Eager Aggregation and Lazy Aggregation

Weipeng P. Yan Per-Åke Larson
Department of Computer Science,
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
{pwyan,palarson}@bluebox.uwaterloo.ca

Abstract

Efficient processing of aggregation queries is essential for decision support applications. This paper describes a class of query transformations, called *eager aggregation* and *lazy aggregation*, that allows a query optimizer to move group-by operations up and down the query tree. Eager aggregation partially pushes a group-by past a join. After a group-by is partially pushed down, we still need to perform the original group-by in the upper query block. Eager aggregation reduces the number of input rows to the join and thus may result in a better overall plan. The reverse transformation, lazy aggregation, pulls a group-by above a join and combines two group-by operations into one. This transformation is typically of interest when an aggregation query references a grouped view (a view containing a group-by). Experimental results show that the technique is very beneficial for queries in the TPC-D benchmark.

1 Introduction

Aggregation is widely used in decision support systems. All queries in the TPC-D[Raa95] benchmark contain aggregation. Efficient processing of aggregation queries is essential for performance in decision support applications and large scale applications.

Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995

We proposed a new query optimization technique, *group-by push down and group-by pull up,* which interchanges the order of group-by and joins[YL94, YL95]. Group-by push down is to push group-by past a join. Its major benefit is that the group-by may reduce the number of input rows of the join. Group-by pull up is to delay the processing of group-by until after a join. Its major benefit is that the join may reduce the number of input rows to the group-by, if the join is selective. Figure 1 shows the idea of commuting group-by and join. In Figure 1(a), we join Table T1(G1,J1,S1) and T2(G2,J2) on join columns J1 and J2 then group the result on grouping columns G1 and G2, followed by aggregation on S1. Figure 1(b) shows an alternative way where group-by is performed before join. Note that group-by and join commutation cannot always be done. The necessary and sufficient condition is provided in [YL94, YL95].

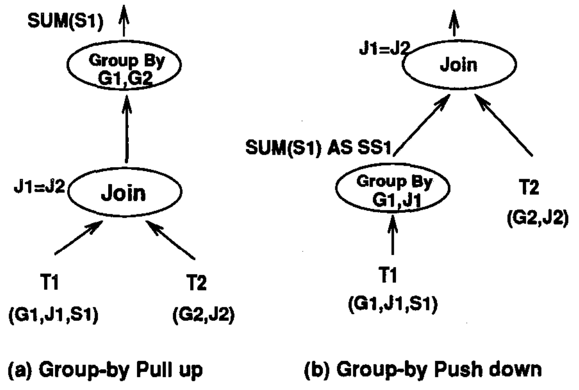

Figure 1: Group-by and Join Commutation

The technique to only partially push down a group-by past a join can be extended. For some queries containing joins and group-by, we can perform group-by on some of the tables, then the join, and finally another group-by. The first group-by, which we call *eager group-by*, reduces the number of input rows to the join and thus may result in a better plan. We call the groups generated by the early group-by *partial groups* because they will be merged by the second group-by. When the amount of data reduction does not justify the cost of eager group-by, we should probably delay group-by until after the join, which we term *lazy group-by.* Both directions of the transformation should be considered in query optimization. We call the technique of performing aggregation before join *eager aggregation*, and delaying aggregation until after join *lazy aggregation.*

Figure 2(a) and (b) show the basic idea of eager/lazy group-by. *Eager group-by* performs eager aggregation on *all tables* containing aggregation columns. Lazy group-by is its reverse transformation.

The following examples illustrate the basic idea of eager group-by and lazy group-by. The examples are based on a subset of the TPC-D database[Raa95]. The tables are defined in Appendix A.

Example 1 : *Find the total loss of revenue on orders handled by each clerk due to parts being returned by customers. Output clerk and loss of revenue.*

```
SELECT   O_CLERK,
         SUM(L_EXTENDEDPRICE * (1-L_DISCOUNT))
FROM     LINEITEM, ORDERS
WHERE    O_ORDERKEY = L_ORDERKEY
  AND    L_RETURNFLAG = 'R'
GROUP BY O_CLERK
```

Each order is handled by one clerk so we can first find the loss of revenue for each order. We then join the aggregated view with table ORDERS to find the total loss for each clerk.

```
SELECT   O_CLERK, SUM(REVENUE)
FROM     (SELECT   L_ORDERKEY, SUM(L_EXTENDEDPRICE
                   *(1-L_DISCOUNT)) AS REVENUE
          FROM     LINEITEM
          WHERE    L_RETURNFLAG = 'R'
          GROUP BY L_ORDERKEY) AS LOSS, ORDERS
WHERE    O_ORDERKEY = L_ORDERKEY
GROUP BY O_CLERK
```

The eager (inner) group-by reduces the number of input rows to the join. If the LINEITEM table is clustered on L_ORDERKEY, the eager group-by can be done at almost no additional cost. Experiment on DB2 V2 Beta3 confirms that eager group-by reduces the elapsed time by 16%. The following example shows that lazy group-by can be beneficial.

Example 2 : *Find the total loss of revenue on orders from May 1995 handled by each clerk due to parts being returned by customers. Output clerk and loss of revenue.*

```
SELECT   O_CLERK, SUM(REVENUE)
FROM     ORDERS, LOSS-BY-ORDER
WHERE    O_ORDERKEY = L_ORDERKEY
  AND    O_ORDERDATE BETWEEN "1995-05-01"
                     AND     "1995-05-31"
GROUP BY O_CLERK
```

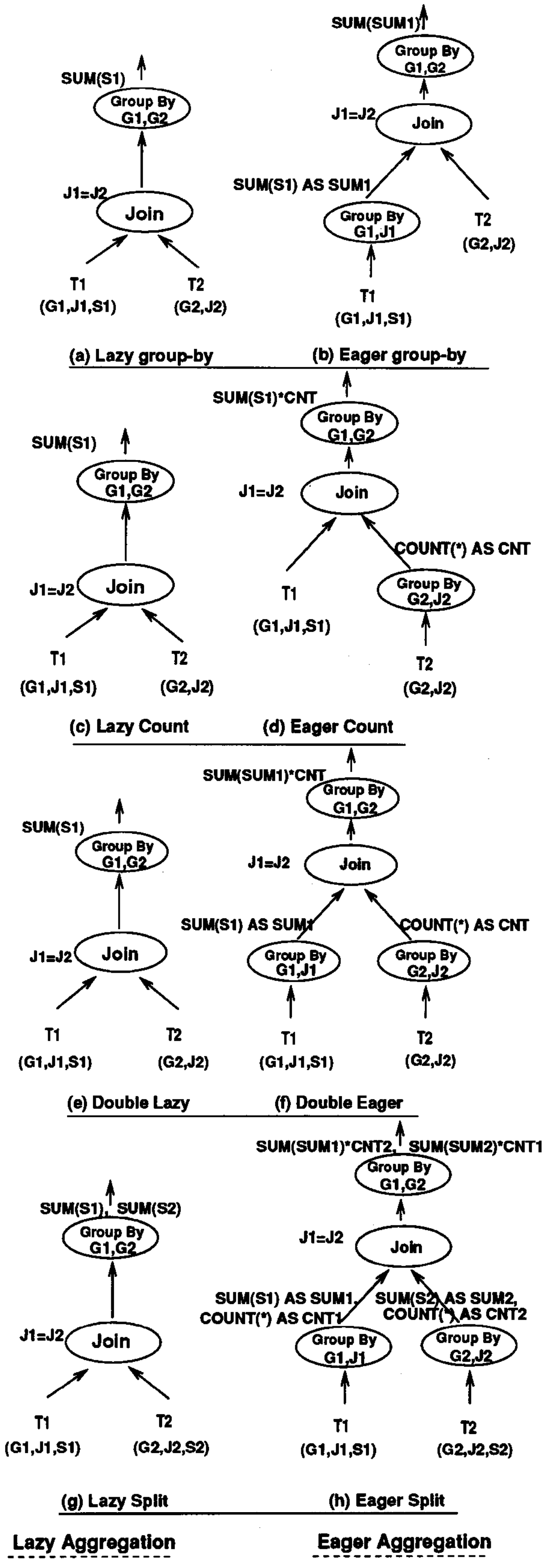

Figure 2: Eager and Lazy Aggregation

where LOSS_BY_ORDER is an aggregated view defined by

```
CREATE VIEW LOSS_BY_ORDER (L_ORDERKEY, REVENUE)
(SELECT   L_ORDERKEY,
          SUM(L_EXTENDEDPRICE * (1-L_DISCOUNT))
 FROM     LINEITEM
 WHERE    L_RETURNFLAG = 'R'
 GROUP BY L_ORDERKEY );
```

We can merge the view with the query and rewrite the query as

```
SELECT   O_CLERK,
         SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
FROM     LINEITEM, ORDERS
WHERE    O_ORDERKEY = L_ORDERKEY
  AND    L_RETURNFLAG = 'R'
  AND    O_ORDERDATE BETWEEN "1995-05-01"
                         AND "1995-05-31"
GROUP BY O_CLERK
```

The predicate on O_ORDERDATE is highly selective. In this case, we should delay the group-by until after the join. A nested loop join with LINEITEM as the inner and ORDERS as the outer looks like a very promising evaluation strategy. Experiment on DB2 V2 Beta3 confirms that lazy group-by reduces the elapsed time by 60%.

These examples show that both directions (eager group-by and lazy group-by) should be considered in query optimization. There may be several ways of performing eager group-by when there are more than two tables in the FROM clause[Yan95].

Figure 2 and 3 show the eager/lazy transformations introduced in this paper. *Eager count* transformation performs eager aggregation on tables not containing aggregation columns, as shown in Figure 2 (d). It first counts the number of rows in each group in the early aggregation, then performs the join, and finally aggregates the original aggregation columns. *Lazy count* transformation is its reverse transformation.

Double eager performs eager count on tables not containing aggregation columns and eager group-by on the remaining tables which may or may not contain aggregation columns, as shown in Figure 2 (f). The reverse transformation is *double lazy.*

Eager groupby-count, as shown in Figure 3, performs eager aggregation on a *subset* of tables containing the aggregation columns. Its reverse transformation is called *lazy groupby-count.*

Eager split, as shown in Figure 2 (h), performs eager groupby-count on both input streams before the join, when both input streams are involved in aggregation. Its reverse transformation is called *lazy split.*

Our experiments show that we can apply group-by push down/pull up and eager/lazy aggregation to twelve of the seventeen queries in the TPC-D benchmark. This significantly reduces the elapsed time for six queries. For example, it reduces the elapsed time of Query 5 by a factor of ten.

1.1 Organization of This Paper

The rest of the paper is organized as follows. Section 2 reviews aggregation functions in SQL2 and introduces the concepts of decomposable aggregation functions, and class C and class D aggregation functions. Section 3 defines the class of queries that we consider and introduces notations. Section 4 presents the formalism that our results are based on. Section 5 introduces and proves our main theorem. Sections 6, 7, 8 and 9 introduce corollaries for eager/lazy group-by, eager/lazy count, double eager/lazy and eager/lazy split transformations. Section 10 proposes algorithms for finding all possible eager/lazy transformations for a query and discusses the way to integrate eager/lazy aggregation and group-by push down/pull up into existing optimizers. In order to simplify the proofs we have not considered HAVING in the theorem and corollaries. Section 11 considers the case when the HAVING clause is present. Section 12 shows that eager/lazy aggregation and group-by push down/pull up is very beneficial for TPC-D official queries. Section 13 discusses related work. Section 14 concludes the paper.

2 Aggregation Functions

In SQL2, a value expression may include aggregation functions. There are five aggregation functions: SUM, AVG, MIN, MAX and COUNT. Consider the query

```
SELECT 2*SUM(T1.C1)/COUNT(DISTINCT T2.C2)*
       MIN(T1.C3*T2.C3)
FROM T1,T2
```

We can rewrite this query as

```
SELECT 2*NC1/NC2*NC3
FROM (SELECT SUM(C1) AS NC1,
             COUNT(DISTINCT T2.C2) AS NC2,
             MIN(T1.C3*T2.C3) AS NC3
      FROM T1,T2 ) TMP_VIEW;
```

Any query block that has an arbitrary value expression containing more than one aggregation functions, can always be rewritten so that the new query block is a SELECT on top of a view that contains value expression having at most one aggregation function. Therefore, without loss of generality, we assume that our query contains no value expression that has more than one aggregation functions.

2.1 Decomposable Aggregation Functions

All sets in this paper are multi sets. Let $\cup_a$ denote set union preserving duplicates, and $\cup_d$ denote set union eliminating duplicates. These operations exist in SQL2 as UNION ALL and UNION, respectively.

Definition 1 : (Decomposable Aggregation Function) *An aggregation function F is* decomposable *if there exist aggregation functions F1 and F2 such that* $F(S_1 \cup_a S_2) = F2(F1(S_1), F1(S2))$, *where* S_1 *and* S_2 *are two sets of values. We call* S_1 *and* S_2 *partial groups.*

SUM(C) is decomposable since $SUM(S1 \cup_a S2) = SUM(SUM(S1), SUM(S2))$;
COUNT(C) is decomposable since $COUNT(S1 \cup_a S2) = SUM(COUNT(S1), COUNT(S2))$;
and MIN(C) is decomposable since $MIN(S1 \cup_a S2) = MIN(MIN(S1), MIN(S2))$; and AVG(C) can be handled as SUM(C) and COUNT(NOT NULL C) and thus is decomposable[1].

For aggregation functions like COUNT(DISTINCT C1), it is not trivial to determine whether it is decomposable. There may be two rows with the same C1 value in S1 and S2. These two rows would then contribute 2 instead of 1 in the final count. However, if we know in advance that column C1 cannot contain duplicate values, then COUNT(DISTINCT C1) is decomposable. Note that, even though C1 has duplicate values, there may be other conditions which ensure that rows with the same C1 value belong to the same partial groups(e.g., C1 is a grouping column). Therefore, an aggregation function may or may not be *decomposable.* Aggregation functions MIN and MAX are always decomposable; SUM and COUNT are decomposable when they contain no DISTINCT. The issue of determining whether an aggregation function is decomposable will not be discussed further. From now on we will assume that we have the knowledge about whether an aggregation function is decomposable.

2.2 Class C and Class D Aggregation Functions

Example 3 : *Find the total number of urgent or high priority lineitems handled by each clerk.*

```
SELECT  O_CLERK,
        SUM(CASE WHEN O_ORDERPRIORITY='1-URGENT'
                   OR O_ORDERPRIORITY='2-HIGH'
                 THEN 1 ELSE 0 END)
FROM     LINEITEM, ORDERS
WHERE    O_ORDERKEY = L_ORDERKEY
GROUP BY O_CLERK
```

It is equivalent to the following query.

```
SELECT  O_CLERK,
        SUM(CASE WHEN O_ORDERPRIORITY='1-URGENT'
                   OR O_ORDERPRIORITY='2-HIGH'
                 THEN 1 ELSE 0 END) * CNT
FROM    (SELECT L_ORDERKEY, COUNT(*) AS CNT
         FROM LINEITEM
         GROUP BY L_ORDERKEY) AS COUNT_BY_ORDER,
         ORDERS
WHERE    O_ORDERKEY = L_ORDERKEY
GROUP BY O_CLERK
```

COUNT_BY_ORDER counts the number of lineitems for each orders, then joins with table ORDERS to find the count required. We call this transformation *eager count*, and its corresponding reverse transformation *lazy count.* Note that, this time, we are performing eager aggregation on a table which contains no aggregation columns. *Eager count* performs eager aggregation on tables not containing any aggregation columns. Experiment on DB2 V2 Beta3 shows that eager count for this query reduces the elapsed time by 40%.

When performing eager count and the original aggregation function is either SUM or COUNT, we need to count the number of rows in each group produced by the inner group-by and multiply the count with the result from the later group by. We call aggregation functions satisfying this property *class C* aggregation functions(C stands for COUNT), and the count obtained from the inner group-by *duplication factor.* If the original aggregation function is SUM(DISTINCT), COUNT(DISTINCT), MIN, MAX, or AVG, we can discard the count in the subquery block. In other words, we can use a DISTINCT in the subquery block. We call aggregation functions satisfying this property *class D* aggregation functions(D stands for DISTINCT). And we call this transformation *eager distinct*, and its corresponding reverse transformation *lazy distinct.* Therefore, combining this with whether the function is decomposable or not, we can have four types of aggregation functions. *Class D* aggregation functions are insensitive to duplication factors.

3 Class of Queries Considered

Any column occurring as an operand of an aggregation function (COUNT, MIN, MAX, SUM, AVG) in the SELECT clause is called an *aggregation column.* Any column occurring in the SELECT clause which is not an aggregation column is called a *selection column.* Aggregation columns may belong to more than one tables. We partition the tables in the FROM clause into two groups: those tables that contain aggregation columns and those that may or may not contain any such columns. Technically, each group can be treated as a single table consisting of the Cartesian product of the member tables. Therefore, without loss of generality, we can assume that the FROM clause contains only two tables, R_d and R_u. Let R_d denote the table containing aggregation columns and R_u the table that may or may not contain any such columns.

The search conditions in the WHERE clause can be

[1] SQL2 does not support COUNT(NOT NULL C1) operation, but it is fairly easy to implement in any existing systems.

expressed as $C_d \wedge C_0 \wedge C_u$, where C_d, C_0, and C_u are in conjunctive normal form, C_d only involves columns in R_d, C_u only involves columns in R_u, and each disjunctive component in C_0 involves columns from both R_d and R_u. Note that subqueries are allowed.

The grouping columns mentioned in the GROUP BY clause may contain columns from R_d and R_u, denoted by GA_d and GA_u, respectively. According to SQL2[ISO92], the selection columns in the SELECT clause must be a subset of the grouping columns. We denote the selection columns as SGA_d and SGA_u, subsets of GA_d and GA_u, respectively. For the time being, we assume that the query does not contain a HAVING clause(relaxed in Section 11). The columns of R_d participating in the join and grouping is denoted by GA_d^+, and the columns of R_u participating in the join and grouping is denoted by GA_u^+.

In summary, we consider queries of the following form:

```
SELECT [ALL/DISTINCT]     SGA_d, SGA_u, F(AA)
FROM                      R_d, R_u
WHERE                     C_d ∧ C_0 ∧ C_u
GROUP BY                  GA_d, GA_u
```

where

GA_d:	grouping columns of table R_d;
GA_u:	grouping columns of table R_u; GA_d and GA_u cannot both be empty.
SGA_d:	selection columns, must be a subset of grouping columns GA_d;
SGA_u:	selection columns, must be a subset of grouping columns GA_u;
AA:	aggregation columns of table R_d and possible table R_u. When considering eager/lazy group-by, eager/lazy count and double eager/lazy, AA belong to R_d. When considering eager/lazy groupby-count and eager/lazy split, AA belong to R_d and R_u and is denoted by the union of aggregation columns AA_u and AA_d,, where AA_u and AA_d belong to R_d and R_u respectively.
C_d:	conjunctive predicates on columns of table R_d;
C_u:	conjunctive predicates on columns of table R_u;
C_0:	conjunctive predicates involving columns of both tables R_d and R_u, e.g., join predicates;
$\alpha(C_0)$:	columns involved in C_0;
F:	array of aggregation functions and/or arithmetic aggregation expressions applied on AA (may be empty). When considering eager/lazy groupby-count and eager/lazy split, F is denoted by the union of aggregation functions F_d and F_u,, where F_d and F_d are applied on AA_d and AA_u respectively.
$F(AA)$:	application of aggregation functions and/or arithmetic aggregation expressions F on aggregation columns AA;
GA_d^+:	$\equiv GA_d \cup \alpha(C_0) - R_u$, i.e., the columns of R_d participating in the join and grouping;
GA_u^+:	$\equiv GA_d \cup \alpha(C_0) - R_d$, i.e., the columns of R_u participating in the join and grouping
FAA:	resulting columns of the application of function array F on AA in the first group-by when eager group-by is performed on the above query.

4 Formalization

In this section we define the formal "machinery" we need for the theorems and proofs to follow.

SQL2[ISO92] represents missing information by a special value NULL. It adopts a three-valued logic in evaluating a conditional expression. We define functional dependencies using strict SQL2 semantics taking into account the effect of NULLs in SQL2. When NULLs do not occur in the the columns involved in a functional dependency, our definition of functional dependency is the same as the traditional functional dependency. The detailed definitions are included in [YL94]. Due to space limitation, they are not included here. Let A and B be two sets of columns, A *functionally determines* B is denoted by $A \longrightarrow B$.

4.1 An Algebra for Representing SQL Queries

Specifying operations using standard SQL is tedious. As a shorthand notation, we define an algebra whose basic operations are defined by simple SQL statements. Because all operations are defined in terms of SQL, there is no need to prove the semantic equivalence between the algebra and the SQL statements. Note that transformation rules for "standard" relational algebra do not necessarily apply to this new algebra. The operations are defined as follows.

- $\mathcal{G}$[GA] R: Group table R on grouping columns $GA = \{GA_1, GA_2, ..., GA_n\}$. This operation is defined by the query [2] SELECT * FROM R ORDER BY GA. The result of this operation is a *grouped table.*

- $R_1 \times R_2$: The Cartesian product of table R_1 and R_2.

- $\sigma[C]R$: Select all rows of table R that satisfy condition C. Duplicate rows are not eliminated. This operation is defined by the query SELECT * FROM R WHERE C.

- $\pi_d[B]R$, where $d = A$ or D: Project table R on columns B, without eliminating duplicates when

[2] Certainly, this query does more than GROUP BY by ordering the resulting groups. However, this appears to be the only valid SQL query that can represent this operation. It is appropriate for our purpose as long as we keep the difference in mind.

$d = A$ and with duplicate elimination when $d = D$. This operation is defined by the query SELECT [ALL /DISTINCT] B FROM R.

- $F[AA]R$: $F[AA] = (f_1(AA), f_2(AA), ..., f_n(AA))$, where $AA = \{A_1, A_2, ..., A_n\}$, and $F = \{f_1, f_2, ..., f_n\}$, AA are aggregation columns of grouped table R and F are arithmetic aggregation expressions operating on AA. We must emphasis the requirement that table R is grouped by some grouping columns C. All rows of table R must agree on the values of all columns except AA columns. Each f_i, where $i = 1, 2, ..., n$, is an arithmetic expression(which can simply be an aggregation function) applied to some columns in AA of each group of R and yields one value. An example of $f_i(AA)$ is $\texttt{COUNT}(A_1) + \texttt{SUM}(A_2 + A_3)$. Duplicates in the overall result are not eliminated. This operation is defined by the query SELECT GA,A, F(AA) FROM R GROUP BY GA, where GA is the grouping columns of R, and A is a set of none grouping columns that are functionally determined by GA and may be empty. Note that this is not a syntactically valid SQL2 statement since the columns A in the SELECT clause are not mentioned in the GROUP BY clause. However, since $GA \longrightarrow A$, from a query processing point of view, this is semantically sound.

Therefore, the class of query we consider can be expressed as

$$\pi_d[SGA_d, SGA_u, FAA]F[AA]\pi_A[GA_d, GA_u, AA]$$
$$\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$$

where $d = A$ or $d = D$, and FAA are the aggregation values after applying $F[AA]$ on each group. The last projection simply projects the rows on the columns wanted, and may eliminate duplicates. If all the columns wanted are the same as all existing columns, and the projection does not eliminate duplicates, then we usually omit the last projection in the expression.

All sets in this paper are multisets which may contain duplicates. r_d, r_u denote instances of table R_d and R_u; $T[S]$ is used as a shorthand for $\pi_A[S]T$, where S is a set of columns and T is a grouped or ungrouped table, or a row.

5 Main Theorem

When performing eager count, we need to consider two cases:

1. F contains only class D aggregation functions. We can simply add a DISTINCT to the SELECT list of the subquery block and no modification to the original aggregation functions is needed.

2. F contains both class C and class D aggregation functions. In this case, we need to use a COUNT aggregation function in the SELECT list of the subquery block. The aggregation value of a class C aggregation function f is the count multiplied by the value resulting from applying f. Therefore, we need to change F into F_a, in which every class C aggregation function f of F is replaced by $f * count$. For example, if $F(C1, C2, C3)$ is (SUM(C1),COUNT(C2),MIN(C3)), then $F_a(C1, C2, C3, count)$ is (SUM(C1),MAX(C2),MIN(C3))$\circ(count, 1, 1)$ = (SUM(C1)$*count$,MAX(C2),MIN(C3)). The operator $\circ$ is vector product. We call F_a the *duplicated aggregation functions* of F. As a shorthand notation, we use $F(C1, C2, ..., C_n) * count$ to represent F_a, while keeping in mind that we only need to multiply class C aggregation function by the count. Note that we need an additional argument to F_a.

Note that, it is not necessary that the functions in F be decomposable.

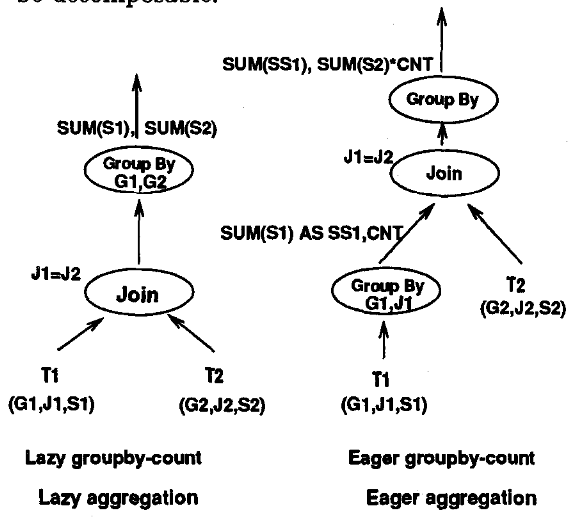

Figure 3: The Main Theorem

Consider the query to the left of Figure 3. It aggregates columns from both input streams. In the query on the right, we can first perform aggregation on one of the input stream. We need to not only find the sum of partial groups, but also keep track of the number of rows in each partial group for the aggregation on the table(T2) that are aggregated only after the join. This is the basic idea of eager groupby-count.

In the following theorem, let (1) NGA_d denote a set of columns in R_d; (2) CNT the column produced by COUNT(*) after grouping $\sigma[C_d]R_d$ on NGA_d; (3) FAA_d the rest of the columns produced by F_d in the first group-by of table $\sigma[C_d]r_d$ on NGA_d; and (4) F_{ua} the duplicated aggregation function of F_u. Also assume that (1) $AA = AA_d \cup_d AA_u$ where AA_d contains only columns in R_d, and AA_u contains only columns in R_u; (2) $F = F_d \cup_d F_u$ where F_d applies to AA_d and F_u applies to AA_u.

Theorem 1 (Eager/Lazy Groupby-Count(Main Theorem)): *The expressions*

$$\begin{aligned} E_1: \ & F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u] \\ & \mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u) \end{aligned}$$

and

$$\begin{aligned} E_2: \ & \pi_d[GA_d, GA_u, FAA] \\ & (F_{ua}[AA_u, CNT], F_{d2}[FAA_d]) \\ & \pi_A[GA_d, GA_u, AA_u, FAA_d, CNT] \\ & \mathcal{G}[GA_d, GA_u]\sigma[C_0, C_u](((F_{d1}[AA_d], COUNT[]) \\ & \pi_A[NGA_d, GA_d^+, AA_d]\mathcal{G}[NGA_d]\sigma[C_d]R_d) \times R_u) \end{aligned}$$

are equivalent if (1) aggregation functions F_d contain only decomposable aggregation functions and can be decomposed into F_{d1} and F_{d2}; (2) F_u contain class C or D aggregation functions and (3) $NGA_d \longrightarrow GA_d^+$ hold in $\sigma[C_d]R_d$.

The main theorem is illustrated in Figure 3. The aggregation columns are split into two sets, which belong to R_d and R_u tables respectively. For the transformation from E_1 to E_2 (eager aggregation), we push down the R_d tables and perform eager aggregation on AA_d and obtain the count before the join. After the join, we then perform aggregation on FAA_d and AA_u. Therefore, we basically split the aggregation into two parts, one is pre-evaluated before the join and one is evaluated after the join. We call the transformation from E_1 to E_2 *eager groupby-count* and its reverse transformation *lazy groupby-count*.

The requirement $NGA_d \longrightarrow GA_d^+$ is not a necessary conditions. If $NGA_d \not\longrightarrow GA_d$ in some instance of $\sigma[C_d]R_d$, then the first group-by of E_2 may group rows together when they do not belong to the same group in E_1. However, incorrectly assigned rows may be eliminated by the join and we may still get the correct result. If NGA_d does not functionally determine the join columns of table R_d, the join in E_2 is undefined since a group may contain different values on the join columns. To obtain necessary and sufficient condition, we need to extend the meaning of $F[AA]$, which is beyond the scope of this paper.

Proof:

Consider a group G_d in $\mathcal{G}[NGA_d]\sigma[C_d]r_d$ for some instance r_d of R_d. Since $NGA_d \longrightarrow GA_d^+$, all rows in G_d have the same GA_d value and have the same value for the join columns of R_d. Therefore, if one row of G_d qualifies in the join of $\sigma[C_d \wedge C_0 \wedge C_u]$ $(r_d \times r_u)$, all rows of G_d qualify. If one row of G_d joins with a set of rows S_u from $\sigma[C_u]r_u$, all rows of G_d join with S_u. Note that the above statements hold for all joins, not just equijoins.

Since S_u depends on G_d, we denote the set of rows joining with G_d as $S_u(G_d)$. The set resulting from the join of G_d and S_u is $G_d \times S_u(G_d)$, i.e., a Cartesian product. $(F_{d1}[AA_d], COUNT[])G_d$ denotes the row resulting from applying F_{d1} and COUNT on AA_d of the group G_d.

Let G_{d1}, G_{d2} be two (partial groups) produced by $\mathcal{G}[NGA_d]\sigma[C_d]r_d$. We have two cases to consider.
Case 1: $G_{d1}[GA_d] = G_{d2}[GA_d]$ and $S_u(G_{d1})[GA_u] = S_u(G_{d2})[GA_u]$. In E_2, after the join, all rows in

$$\begin{aligned} & ((F_{d1}[AA_d], COUNT[]) \\ & (\pi[NGA_d, GA_d^+, AA_d]G_{d1}) \times S_u(G_{d1}) \end{aligned}$$

and

$$\begin{aligned} & ((F_{d1}[AA_d], COUNT[]) \\ & \pi[NGA_d, GA_d^+, AA_d]G_{d2}) \times S_u(G_{d2}) \end{aligned}$$

are merged into the same group by the second group-by(after the join).

In E_1, each row in G_{d1} and G_{d2} joins with each row in $S_u(G_{d1})$ and $S_u(G_{d2})$, respectively. Therefore, all rows in $G_{d1} \times S_u(G_{d1})$ and $G_{d2} \times S_u(G_{d2})$ are merged into the same group by the group-by. Since every aggregation function in F_d can be decomposed as F_{d1} and F_{d2}, the aggregation values in the row produced by

$$\begin{aligned} & F_d[AA_d]\pi_A[GA_d, GA_u, AA_d] \\ & ((G_{d1} \times S_u(G_{d1}))\cup_a(G_{d2} \times S_u(G_{d2}))) \end{aligned}$$

in E_1 are equal to the aggregation values produced by

$$\begin{aligned} & F_{d2}[FAA_d]\pi_A[GA_d, GA_u, FAA_d] \\ & (((F_{d1}[AA_d]\pi_A[NGA_d, GA_d^+, AA_d]G_{d1}) \times S_u(G_{d1})) \\ & \cup_a((F_{d1}[AA_d]\pi_A[NGA_d, GA_d^+, AA_d]G_{d2}) \times S_u(G_{d2}))) \end{aligned}$$

in E_2.

Since every aggregation function in F_u is either class C or D, the aggregation values in the row produced by

$$\begin{aligned} & F_u[AA_u]\pi_A[GA_d, GA_u, AA_u]((G_{d1} \times S_u(G_{d1})) \\ & \cup_a G_{d2}(\times S_u(G_{d2}))) \end{aligned}$$

in E_1 are equal to the aggregation values produced by

$$\begin{aligned} & F_{ua}[AA_u, CNT]\pi_A[GA_d, GA_u, AA_u, CNT] \\ & (((COUNT[]\pi_A[NGA_d, GA_d^+]G_{d1}) \times S_u(G_{d1}))\cup_a \\ & ((COUNT[]\pi_A[NGA_d, GA_d^+]G_{d2}) \times S_u(G_{d2}))) \end{aligned}$$

in E_2.
Case 2: $G_{d1}[GA_d] \neq G_{d2}[GA_d]$ or $S_u(G_{d1})[GA_u] \neq S_u(G_{d2})[GA_u]$. In E_2, the rows in

$$\begin{aligned} & ((F_{d1}[AA_d], COUNT[]) \\ & \pi[NGA_d, GA_d^+, AA_d]G_{d1}) \times S_u(G_{d1}) \end{aligned}$$

and

$$((F_{d1}[AA_d], COUNT[])$$
$$\pi[NGA_d, GA_d^+, AA_d]G_{d2}) \times S_u(G_{d2})$$

are not merged into the same group by the second group-by(after the join). In E_1, each row in G_{d1} and G_{d2} joins with each row in $S_u(G_{d1})$ and $S_u(G_{d2})$, respectively. However, the rows in $G_{d1} \times S_u(G_{d1})$ and $G_{d2} \times S_u(G_{d2})$ are not merged into the same group by the group-by. Since F is decomposable, the aggregation values in

$$F_d[AA_d]\pi_A[GA_d, GA_u, AA_d]$$
$$(G_{d1} \times S_u(G_{d1})$$

in E_1 are equal to the aggregation values in

$$F_{d2}[FAA_d]\pi_A[GA_d, GA_u, FAA_d]$$
$$((F_{d1}[AA_d]\pi_A[NGA_d, GA_d^+, AA_d]G_{d1}) \times S_u(G_{d1}))$$

in E_2.

Also, the aggregation values in the row produced by

$$F_u[AA_u, CNT]\pi_A[GA_d, GA_u, AA_u]$$
$$(G_{d1} \times S_u(G_{d1}))$$

in E_1 are equal to the aggregation values produced by

$$F_{ua}[AA_u, CNT]\pi_A[GA_d, GA_u, AA_u, CNT]$$
$$((COUNT[]\pi_A[NGA_d, GA_d^+]G_{d1}) \times S_u(G_{d1}))$$

in E_2. □

The Main Theorem assumes that the final selection columns are the same as the grouping columns(GA_d, GA_u) and the final projection must be an ALL projection. We can actually relaxes these two restrictions, i.e., the final selection columns may be a subset(SGA_d, SGA_u) of the grouping columns(GA_d, GA_u), and the final projection may be a DISTINCT projection. This is also true for all other corollaries in this paper. For a formal description of the transformation and proof, please refer to [Yan95].

6 Eager Group-by and Lazy Group-by

In the Main Theorem, if we let GA_d contain all the aggregation columns, that is, all aggregation columns belong to R_d tables, then we obtain the following corollary.

In the following corollary, let NGA_d denote a set of columns in table R_d. and FAA_d the columns produced by applying $F[AA]$ after grouping table R_d on NGA_d.

Corollary 1 (Eager Group-by and Lazy Group-by): *The expressions*

$$E_1: \quad F[AA]\pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u]$$
$$\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$$

and

$$E_2: \quad F_2[FAA_d]\pi_A[GA_d, GA_u, FAA_d]\mathcal{G}[GA_d, GA_u]$$
$$\pi_A[GA_d, GA_u, FAA_d]\sigma[C_0 \wedge C_u]$$
$$((F_1[AA]\pi_A[NGA_d, GA_d^+, AA]$$
$$\mathcal{G}[NGA_d]\sigma[C_d]R_d) \times R_u)$$

are equivalent if $NGA_d \longrightarrow GA_d^+$ *holds in* $\sigma[C_d]R_d$ *and all aggregation functions in* $F[AA]$ *are decomposable and can be decomposed into* F_1 *and* F_2.

Eager group-by transformation introduces a new group-by, and lazy group-by transformation eliminates a group-by.

The proof of the corollary is straightforward. Since AA_u is empty, $F_{ua}[AA_u, CNT]$ is empty. Deleting all terms relating to AA_u in E_2 of the Main Theorem gives E_2 of the corollary.

7 Eager/Lazy Count and Eager/Lazy Distinct

In the Main Theorem, if we let GA_u contain all the aggregation columns, that is, all aggregation columns belong to R_u tables, then we obtain the following corollary. In the following corollary, NGA_d denotes a set of grouping columns belonging to R_d, and CNT the column produced by COUNT(*) after grouping $\sigma[C_d]R_d$ on NGA_d.

Corollary 2 (Eager Count/Lazy Count): *The expressions*

$$E_1: \quad F[AA]\pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u]$$
$$\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$$

and

$$E_2: \quad F_a[AA, CNT]\pi_A[GA_d, GA_u, AA, CNT]$$
$$\mathcal{G}[GA_d, GA_u]\pi_A[GA_d, GA_u, AA, CNT]$$
$$\sigma[C_0, C_u]((COUNT[]\pi_A[NGA_d, GA_d+]$$
$$\mathcal{G}[NGA_d]\sigma[C_d]R_d) \times R_u)$$

are equivalent if F are class C or class D aggregation functions and $NGA_d \longrightarrow GA_d^+$ *hold in* $\sigma[C_d]R_d$.

In E_2 above, COUNT[] after the inner group-by in E_2 means that we add a COUNT(*) to the select list of the subquery block.

The proof of the corollary is straightforward. Since AA_d is empty, F_d, F_{d1} and F_{d2} are all empty. Removing all terms relating to AA_d in E_2 of the Main Theorem gives E_2 of the corollary.

We call the transformation from E_1 to E_2 *eager count* and from E_2 to E_1 *lazy count.*

Clearly, when F in the theorem contains only class D aggregation functions, we can simply use a DISTINCT in the subquery block. We then call the transformation from E_1 to E_2 *eager distinct* and from E_2 to E_1 *lazy distinct*. Note that in this case, F_a is the same as F.

8 Double Eager and Double Lazy

Now we are ready to tackle the double eager and double lazy transformations. Consider the query in Figure 2(e). It aggregates the columns belonging to one input stream (T1). In the query in Figure 2(f), we perform eager group-by on the stream (T1) containing aggregation columns and eager count on the stream (T2) not containing any aggregation columns. We call the transformation double eager. Double eager can be understood as an eager group-by followed by an eager count transformation. The reverse transformation is called double lazy.

In the following corollary, NGA_u denotes a set of columns in R_u, NGA_d a set of grouping columns belonging to R_d tables, FAA the columns produced by F_1 in the first group-by of table $\sigma[C_d]R_d$ on NGA_d, and CNT the column produced by COUNT(*) after grouping $\sigma[C_u]R_u$ on NGA_u. Also assume that AA belongs to R_d.

Corollary 3 (Double Eager/Double Lazy): *the expressions*

$$E_1: \; F[AA]\pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u] \\ \sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$$

and

$$E_2: \; F_a[F_2[FAA], CNT]\pi_A[GA_d, GA_u, FAA, CNT] \\ \mathcal{G}[GA_d, GA_u]\sigma[C_0]((COUNT[] \\ \pi_A[NGA_u, GA_u+]\mathcal{G}[NGA_u]\sigma[C_u]R_u) \\ \times(F_1[AA]\pi_A[NGA_d, GA_d+, AA]\mathcal{G}[NGA_d] \\ \sigma[C_d]R_d))$$

are equivalent if (1) $NGA_u \longrightarrow GA_u$ holds in $\sigma[C_u]R_u$, (2) $NGA_d \longrightarrow GA_d$ holds in $\sigma[C_d]R_d$, (3) all aggregation functions in F are decomposable and can be decomposed as F_1 and F_2, (4) all aggregation functions in F are class C or D and its duplicated aggregation function is F_a.

The proof of this corollary is straightforward. It can be done by first performing an eager/lazy group-by and then an eager/lazy count.

Again, when F in the corollary contains only class D aggregation functions, we can simply use a DISTINCT in the subquery block of R_u. Note that in this case, F_a is the same as F. The following corollary shows when the group-by at the top query block may be eliminated.

Corollary 4
(Double Group-by Push-Down/Double Group-by Pull-Up): *Assume that the conditions in Corollary 3 holds. If, in addition, (1) $GA_d^+ \longrightarrow NGA_d$ holds in $\sigma[C_d]R_d$, (2) $GA_u^+ \longrightarrow NGA_u$ holds in $\sigma[C_u]R_u$, and (3) (GA_u, GA_d) functionally determines the join columns in $\sigma[C_d \wedge C_0 \wedge C_u]$ $(R_d \times R_u)$, then the expressions*

$$E_1: \; F[AA]\pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u] \\ \sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$$

and

$$E_2: \; \pi_A[GA_d, GA_u, FAA * CNT]\mathcal{G}[GA_d, GA_u] \\ \sigma[C_0]((COUNT[]\pi_A[NGA_u, GA_u+]\mathcal{G}[NGA_u] \\ \sigma[C_u]R_u) \times (F[AA]\pi_A[NGA_d, GA_d+, AA] \\ \mathcal{G}[NGA_d]\sigma[C_d]R_d))$$

are equivalent.

This Corollary eliminates the group-by at the top query block. This can be viewed as a more general case of group-by push down, which pushs down group-by into two lower query blocks. We call the transformation *double group-by push down*. Its reverse transformation, which pull up group-by's from two lower query blocks, is called *double group-by pull up*. Please refer to [Yan95] for the proof.

A simple way to ensure that the conditions of the corollary hold is to use GA_d^+ and GA_u^+ as NGA_d and NGA_u. Then, if (GA_u, GA_d) functionally determines the join columns, we can apply the Corollary.

Similarly, it is also possible to eliminate the group-by at the top query block after eager count, eager groupby-count and eager split to obtain the *push down versions* for these transformations, and analogly, the *pull up versions* for the lazy aggregations. Due to space limitation, we cannot provide the conditions here. Please refer to [Yan95] for detailed conditions and proofs. Note that, the push down/pull up version of eager/lazy group-by is group-by push down/pull up.

9 Eager Split and Lazy Split

If we apply eager groupby-count twice to R_d and R_u respectively, we can perform eager aggregation on both tables before the join. We call this transformation *eager split* since the aggregation is computed separately before the join. We call the reverse transformation *lazy split*. Both transformations are illustrated in Figure 2(g) and (h).

In the following corollary, (1) NGA_d and NGA_u denotes a set of columns in R_d and R_u, respectively; (2) CNT_1 the column produced by COUNT(*) after grouping $\sigma[C_d]R_d$ on NGA_d; (3) CNT_2 the column produced by COUNT(*) after grouping $\sigma[C_u]R_u$ on NGA_u;

(4) FAA_d the columns produced by F_d in the first aggregation of table $\sigma[C_d]R_d$ on NGA_d; (5) FAA_u the columns produced by F_u in the first aggregation of table $\sigma[C_u]R_u$ on NGA_u; and (6) F_{da} and F_{ua} the duplicated aggregation function of F_d and F_u, respectively. Also assume that (1) $AA = AA_d \cup_d AA_u$ where AA_d contains only columns in R_d, and AA_u contains only columns in R_u; (2) $F = F_d \cup_d F_u$ where F_d applies to AA_d and F_u applies to AA_u.

Corollary 5 (Eager Split and Lazy Split:) *The expressions*

$$\begin{aligned} E_1: \quad & F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u] \\ & \mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u) \end{aligned}$$

and

$$\begin{aligned} E_2: \quad & \pi_d[GA_d, GA_u, FAA] \\ & (F_{ua}[F_{u2}[FAA_u], CNT_1], F_{da}[F_{d2}[FAA_d], CNT_2]) \\ & \pi_A[GA_d, GA_u, FAA_u, FAA_d, CNT_1, CNT_2] \\ & \mathcal{G}[GA_d, GA_u]\sigma[C_0, C_u]((((F_{d1}[AA_d], COUNT[]) \\ & \pi_A[NGA_d, GA_d^+, AA_d]\mathcal{G}[NGA_d]\sigma[C_d]R_d) \\ & \times((F_{u1}[AA_u], COUNT[]) \\ & \pi_A[NGA_u, GA_u^+, AA_u]\mathcal{G}[NGA_u]\sigma[C_u]R_u)) \end{aligned}$$

are equivalent if (1) aggregation functions F_d contain only decomposable aggregation functions that can be decomposed into F_{d1} and F_{d2}; (2) aggregation functions F_u contain only decomposable aggregation functions that can be decomposed into F_{u1} and F_{u2}; (3) F_u and F_d contain class C or D aggregation functions; (4) $NGA_d \longrightarrow GA_d^+$ holds in $\sigma[C_d]R_d$; (5) $NGA_u \longrightarrow GA_u^+$ holds in $\sigma[C_u]R_u$.

The proof of this corollary is also straightforward. It can be done by first performing an eager/lazy groupby-count on R_d, and then an eager/lazy groupby-count on R_u.

10 Algorithms and Implementation

10.1 Algorithm for Eager Aggregation

In this section, we present a practical algorithm for recognizing all valid eager transformations for a given query. We assume that R_d tables contain aggregation columns and R_u tables do not. That is, all queries belong to the class of queries specified in Section 3.

10.1.1 Finding the Eager Grouping Columns for Eager Aggregation

Given two sets of tables, R_d and R_u, with R_d tables containing aggregation columns and R_u tables not, let's first consider eager group-by. We can start with NGA_d using GA_d^+ as the eager grouping columns(NGA_d). According to Corollary 1, we can add more R_d columns to NGA_d without changing the result of the query. Normally we want to choose a new set of grouping columns only if the new set has some ordering properties that save sorting time. For example, if the ordering property on a new column is supported by a clustering index, then after the new column is added into NGA_d, it can be used as the major of the sorting columns(assuming sorting is used for GROUP BY). The subsequent sort may be faster since the minor columns are sorted in a smaller range, plus the advantage of sequential fetching of data rows. In this case, even if one of the GA_d^+ columns has an index, since the index is not clustered, it may be more expensive to perform the grouping using GA_d^+ as the grouping columns than using the clustering index column and GA_d^+ as the grouping columns. Therefore, we want to consider possible beneficial addition of columns to GA_d^+ as eager grouping columns. We call such columns *promising columns.* Since adding new columns is often not beneficial, a good heuristic might be not to add grouping columns beyond GA_d^+.

When performing eager aggregation, our objective is to achieve data reduction before the join, so we want each partial group to contain as many rows as possible. Therefore, if NGA_d contains a unique key of $\sigma[C_d]R_d$, we should immediately abandon using this set for eager group-by.

10.1.2 Table Partitioning

When the query contains more than two tables, there may be several ways of performing eager aggregation. The question is how to partition the tables in the FROM clause into R_d tables and R_u tables. Section 10.3 discusses the way to partition tables to obtain all possible transformations. We assume that table partitioning has been done before calling the algorithm in Section 10.1.3.

10.1.3 The Algorithm

Assuming table partitioning is done, we have the following algorithm for finding valid eager aggregation. In this algorithm, we choose not to add new columns to either NGA_d or NGA_u. In the following algorithm, R_d tables must contain aggregation columns.

Algorithm 1 Eager Aggregation

```
Inputs: input query, $R_d$, $R_u$, AA
Output: all possible rewritten queries
    $NGA_d := GA_d^+$ and $NGA_u := GA_u^+$
    $eager_d = false$, $eager_u = false$
    if $NGA_d$ is not a unique key of $\sigma[C_d]R_d$
        $eager_d = true$
    end if
```

```
if $NGA_u$ is not a unique key of $\sigma[C_u]R_u$
    $eager_u = true$
end if
if $eager_u$ and $eager_d$
    if no aggregation columns in $R_u$
        Apply double eager on $R_d$ and $R_u$
        Output the rewritten query
    else
        Apply eager split on both $R_d$ and $R_u$
        Output the rewritten query
        Apply eager groupby-count on $R_d$
        Output the rewritten query
    end if
else if $eager_d$ and not $eager_u$
    if no aggregation columns in $R_u$
        Apply eager group-by on $R_d$
    else
        Apply eager groupby-count on $R_d$
    end if
    Output the rewritten query
else if not $eager_d$ and $eager_u$
    if no aggregation columns in $R_u$
        Apply eager count on $R_u$
    else
        Apply eager groupby-count on $R_u$
    end if
    Output the rewritten query
else
    Output "No transformation"
end if
```
END Algorithm 1

10.2 Algorithm for Lazy Aggregation

Now consider lazy aggregation. Whenever a query matches the form in any one of our theorems and satisfies their conditions, we can perform a lazy aggregation to eliminate one GROUP BY(or DISTINCT), and delay grouping until after the join. Lazy aggregation is especially useful when the join is highly selective. The algorithm to find all valid lazy aggregation transformation for a given query is to iterate through each available transformation and output the rewritten forms. Please refer to [Yan95] for a detailed description for the algorithm.

10.3 Implementation

We need to find a way to efficiently integrate eager/lazy aggregation and group-by push down/pull up into existing optimizers. The standard technique for determinating join order in a cost-based optimizer is dynamic programming in a bottom up (e.g., Starburst[Loh88]) fashion. During the dynamic programming process, plans for table accesses, two-table joins, three-table joins and joins involving more tables are constructed and kept until the final query plan is obtained. To integrate the transformations into such an optimizer, we can first perform group-by pull up and lazy aggregation to obtain a canonical form in which all group-bys are delayed as late as possible. Then, during dynamic programming process, whenever a table access plan or join plan is constructed, we can consider adding a group-by on top of the plan. All tables in the query are then partitioned into two sets, the set containing all tables in the current join plan, and the set containing the remaining tables. We can then apply Algorithm Eager Aggregation to find all possible eager aggregations. There can be several possible ways for adding an aggregation on top of a plan. The optimizer may want to choose the cheapest way for each plan to reduce optimization cost. Then, for each original join plan, there is at most one additional plan that performs a group-by at the top. On the other hand, when considering join plan for two input streams, the optimizer can consider the alternatives of taking the streams with or without aggregation. If the optimizer employs an exhaust search and considers all possible join plans in the dynamic programming process(e.g., Starburst), all possible transformations can be found in this process. This approach is also suitable for dynamic programming process that generates only left deep trees or right deep trees. However, it might overlook some possible rewrites.

11 Queries Including HAVING

A query with a HAVING clause can always be transformed into one without. This technique is well known and is used in existing database systems. For example, the Starburst optimizer always transforms a query with a HAVING into one without at the beginning of the query rewrite phase[PHH92]. After the HAVING is eliminated, we can perform eager aggregation transformation on the view created.

Now consider lazy aggregation. When a HAVING is eliminated in a subquery block with an aggregation (either group-by or distinct), and the HAVING clause contains no aggregations, then the predicate in the HAVING clause can be moved to the WHERE clause and we can then try to apply one of our lazy aggregation theorems. If the HAVING clause contains aggregations, we usually give up performing lazy aggregations because the HAVING predicates have to be evaluated before the join. However, it is possible to perform lazy aggregation when the HAVING clause of a query contains aggregation.

We formally proved our theorem for the conditions of group-by push down transformation for queries containing a HAVING clause in [YL95]. The process to prove conditions of eager aggregation for queries with a HAVING clause is completely analog to our previous effort. Due to space limitation we shall not present the conditions and proof here.

12 TPC-D Queries

We can apply group-by push down/pull up and eager/lazy aggregation to twelve of the seventeen queries in the TPC-D benchmark and significantly reduce the elapsed time of six queries on DB2 V2 Beta 3, as shown in Table 1. For example, it improves the elapsed time of Query 5 by a factor of ten. Table 2 shows the ratio between best and worst elapsed time for all TPC-D official queries that can be transformed[3]. The performance difference between a badly formed query and a better formed query can be very significant. Particularly, in applications where queries are generated by tools or inexperienced users, automatic transformation of queries is indeed very important.

In both Table 1 and 2, each table is represented by the first letter of its name, except that table PARTSUPP is represented by PS. Also, we use PD, PU, EG, EC and DC to represent group-by push down, group-by pull up, eager group-by, eager count and query decorrelation transformations respectively.

Table 1: TPC-D Queries With Reduced Elapsed Time (Compared With Original Formulation)

Query	Transformation	Reduction in Elapsed time
5	EG on L/O and C	90.23%
7	EG on L	61.36%
8	EG on L/P/S	47.69%
10	PD on L/O	8.34%
14	EG on L	6.73%
17	DC then PU	29.09%

13 Related Work

We proposed the idea of eager aggregation and lazy aggregation in [Yan94]. Chaudhuri and Shim[CS94] also independently discovered eager group-by and eager count. Their simple coalescing grouping and generalized coalescing grouping correspond to our eager group-by and eager count transformation, respectively. They also proposed an algorithm to integrate group-by push down, eager group-by and eager count into a greedy join enumeration algorithm which produces left deep trees in a cost based optimizer. However, they did not discuss lazy aggregation transformation in the paper.

Gupta, Harinarayan and Quass[GHQ95] generalized group-by push down in another fashion. They showed that it is possible to perform early duplicate removal before a join when there is no aggregation in the original query. The access plan must maintain a count of the number of duplicates being removed. Then, after or during the join, the access plan must restore the duplicates. Chaudhuri and Shim[CS95] also generalized group-by pull up to handle the case when the join is a many to many join.

[3] The ratio marked as 'infinity' means that the query with the worst formulation ran out of system space and did not finish.

Table 2: Ratio Between Best And Worst Elapsed Time For All TPC-D Queries That Can Be Transformed

Q	# of rewrites	Worst Formulation	Best Formulation	W/B Ratio
3	3	PD on L/O	Original	3.93
5	7	EG on L/O/C/S	EG on L/O/C	43.71
7	7	Original	EG on L	2.58
8	14	EG on L/S	EG on L/P/S	501.02
9	8	EG on L/PS/P	Original	infinity
10	4	EG on L	PD on L/O	1.20
11	9	EG on PS/S for both aggregations	Original	6.78
12	2	EC on L	Original	1.02
13	2	EG on L	Original	16.59
14	2	Original	EG on L	1.07
15	2	PU	Original	2.51
17	3	DC	DC then PU	25.58

14 Conclusion

Group-by push down and group-by pull up interchange the order of join and group-by. The number of group-by's is unchanged. Eager aggregation introduces an additional group-by before a join, and lazy aggregation eliminates a group-by before a join. Group-by push down and eager aggregation reduces the number of rows participating in a join, group-by pull up and lazy aggregation reduces the number of input rows to the group-by. Both directions of transformation should be considered during query optimization.

We classify eager aggregation into five different types: eager group-by, eager count, double eager, eager groupby-count and eager split. Eager group-by partially pushs down a group-by on the tables that contain all aggregation columns; eager count partially pushs down a group-by on the tables that do not contain any aggregation columns; double eager partially pushs down a group-by on both types of tables; eager groupby-count partially pushs a group-by into a subset of tables containing the aggregation columns; eager split splits a group-by into two group-bys and partially pushs the group-bys down the two input streams of the join. As a special case of double eager, we can completely push down group-by into two input streams,

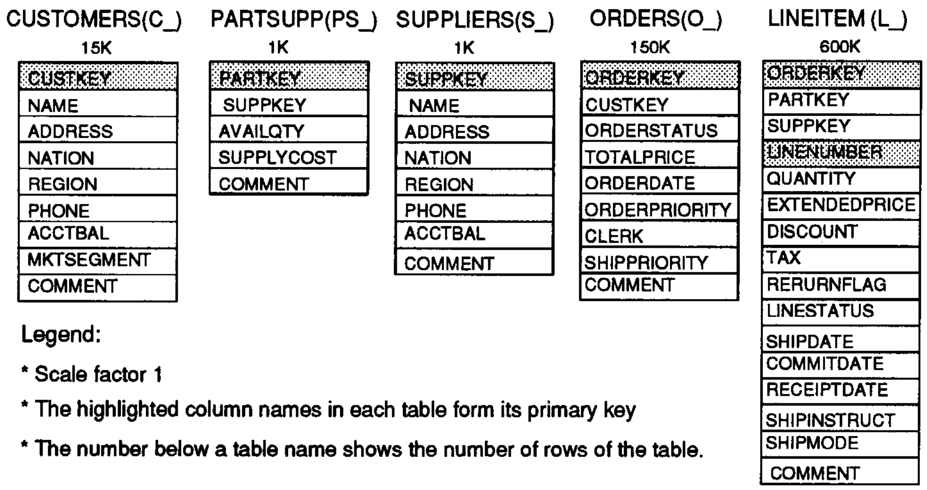

Figure 4: Subset of the TPC-D Database

which is call double group-by push down. Similarly, we classify lazy aggregation into lazy group-by, lazy count, double lazy, lazy groupby-count and lazy split, which perform the reverse transformations of their eager counterparts. We also provide practical algorithms for identifying all possible transformations. The algorithms do not restrict the join order of the query.

Future work includes: (1) finding the conditions of lazy transformation for subqueries containing an aggregating `HAVING` clause; (2) finding necessary and sufficient conditions for all the transformations in the paper and (3) finding eager/lazy aggregation transformations involving other binary relational operations (e.g., `UNION, INTERSECT, EXCEPT and OUTER JOIN`).

Acknowledgements

We thank the referees for their many useful comments. We also would like to thank Guy M. Lohman for suggesting the word *eager* and K. Bernhard Schiefer for his help with the TPC-D experiments. We also want to extend our appreciation to Surajit Chaudhuri and Kyuseok Shim for their valuable comments.

A The TPC-D Database

TPC-D is a decision support benchmark proposed by the the *Transaction Processing Performance Council(TPC)*. It is a suite of business oriented queries to be executed against a database that allows continuous access as well as concurrent updates[Raa95]. The size of the database is scalable adjusted by a *scale factor*. The scale factor for a 100MB database is 0.1. The size of the database we used through out this paper is 100MB. Figure 4 shows the subset of the TPC-D database we used in this paper.

References

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. VLDB Conf.*, pages 354–366, Santiago, Chile, Sep. 1994.

[CS95] S. Chaudhuri and K. Shim. Optimizing complex queries: A unifying approach. Tech. Report HPL-DTD-95-20, HP Lab, Mar. 1995.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. VLDB Conf.*, 1995.

[ISO92] ISO. *Information Technology - Database languages - SQL*. Reference number ISO/IEC 9075:1992(E), Nov. 1992.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD Conf.*, pages 18–27, Chicago, Illinois, June 1988.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in STARBURST. In *Proc. SIGMOD Conf.*, pages 39–48, San Diego, California, June 1992.

[Raa95] F. Raab, editor. *TPC Benchmark(tm) D (Decision Support), Working Draft 9.1.* Transaction Processing Performance Council, San Jose CA, 95112-6311, USA, February 1995.

[Yan94] W. P. Yan. Query optimization techniques for aggregation queries. Research Proposal, University of Waterloo, April 1994.

[Yan95] W. P. Yan. *Rewrite optimization of* SQL *queries containing* GROUP-BY. PhD thesis, Department of Comp. Sci., University of Waterloo, Sep. 1995.

[YL94] W. P. Yan and Per-Åke Larson. Performing group-by before join. In *Proc. IEEE ICDE*, pages 89–100, Houston, Texas, Feb. 1994.

[YL95] W. P. Yan and Per-Åke Larson. Interchanging the order of grouping and join. Technical Report CS 95-09, University of Waterloo, Feb. 1995.