. We evaluated them across a wide variety of workloads and observed that query elapsed times improve by factors ranging from 3X to 40X, thus validating our ideas.

6 CITATIONS

The idea of grouping before joins was first introduced by [22]. Later [5] generalized it describing various grouping techniques that could be placed at multiple positions commuting them with joins in a join-tree. Our work on the other hand, focuses on techniques on decomposing complex aggregate expressions opening up additional options to placing group-by operations interspersed with joins on subsets of tables that appear in the aggregate expressions.

Partial pre-aggregation of data on columns that functionally determine the columns that are grouped and aggregated is discussed in [13]. Papers [18–20] describe processing of joins and aggregates in parallel database systems. [14] introduces the idea of merging join and aggregation operations into one operator and establishing equivalence to prove correctness. However, the papers do not address the decomposition problem.

Sharing of common sub-expression computation within and across queries was explored earlier in [17, 24]. The authors propose materializing the result of common sub-expressions and rewriting queries to re-use the result whenever feasible. While such techniques are more general, our subsumption transformation is light-weight and is targeted towards specific patterns - it doesn't involve materialization and retrieval as we create subsuming views in-situ by combining other views. These subsumed views can further participate in other transformations – e.g., they can be merged into their respective outer query blocks etc. They do not create a physical boundary induced by the materialize operator.

To our knowledge, the usage of a new operator as a query rewrite technique to perform disjunctive joins by replicating rows on-the-fly hasn't been discussed by prior body of work. It is possible to implement hash joins that maintain multiple hash tables or other similar structures to apply disjunctive join predicates on each of them and combine the result. Our proposed operator is more general as sort-merge joins can also be used.

REFERENCES

[1] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated generation of materialized views in Oracle. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3046–3058. https://doi.org/10.14778/3415478.3415533

[2] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-based query transformation in Oracle. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1026–1036.

[3] Rafi Ahmed, Rajkumar Sen, Meikel Poess, and Sunil Chakkappen. 2014. Of snowstorms and bushy trees. *Proc. VLDB Endow.* 7, 13 (aug 2014), 1452–1461. https://doi.org/10.14778/2733004.2733017

[4] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zait, and Chun-Chieh Lin. 2009. Enhanced subquery optimizations in Oracle. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1366–1377. https://doi.org/10.14778/1687553.1687563

[5] Surajit Chaudhuri and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 354–366.

[6] Surajit Chaudhuri and Kyuseok Shim. 1996. Optimizing Queries with Aggregate Views. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '96)*. Springer-Verlag, Berlin, Heidelberg, 167–182.

[7] Jens Claussen, Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. 2000. Optimization and Evaluation of Disjunctive Queries. *IEEE Trans. on Knowl. and Data Eng.* 12, 2 (mar 2000), 238–260. https://doi.org/10.1109/69.842265

[8] Sara Cohen, Werner Nutt, and Alexander Serebrenik. 1999. Rewriting aggregate queries using views. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pennsylvania, USA) *(PODS '99)*. Association for Computing Machinery, New York, NY, USA, 155–166. https://doi.org/10.1145/303976.303992

[9] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. 2015. Query optimization in Oracle 12c database in-memory. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1770–1781. https://doi.org/10.14778/2824032.2824074

[10] Sheldon Finkelstein. 1982. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data* (Orlando, Florida) *(SIGMOD '82)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/582353.582400

[11] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 358–369.

[12] Albert Kim and Samuel Madden. 2024. Optimizing Disjunctive Queries with Tagged Execution. *Proc. ACM Manag. Data* 2, 3, Article 158 (may 2024), 25 pages. https://doi.org/10.1145/3654961

[13] P.-A. Larson. 2002. Data reduction by partial preaggregation. In *Proceedings 18th International Conference on Data Engineering*. 706–715. https://doi.org/10.1109/ICDE.2002.994787

[14] Guido Moerkotte and Thomas Neumann. 2011. Accelerating queries with group-by and join by groupjoin. *Proc. VLDB Endow.* 4, 11 (aug 2011), 843–851. https://doi.org/10.14778/3402707.3402723

[15] Werner Nutt, Yehoshus Sagiv, and Sara Shurin. 1998. Deciding equivalences among aggregate queries. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 214–223. https://doi.org/10.1145/275487.275512

[16] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. *SIGMOD Rec.* 21, 2 (jun 1992), 39–48. https://doi.org/10.1145/141484.130294

[17] Yasin N. Silva, Paul-Ake Larson, and Jingren Zhou. 2012. Exploiting Common Subexpressions for Cloud Query Processing. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE '12)*. IEEE Computer Society, USA, 1337–1348. https://doi.org/10.1109/ICDE.2012.106

[18] D. Taniar, Y. Jiang, K.H. Liu, and C.H.C. Leung. 2000. Aggregate-join query processing in parallel database systems. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, Vol. 2. 824–829 vol.2. https://doi.org/10.1109/HPC.2000.843554

[19] D. Taniar and J.W. Rahayu. 2001. Parallel processing of "GroupBy-Before-Join" queries in cluster architecture. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. 178–185. https://doi.org/10.1109/CCGRID.2001.923191

[20] D. Taniar and W. Rahayu. 2006. Parallel "GroupBy-Before-Join" Query Processing for High Performance Parallel/Distributed Database Systems. In *20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, Vol. 1. 693–700. https://doi.org/10.1109/AINA.2006.256

[21] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Shen, and Sankar Subramanian. 2003. Spreadsheets in RDBMS for OLAP. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 52–63. https://doi.org/10.1145/872757.872767

[22] W.P. Yan and P.-A. Larson. 1994. Performing group-by before join /spl lsqb/query processing/spl rsqb/. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. 89–100. https://doi.org/10.1109/ICDE.1994.283001

[23] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 345–357.

[24] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/1247480.1247540