Performing Group-By before Join

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

Appears in *Proceedings of the 10th IEEE International Conference on Data Engineering, p89–100, Houston, TX, 1994*

Abstract

Assume that we have an SQL *query containing joins and a group-by. The standard way of evaluating this type of query is to first perform all the joins and then the group-by operation. However, it may be possible to perform the group-by early, that is, to push the group-by operation past one or more joins. Early grouping may reduce the query processing cost by reducing the amount of data participating in joins. We formally define the problem, adhering strictly to the semantics of* NULL *and duplicate elimination in* SQL*2, and prove necessary and sufficient conditions for deciding when this transformation is valid. In practice, it may be expensive or even impossible to test whether the conditions are satisfied. Therefore, we also present a more practical algorithm that tests a simpler, sufficient condition. This algorithm is fast and detects a large subclass of transformable queries.*

1 Introduction

SQL queries containing joins and group-by are fairly common. The standard way of evaluating such a query is to perform all joins first and then the group-by operation. However, it may be possible to interchange the evaluation order, that is, to push the group-by operation past one or more joins.

Example 1 *:* Assume that we have the two tables:

```
Employee(EmpID, LastName, FirstName, DeptID)
Department(DeptID, Name)
```

EmpID is the primary key in the Employee table and DeptID is the primary key of Department. Each Employee row references the department (DeptID) to which the employee belongs. Consider the following query:

```
SELECT      D.DeptID, D.Name, COUNT(E.EmpID)
FROM        Employee E, Department D
WHERE       E.DeptID = D.DeptID
GROUP BY    D.DeptID, D.Name
```

Plan 1 in Figure 1 illustrates the standard way of evaluating the query: fetch the rows in tables E and D, perform the join, and group the result by D.DeptID and D.Name, while at the same time counting the number of rows in each group. Assuming that there are 10000 employees and 100 departments, the input to the join is 10000 Employee rows and 100 Department rows and the input to the group-by consists of 10000 rows. Now consider Plan 2 in Figure 1. We first group the Employee table DeptID and perform the COUNT, then join the resulting 100 rows to the 100 Department rows. This reduces the join from (10000×100) to (100×100). The input cardinality of the group-by remains the same, resulting in an overall reduction of query processing time. □

In the above example, it was both possible and advantageous to perform the group-by operation before the join. However, it is also easy to find examples where this is (a) *not* possible or (b) possible but *not* advantageous. This raises the following general questions:

1. Exactly under what conditions is it possible to perform a group-by operation before a join?
2. Under what conditions does this transformation reduce the query processing cost?

This paper concentrates on answering the first question. Our main theorem provides sufficient and necessary conditions for deciding when this transformation is valid. The conditions cannot always be tested efficiently so we also propose a more practical algorithm which tests a simpler, sufficient condition.

The rest of the paper is organized as follows. Section 2 summarizes related research work. Section 3

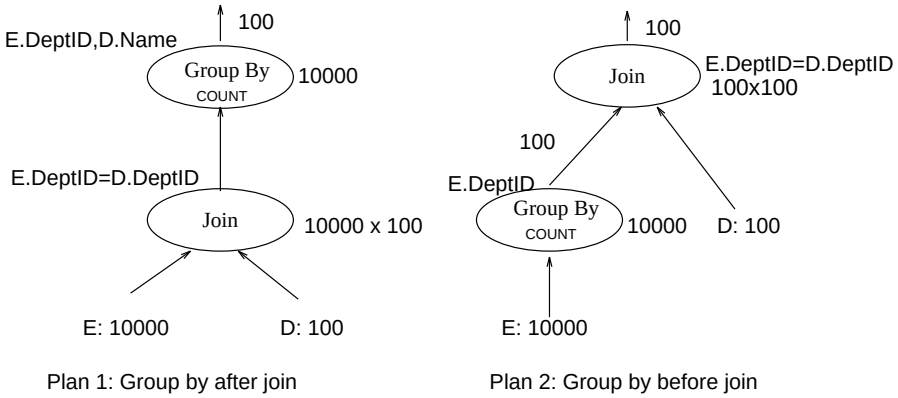


Figure 1: Two access plans for Example 1

defines the class of queries that we consider. Section 4 presents the formalism that our results are based on. Section 4.1 presents an SQL2 algebra whose operations are defined strictly in terms of SQL2. Section 4.2 discusses the semantics of NULLs in SQL2. Section 4.3 formally defines functional dependencies using strict SQL2 semantics taking into account the effect of NULLs, and discusses derived functional dependencies. Section 5 introduces and proves the main theorem, which states necessary and sufficient conditions for performing the proposed transformation. Section 6 proposes an efficient algorithm for deciding whether group-by can be pushed past a join. Section 7 continues some observations about the trade-offs of the transformation. Section 8 points out that the reverse direction of the transformation is also possible and sometimes can be beneficial. Section 9 concludes the paper.

2 Related work

We have not found any papers dealing with the problem of performing group-by before joins. However, some results have been reported on the processing of queries with aggregation. It is widely recognized that the aggregation computation can be performed while grouping(which is usually implemented by sorting). This can save both time and space since the amount of data to be sorted decreases during sorting. This technique is referred to as pipelining.

Klug[6] observed that in some cases, the result from a join is already grouped correctly. Nested-loop and sort merge joins, the most widely used join methods, both have this property. In this case, explicit grouping is not needed and the join can be pipelined with aggregation. Dayal[2] stated, without proof, that the necessary condition for such technique is that the group-by columns must be a primary key of the outer table in the join. This is the only work we know of which attempts to reduce the cost of group-by by utilizing information about primary keys.

Several researchers ([5, 4, 3, 10, 9]) have investigated when a nested query can be transformed into a semantically equivalent query that does not contain nesting. As part of this work, techniques to handle aggregate functions in the nested query were discussed. However, none considered interchanging the order of joins and group-by.

3 Class of queries considered

A table can be a base table or a view in this paper. 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 be drawn from more than one table. Clearly, the transformation cannot be applied unless at least one table contains no aggregation columns. Therefore, we partition the tables in the FROM clause into two groups: those tables that contain at least one aggregation column and those that do 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_1 and R_2. Let R_1 denote the table containing aggregation columns and R_2 the table not containing any such columns.

The search conditions in the WHERE clause can be expressed as $C_1 \wedge C_0 \wedge C_2$, where C_1, C_0, and C_2 are in conjunctive normal form, C_1 only involves columns

in R_1, C_2 only involves columns in R_2, and each disjunctive component in C_0 involves columns from both R_1 and R_2. Note that subqueries are allowed.

The **grouping columns** mentioned in the GROUP BY clause may contain columns from R_1 and R_2, denoted by GA_1 and GA_2, respectively. According to SQL2[7], the selection columns in the SELECT clause must be a subset of the grouping columns. We denote the selection columns as SGA_1 and SGA_2, subsets of GA_1 and GA_2, respectively. For the time being, we assume that the query does not contain a HAVING clause. The columns of R_1 participating in the join and grouping is denoted by GA_1^+, and the columns of R_2 participating in the join and grouping is denoted by GA_2^+.

In summary, we consider queries of the following form:

SELECT [ALL/DISTINCT]	SGA_1, SGA_2, $F(AA)$
FROM	R_1, R_2
WHERE	$C_1 \wedge C_0 \wedge C_2$
GROUP BY	GA_1, GA_2

where:

- GA_1: grouping columns of table R_1;
- GA_2: grouping columns of table R_2; (GA_1 and GA_2 cannot both be empty. If they are, the query does not contain a group-by clause)
- SGA_1: selection columns, must be a subset of grouping columns GA_1;
- SGA_2: selection columns, must be a subset of grouping columns GA_2;
- AA: aggregation columns of table R_1 (may be * or empty);
- C_1: conjunctive predicates on columns of table R_1;
- C_2: conjunctive predicates on columns of table R_2;
- C_0: conjunctive predicates involving columns of both tables R_1 and R_2, e.g., join predicates;
- $\alpha(C_0)$: columns involved in C_0;
- F: array of arithmetic expressions applied on AA (may be empty);
- GA_1^+: $\equiv GA_1 \cup \alpha(C_0) - R_2$, i.e., the columns of R_1 participating in the join and grouping;
- GA_2^+: $\equiv GA_1 \cup \alpha(C_0) - R_1$, i.e., the columns of R_2 participating in the join and grouping

Our objective is to determine under what conditions the query can be evaluated in the following way:

SELECT [ALL/DISTINCT]	SGA_1, SGA_2, FAA
FROM	R_1', R_2'
WHERE	C_0

where

$R_1'(GA_1+, FAA) ==$	
SELECT ALL	$GA_1+, F(AA)$
FROM	R_1
WHERE	C_1
GROUP BY	GA_1+

and

$R_2'(GA_2+) ==$	
SELECT ALL	GA_2+
FROM	R_2
WHERE	C_2

In SQL2, $F(AA)$ transfers a group of rows into one single row, even when $F(AA)$ is empty. Therefore, through out this paper, the only assumption we make about $F(AA)$ is that it produces one row for each group.

4 Formalization

In this section we define the formal "machinery" we need for the theorems and proofs to follow. This consists of an algebra for representing SQL queries and clarification of the effect of NULLs on comparisons, duplicate eliminations, and functional dependencies when using strict SQL2 semantics.

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 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 [1] 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.

[1] 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.

- $\pi_d[B]R$, where $d = A$ or D: Project table R on columns B, without eliminating duplicates when 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 expressions operating on AA. For $i = 1, 2, ..., n$, f_i is an arithmetic expression(which can just 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 `COUNT`$(A_1) +$ `SUM`$(A_2 + A_3)$. Duplicates in the overall result are not eliminated. This operation is defined by the query `SELECT GA, F(AA) FROM R GROUP BY GA`, where GA is the grouping columns of R.

We also use $\Rightarrow$, $\Longleftrightarrow$, $\wedge$ and $\vee$ to represent logical implication, logical equivalence, logical conjunction and logical disjunction respectively. Then, the class of `SQL` queries we consider can be expressed as [2]:

$$F[AA]\pi_d[SGA_1, SGA_2, AA]\mathcal{G}[GA_1, GA_2] \\ \sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2).$$

Our objective is to determine under what conditions this expression is equivalent to

$$\pi_d[SGA_1, SGA_2, FAA]\sigma[C_0](F[AA]\pi_A[GA_1+, AA] \\ \mathcal{G}[GA_1]\sigma[C_1]R_1 \times \pi_A[GA_2+]\sigma[C_2]R_2)$$

where FAA are the columns generated by applying the arithmetic expressions F to columns AA.

4.2 The semantics of NULL in SQL2

`SQL2`[7, 8, 1] represents missing information by a special value `NULL`. It adopts a three-valued logic in evaluating a conditional expression, having three possible truth values, namely `true`, `false` and `unknown`. Figure 2 shows the truth tables for the Boolean operations `AND` and `OR`. Testing the equality of two values in a search condition returns `unknown` if any one of the values is `NULL` or both values are `NULL`. A row qualifies only if the condition in the `WHERE` clause evaluates to `true`, that is, `unknown` is interpreted as `false`.

However, the effect of `NULL`s on duplicate operations is different. Duplicate operations include `DISTINCT`, `GROUP BY`, `UNION`, `EXCEPT` and `INTERSECT`, which all involve the detection of duplicate rows. Two rows are defined to be *duplicates* of one another exactly when each pair of corresponding column values are duplicate. Two column values are defined to be *duplicates* exactly when they are equal and both not `NULL` or when they are both `NULL`s. In other words, `SQL2` considers "`NULL` equal to `NULL`" when determining duplicates.

AND	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

OR	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

Figure 2: The semantics of `AND` and `OR` in `SQL2`

Note that we do not include the `UNIQUE` predicate among the duplicate operations. `SQL2` uses "`NULL` not equal to `NULL`" semantics when considering `UNIQUE`.

We need[3] some special 'interpreters' capable of transferring the three-valued result to the usual two-valued result based on `SQL2` semantics in order to formally define functional dependencies and `SQL` operations. We adopt two interpretation operators $\lfloor P \rfloor$ and $\lceil P \rceil$ specified in Figure 3 for interpreting `unknown` to `false` and `true` respectively. In addition, a special equality operator, $\stackrel{n}{=}$, which is also specified in Figure 3, is proposed to reflect the "`NULL` equal to `NULL`" characteristics of `SQL` duplicate operations.

4.3 Functional dependencies

`SQL2` [7] provides facilities for defining (primary) keys of base tables. Note that a key definition implies two constraints: (a) no two rows can have the same key value and (b) no column of a key can be `NULL`. We can exploit knowledge about keys to determine whether the proposed transformation is valid.

Defining a key implies that all columns of the table are functionally dependent on the key. This type of functional dependency is called a *key dependency*. Keys can be defined for base tables only. For our purpose, *derived* functional dependencies are of more in-

[2] In the case that there exists $f_i(A_i) \equiv$ `COUNT`$(*) \in F(AA)$, we can replace it with `COUNT`$(GA1)$ without changing the result of the query.

[3] There certainly exist other solutions to this problem. We just present the one we think is most appropriate for our purpose.

Operation	Result		
P is a predicate	P is `true`	P is `unknown`	P is `false`
P	`true`	`unknown`	`false`
$\lfloor P \rfloor$	`true`	`false`	`false`
$\lceil P \rceil$	`true`	`true`	`false`
X, Y are variables	X is `NULL` & Y is `NULL`		Otherwise
$X \stackrel{n}{=} Y$	`true`		$\lfloor X = Y \rfloor$

Figure 3: The definition of interpretation operators

terest. A derived table is a table defined by a query (or view). A derived functional dependency is a functional dependency that holds in a derived table. Similarly, a derived key dependency is a key dependency that holds in a derived table. The following example illustrates derived dependencies.

Example 2 : Assume that we have the following two tables:

```
Part(ClassCode, PartNo, PartName, SupplierNo)
Supplier(SupplierNo, Name, Address)
```

where(`ClassCode, PartNo`) is the key of `Part` and `SupplierNo` is the key of `Supplier`. Consider the derived table defined by

```
SELECT    P.PartNo, P.PartName,
          S.SupplierNo, S.Name
FROM      Part P, Supplier S
WHERE     P.ClassCode = 25 and
          P.SupplierNo = S.SupplierNo
```

We claim that `PartNo` is a key of the derived table. The reasoning goes as follows. Clearly, `PartNo` is a key of the derived table `T` defined by $T = \sigma[ClassCode = 25](Part)$. When `T` is joined with `Supplier`, each row joins with at most one `Supplier` row because `SupplierNo` is the key of `Supplier`. (If `P.SupplierNo` is `NULL`, the row does not join with any `Supplier` row.) Consequently, `PartNo` remains a key of the joined table and also of the final result table obtained after projection.

In `Supplier`, `Name` is functionally dependent on `SupplierNo` because `SupplierNo` is a key of `Supplier`. It is obvious that this functional dependency must still hold in the derived table. That is, a key dependency in one of the source tables resulted in a non-key functional dependency in the derived table. □

Even though `SQL` does not permit `NULL` values in any columns of a key, columns on the right hand side of a key dependency may allow `NULL` values. In a derived dependency, columns allowing `NULL` values may occur on both the left and the right hand side of a functional dependency. The essence of the problem is how to define the result of the comparison `NULL` = `NULL`.

Consider a row $t \in r$, where r is an instance of a table R. Assuming that a is an column of R, we denote the value of a in t as $t[a]$.

Definition 1:(*Row Equivalence*): Consider a table scheme $R(..., A, ...)$, where A is a set of columns $\{a_1, a_2, ..., a_n\}$, and an instance r of R. Two rows t, $t' \in r$ are *equivalent with respect to* A if:

$$\bigwedge_{i=1,...,n} (t[a_i] \stackrel{n}{=} t'[a_i]),$$

which we also write as $t[A] \stackrel{n}{=} t'[A]$.

Definition 2: (*Functional Dependency*) Consider a table $R(A, B, ...)$, where $A = \{A_1, A_2, ..., A_n\}$ is a set of columns and B is a single column. Let r be an instance of R. *A functionally determines* B, denoted by $A \longrightarrow B$, in r if the following condition holds:

$$\forall t, t' \in r, \{(t[A] \stackrel{n}{=} t'[A]) \Rightarrow (t[B] \stackrel{n}{=} t'[B])\}.$$

Let $Key(R)$ denote a candidate key of table R. We can now formally specify a key dependency as

$$\forall r(R), \forall t, t' \in r,$$
$$\{t[Key(R)] \stackrel{n}{=} t'[Key(R)] \Rightarrow t[\alpha(R)] \stackrel{n}{=} t'[\alpha(R)]\}.$$

Note that, since `NULL` is allowed for a candidate key, we need to consider "`NULL` equals to `NULL`" condition in the statement.

The basic data type in `SQL` is a table, not relation. A table may contain duplicate rows and is therefore a multiset. In this paper, we use the term 'set' to refer to 'multiset'. In order to distinguish the duplicates in a table in our analysis, we assume that there always exists a column in each table called "`RowID`", which can uniquely identify a row. It is not important whether this column is actually implemented by the underlining database system. We use `RowID`(R) to denote the `RowID` column of a table R.

We use the notation $E(r_1, r_2)$ to denote the result generated by an `SQL` expression E evaluating on instances r_1 and r_2 of tables R_1 and R_2, respectively. We summarize all symbols defined in Section 4.2 and this section in Figure 4. The symbol "$\circ$" is also defined as the concatenation operator.

5 Theorems and proofs

Theorem 1 (Main Theorem): *The expressions*

Symbol	Definitions
r_1, r_2	Instances of table R_1 and R_2
$A \circ B$	the concatenation of two rows A and B into one row
$g \circ B$	the concatenation of a grouped table g and a row B into one new grouped table. Each row in the new grouped table is the result of a row in g concatenates with B.
$T[S]$	shorthand for $\pi_A[S]T$, where S is a set of columns and T is a grouped or ungrouped table, or a row.
$E(r_1, r_2)$	the result from applying E on instances r_1 and r_1.
RowID(R)	the RowID of table R

Figure 4: Summary of symbols

$$E_1 : F[AA]\pi_A[GA_1, GA_2, AA]\mathcal{G}[GA_1, GA_2] \sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$$

and

$$E_2: \pi_A[GA_1, GA_2, FAA]\sigma[C_0](F[AA]\pi_A[GA_1+, AA] \mathcal{G}[GA_1+]\sigma[C_1]R_1 \times \pi_A[GA_2+]\sigma[C_2]R_2)$$

are equivalent **if and only if** *the following two functional dependencies hold in the join of* R_1 *and* R_2, $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$*:*

$$FD_1 : (GA_1, GA_2) \longrightarrow GA_1+$$
$$FD_2 : (GA_1+, GA_2) \longrightarrow \texttt{RowID}(R_2).$$

FD_2 means that for all valid instances r_1 and r_2 of R_1 and R_2, respectively, if two different rows in $\sigma[C_d \wedge C_0 \wedge C_u]$ $(r_d \times r_u)$ have the same value for columns (GA_1+, GA_2), then the two rows must be produced from the join of one row in $\sigma[C_2]r_2$ and two rows (could be duplicates) in $\sigma[C_1]r_1$.

Note that R_2 does not necessarily have to include a column RowID. The notation "$(GA_1+, GA_2) \longrightarrow$RowID$(R_2)$ in the join of R_1 and R_2" is simply a shorthand for the requirement that (GA_1+, GA_2) uniquely identifies a row of R_2 in the join of R_1 and R_2.

The intuitive meaning of FD_1 and FD_2 is as follows. FD_1 ensures that each group in $\mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$ (grouped by GA_1, GA_2 on the join result of R_1 and R_2, using E_1 for the query) corresponds to exactly one group in $\mathcal{G}[GA_1+]\sigma[C_1]R_1$ (grouped by GA_1+ on the selection result of R_1, using E_2 for the query). Exact correspondence means that there is an one to one matching between rows in the two groups, with matching rows having the same value for the columns of R_1. This condition guarantees that these two groups, based on E_1 and E_2 respectively for the query, produce the same aggregation value. Note that the aggregation functions and arithmetic expressions only operate on columns of R_1.

FD_2 ensures that each row in $F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]R_1$ (grouping and aggregating on R_1, using E_2 for the query) contributes at most one row in the overall result of E_2 by joining with at most one row from $\sigma[C_2]R_2$. In other words, FD_2 prevents such a row from contributing two or more rows in the overall result of E_2. The rationale of FD_2 is that if such a row does contribute two or more rows in the overall result of E_2, then, since (a) the rows corresponding to these rows before the aggregation will belong to the same group in $\mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$ (grouped by GA_1, GA_2 on the join result of R_1 and R_2, using E_1 for the query), and (b) each group in $\mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$ yields one row in the overall result of E_1, therefore, E_1 contains one row correspodning to more than one rows in E_2, and consequently the transformation cannot be valid.

For some proofs in this paper, we only present brief sketches to save space. We also omit proofs that are trivial. Complete proofs can be found [11].

Lemma 1 : *The expression*

$$E_2': \pi_A[GA_1, GA_2, FAA]\sigma[C_0](F[AA] \pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]R_1 \times \sigma[C_2]R_2)$$

is equivalent to E_2.

The difference between E_2 and E_2' is that E_2' does not remove the columns other than GA_2+ of table $\sigma[C_2]R_2$ before the join. In practice, the optimizer usually removes these unnecessary columns to reduce the data volume. See [11] for the proof.

It follows from Lemma 1 that we only need to prove that E_1 is equivalent to E_2' if and only if FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$. Lemmas 2 - 6 essentially prove the Main Theorem in the case when GA_1+ and GA_2+ are both non-empty. The proof is derived into several steps: Lemma 2 and Lemma 3 show the necessity of FD_1 and FD_2; Lemma 4 and Lemma 5 demonstrate that there are no duplicates in the result of E_1 and E_2'; Lemma 6 proves the sufficiency. Finally we prove the Main Theorem based on these lemmas.

5.1 Necessity

Lemma 2 : *If the two expressions* E_1 *and* E_2' *are equivalent, and* GA_1+ *and* GA_2+ *are both non-empty, then* FD_1 *holds in* $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$.

Proof(sketch): We prove the lemma by contradiction. Assume that E_1 and E_2' are equivalent, and GA_1+ and GA_2+ are both non-empty, but FD_1 does not hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$. Then there must exist two valid instances r_1 and r_2 of R_1 and R_2, respectively, with the following properties: (a) $E_1(r_1, r_2)$ and $E_2'(r_1, r_2)$ produce the same result and (b) there exist two rows t and $t' \in \sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ such that $t[GA_1, GA_2] \stackrel{n}{=} t'[GA_1, GA_2]$ but $t[GA_1+] \stackrel{n}{\neq} t'[GA_1+]$. Then, $E_1(r_1, r_2)$ can be shown to have exactly one row whose value for columns $[GA_1, GA_2]$ is $t[GA_1, GA_2]$; and $E_2'(r_1, r_2)$ can be shown to have at least two rows with the same value $t[GA_1, GA_2]$ for columns $[GA_1, GA_2]$. Therefore, $E_1(r_1, r_2)$ and $E_2(r_1, r_2)'$ cannot be equivalent. □

Lemma 3 : *If the two expressions E_1 and E_2' are equivalent, and GA_1+ and GA_2+ are both non-empty, then FD_2 holds in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$.*

Proof(sketch): We prove the lemma by contradiction. Assume that E_1 and E_2' are equivalent, and GA_1+ and GA_2+ are both non-empty, but FD_2 does not hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$. Then, there must exist two valid instances r_1 and r_2 of R_1 and R_2, respectively, with the following properties: (a) $E_1(r_1, r_2) = E_2'(r_1, r_2)$, and (b) there exist two rows t and $t' \in \sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ such that $t[GA_1+, GA_2] \stackrel{n}{=} t'[GA_1+, GA_2]$ but $t[\alpha(R_2)] \stackrel{n}{\neq} t'[\alpha(R_2)]$. Then, $E_1(r_1, r_2)$ can be shown to have exactly one row having the value $t[GA_1, GA_2]$ for columns $[GA_1, GA_2]$; and $E_2'(r_1, r_2)$ can be shown to have at least two rows with the same value $t[GA_1, GA_2]$ for columns $[GA_1, GA_2]$. Therefore, $E_1(r_1, r_2)$ and $E_2'(r_1, r_2)$ cannot be equivalent, and the Lemma is proved. □

Lemmas 2 and 3 prove that FD_1 and FD_2 must hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$ if E_1 and E_2' are equivalent and GA_1+ and GA_2+ are both non-empty.

5.2 Distinctness

Lemma 4 : *The table produced by expression E_1 contains no duplicate rows.*

See [11] for the proof.

Lemma 5 : *If FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$, and GA_1+ and GA_2+ are both non-empty, then there are no duplicate rows in the table produced by expression E_2'.*

Proof(sketch): We prove the lemma by contradiction. Assume that there exist two valid instances r_1 and r_2 of R_1 and R_2, respectively, such that, FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$, but there exist two different rows $t, t' \in E_2'(r_1, r_2)$ which are duplicates of each other, that is, $t \stackrel{n}{=} t'$. Then there must exist two rows, $t_1, t_1' \in \sigma[C_0](F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1 \times \sigma[C_2]r_2)$, such that $t = t_1[GA_1, GA_2, FAA]$, and $t' = t_1'[GA_1, GA_2, FAA]$. t_1 and t_1' must be produced by the join between rows in $F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1$ and $\sigma[C_2]r_2$. Assume $t_1 = t_{21} \circ t_{22}$ and $t_1' = t_{21}' \circ t_{22}'$, where $t_{21}, t_{21}' \in F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1$ and $t_{22}, t_{22}' \in \sigma[C_2]r_2$. There are only two cases to consider. Case 1 is when $t_{21}[GA_1+] \stackrel{n}{\neq} t_{21}'[GA_1+]$, which can be shown to lead to $t_{21}[GA_1+] \stackrel{n}{=} t_{21}'[GA_1+]$, which is a contradiction. Case 2 is when $t_{21}[GA_1+] \stackrel{n}{=} t_{21}'[GA_1+]$, which can be shown to lead to the fact that t and t' must be the same row, which is again a contradiction. □

5.3 Sufficiency

Lemma 6 : *If FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$, and GA_1+ and GA_2+ are both non-empty, then the two expressions E_1 and E_2' are equivalent.*

Proof(sketch): Lemma 4 and Lemma 5 guarantee that neither E_1 nor E_2' produces duplicate rows if GA_1+ and GA_2+ are both non-empty. Let r_1 and r_2 be valid instances of R_1 and R_2 respectively. All we need to prove is that, provided that GA_1+ and GA_2+ are both non-empty, if $t \in E_1(r_1, r_2)$, then $t \in E_2'(r_1, r_2)$; and vice versa.

First we want to prove that if $t \in E_1(r_1, r_2)$, then $t \in E_2'(r_1, r_2)$. For any t in $E_1(r_1, r_2)$, there must exist a group g in $\mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ to produce t in $E_1(r_1, r_2)$. It can be proved that g is produced by a join between exactly one tuple t_2 in $\sigma[C_2]r_2$ and a subset g_1 from $\sigma[C_1]r_1$. We can then go on to prove that g_1 is a group in $\mathcal{G}[GA_1+]\sigma[C_1]r_1$. Therefore the row generated by aggregating g_1 in $E_1(r_1, r_2)$ joins with t_2 can produce t in $E_2'(r_1, r_2)$.

Secondly we want to prove: $t \in E_2'(r_1, r_2) \Rightarrow t \in E_1(r_1, r_2)$. A similar approach as in the first case can be employed to show that this is true. □

Proof of the Main Theorem(sketch):

For the case that GA_1+ and GA_2+ are both non-empty, Lemma 2 and Lemma 3 prove that FD_1, FD_2 must hold in $\sigma[C_d \wedge C_0 \wedge C_u]$ $(r_d \times r_u)$ if E_1 and E_2' are equivalent(necessity). Lemma 6 shows that E_1 and E_2' are equivalent if FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u]$ $(r_d \times r_u)$(sufficiency). Lemma 1 ensures that $E_2 = E_2'$. These lemmas together prove

the theorem for case that GA_1+ and GA_2+ are both non-empty. GA_1+ and GA_2+ cannot both be empty because in that case (GA_1, GA_2) would be empty and the query does not belong to the class of queries we consider. Therefore there are two cases left to consider.

Case 1: GA_1+ is empty but GA_2+ is not empty. Then, it can be proved that E_1 and E_2' degenerate to:

$$E_1: \quad F[AA]\pi_A[GA_2, AA]\mathcal{G}[GA_2] \\ \sigma[C_1 \wedge C_2](R_1 \times R_2)$$

and

$$E_2: \quad \pi_A[GA_2, FAA](F[AA]\pi_A[AA]\sigma[C_1]R_1 \\ \times \pi_A[GA_2+]\sigma[C_2]R_2).$$

and FD_1 and FD_2 degenerate to: $(GA_2) \longrightarrow \phi$ and $(GA_2) \longrightarrow RowId(R_2)$, respectively. Note that, FD_1 is always true. Thus the necessary and sufficient condition is that FD_2 holds in $\sigma[C_1 \wedge C_2](R_1 \times R_2)$. It is then easy to prove that the Main Theorem is true.

Case 2: GA_2+ is empty but GA_1+ is not empty. Since GA_2+ is empty, GA_2 and C_0 must be empty. Therefore the join is a Cartesian product. Since C_0 is empty, GA_1+ must be the same as GA_1. Hence, E_1 and E_2 degenerate to:

$$E_1: \quad F[AA]\pi_A[GA_1, AA]\mathcal{G}[GA_1] \\ \sigma[C_1 \wedge C_2](R_1 \times R_2)$$

and

$$E_2: \quad \pi_A[GA_1, FAA]\sigma[C_0](F[AA]\pi_A[GA_1, AA] \\ \mathcal{G}[GA_1]\sigma[C_1]R_1 \times \sigma[C_2]R_2)$$

respectively, and FD_1 and FD_2 degenerate to $(GA_1) \longrightarrow GA_1$ and $(GA_1) \longrightarrow RowId(R_2)$ respectively. It is then easy to prove that the Main Theorem holds when GA_2+ is empty. □

Theorem 2 : *Consider the following two expressions:*

$$F[AA]\pi_d[SGA_1, SGA_2, AA] \\ \mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$$

and

$$\pi_d[SGA_1, SGA_2, FAA]\sigma[C_0](F[AA]\pi_A[GA_1+, AA] \\ \mathcal{G}[GA_1+]\sigma[C_1]R_1 \times \pi_A[GA_2+]\sigma[C_2]R_2),$$

where d is either A or D. The two expressions are equivalent if FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$.

Note that, the Main Theorem assumes that the final selection columns are the same as the grouping columns(GA_1, GA_2) and the final projection must be an `ALL` projection; this theorem relaxes these two restrictions, i.e., the final selection columns can be a subset(SGA_1, SGA_2) of the grouping columns(GA_1, GA_2), and the final projection can be a `DISTINCT` projection. Consequently, the two conditions FD_1 and FD_2 become sufficient but not necessary. See [11] for the proof.

6 TestFD: a fast algorithm to test the condition

To apply the transformation in Theorem 2, i.e., to push grouping past a join, we need an algorithm to test whether the functional dependencies FD_1 and FD_2 are guaranteed to hold in the join result of R_1 and R_2. To achieve this, we can make use of semantic integrity constraints and the conditions specified in the query. `SQL2` [7] allows users to specify integrity constraints on the valid state of `SQL` data and these constraints are enforced by the `SQL` implementation. Therefore, in any valid database instance, we can assume that all integrity constraints hold in the join result of R_1 and R_2. Similarly, the conditions of the query also hold in the join result. We can make use of this information to determine whether the functional dependencies FD_1 and FD_2 hold.

In [11], we proposed a method to test the conditions FD_1 and FD_2 which exploits the semantic constraints in `SQL`.

In this section, we will present an efficient algorithm, which uses key constraints, equality join predicates and column and domain constraints in `SQL`, to handle a large subclass of queries. This algorithm returns `YES` when it can determine that FD_1 and FD_2 hold in the join result $\sigma[C_1 \wedge C_0 \wedge C2](R_1 \times R_2)$, and returns `NO` when it cannot.

Atomic conditions not involving '=' are seldom useful for generating new functional dependencies. We define two types of atomic conditions: Type 1 of the form $(v = c)$ and Type 2 of the form $(v1 = v2)$, where $v1, v2, v$ are columns and c is a constant or a host variable. A host variable can be handled as a constant because its value is fixed when evaluating the query. We use T_1 and T_2 to denote the Boolean expressions representing domain and column constraints[4] of table R_1 and R_2. The algorithm follows:

.....

Algorithm TestFD: *determine whether group-by can be performed before join.*
Input: *Predicates C_1, C_0, C_2, T_1, T_2; key constraints of R_1 and R_2.*
Output: `YES` *or* `NO`.

[4] See [11] for delailed description for these constraints.

1. Convert $C_1 \wedge C_0 \wedge C_2 \wedge T_1 \wedge T_2$ into conjunctive normal form: $C = D_1 \wedge D_2 \wedge ... \wedge D_m$.

2. For each D_i, if D_i contains an atomic condition not of Type 1 or Type 2, delete D_i from C.

3. If C is empty, return NO and stop. Otherwise convert C into disjunctive normal form: $C = E_1 \vee E_2 \vee ... \vee E_n$.

4. For each conjunctive component E_i of C do

   (a) Create a set S containing all columns in GA_1 and GA_2.

   (b) For each atomic condition of Type 1 ($v = c$) in E_i, add v into S.

   (c) Compute the transitive closure of S based on Type 2 atomic conditions in E_i and the key constraints. That is, perform the operation: while (($\exists$ a Type 2 condition $v1 = v2 \in C$ such that $v1 \in S$ and $v2 \notin S$) or ($\exists Key(R_1) \in S$ and $v2 \in R_1$ and $v2 \notin S$) or ($\exists Key(R_2) \in S$ and $v2 \in R_2$ and $v2 \notin S$)), add $v2$ to S.

   (d) If a (primary or candidate) key of R_2 is in S, proceed. Otherwise return NO and stop.

   (e) Create a set S containing all columns in GA_1 and GA_2;

   (f) For each atomic condition of Type 2 ($v = c$) in E_i, add v into S.

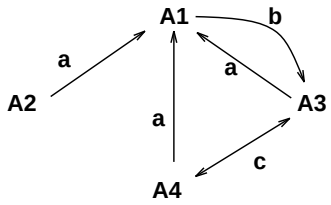
   (g) Compute the transitive closure on S based on Type 2 atomic conditions and key constraints in E_i(see Step (c)).

   (h) If GA_1+ is in S, proceed. Otherwise return NO and stop.

5. Return YES and stop.

.....

The idea of TestFD is explained as follows. Step 1 and 2 first discard all non-equality conditions in the join conditions and semantic constraints. The rest is best illustrated by Figure 5. Assume that the conditions and constraints $\{a : A_1 = 25;\ b : A_1 \longrightarrow A_3;\ c : A_3 = A_4\}$ are satisfied in the join result. Then, since A_1 is a constant in the join result, every column functionally determines A_1. These functional dependencies are represented by the directed arcs marked by a in Figure 5. Furthermore, since A_3 equals to A_4, they functionally determine one another. This is illustrated by a bi-directed arc marked by c in Figure 5. $A_1 \longrightarrow A_3$ is also shown as a directed arc marked by b



Known conditions and constraints: $a : A_1 = 25$; $b : A_1 \longrightarrow A_3$; $c : A_3 = A_4$
Conclusion: $A_2 \longrightarrow A_4$

Figure 5: Illustration of Algorithm TestFD

in the figure. Due to the transitive property of functional dependencies, we can draw the conclusion that $A_2 \longrightarrow A_4$. Therefore, in TestFD, if $A_i \longrightarrow A_j$ is to be tested, where A_i and A_j are some sets of columns, one can start up with a set containing A_i, then perform a transitive closure on the set until no new column is added. If A_j is in the final set, then $A_i \longrightarrow A_j$ is true. This is essentially what one iteration of Step 4 does: determining whether $FD_1 : (GA_1, GA_2) \longrightarrow GA_1+$ and $FD_2 : (GA_1+, GA_2) \longrightarrow RowID(R_2)$ are true. If each iteration of Step 4 returns true, then the whole condition C can imply that FD_1 and FD_2 hold in the join result.

Theorem 3 : *If the algorithm TestFD returns* YES, *FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C2](R_1 \times R_2)$.*

See [11] for the proof of this theorem.

Example 3 : Assume that we have three tables:

```
UserAccount(UserId, Machine, UserName)
PrinterAuth(UserId, Machine, PNo, Usage)
Printer(PNo, Speed, Make)
```

The UserAccount table stores information about user accounts. (UserId, Machine) is the primary key. The PrinterAuth table records which printers each user is authorized to use and his/her total usage of each printer. The primary key is (UserId, Machine, PNo). The Printer table maintains information about the speed and make of each printer. PNo is the primary key.

Consider the query: for each user on machine 'dragon', find the UserId, UserName, his/her total printer usage, and the maximum and minimum speeds of printers accessible to the user. This query can be expressed in SQL as

```
SELECT    U.UserId, U.UserName, SUM( A.Usage),
          MAX(P.Speed), MIN(P.Speed)
FROM      UserAccount U, PrinterAuth A, Printer P
WHERE     U.UserId = A.UserId and
```

```
          U.Machine = A.Machine and A.PNo = P.PNo
          and U.Machine = 'dragon'
GROUP BY  U.UserId, U.UserName
```

Because AA = (A.Usage, P.Speed) we partition the tables in the FROM clause into: $R_1 = (A, P)$ and $R_2 = (U)$. Consequently, $SGA_1 = GA_1 = \emptyset$, $SGA_2 = GA_2 = (U.UserId, U.UserName)$, $GA_1+ = (A.UserId, A.Machine)$, $GA_2+ = (U.UserId, U.Machine, U.UserName)$, $F =$ (SUM($A.Usage$), MAX($P.Speed$), MIN($P.Speed$)), $C_0 =$ '$U.UserId = A.UserId \wedge U.Machine = A.Machine$', C_1 = 'A.PNo = P.PNo', and C_2 = 'U.Machine = 'dragon' '. We now apply algorithm TestFD.

Step 1: $C \Longleftrightarrow U.UserId = A.UserId \wedge U.Machine = A.Machine \wedge A.PNo = P.PNo \wedge U.Machine =' dragon'$

Step 2-3: C remains unchanged.

Step 4: $E_1 \Longleftrightarrow U.UserId = A.UserId \wedge U.Machine = A.Machine \wedge A.PNo = P.PNo \wedge U.Machine =' dragon'$;

- Step a: $S = \{U.UserId, U.UserName\}$;
- Step b: Add $U.Machine$ to S due to $U.Machine = 'dragon'$, yielding $S = \{U.UserId, U.UserName, U.Machine\}$;
- Step c: The result after the transitive closure is: $S = \{A.UserId, A.Machine, U.UserName, U.Machine, U.UserId\}$;
- Step d: S contains the primary key ($U.Machine$, $U.UserId$) of table U (i.e. R_2);
- Step e: $S = \{U.UserId, U.UserName\}$;
- Step f: Add U.Machine to S due to $U.Machine =' dragon'$, yielding $S = \{U.UserId, U.UserName, U.Machine\}$;
- Step g: The result after the transitive closure is: $S = \{U.UserId, U.UserName, U.Machine, A.Machine, A.UserId\}$;
- Step h: S contains $GA_1+ = (A.Machine, A.UserId)$;

Step 5: Return YES and stop.

Therefore, the query can be evaluated as follows:

```
SELECT      UserId, UserName, TotUsage,
            MaxSpeed, MinSpeed
FROM        R'1 , R'2
WHERE       R'1.UserId = R'2.UserId and
            R'1.Machine = R'2.Machine
```

where

```
R'1 (UserId, Machine, TotUsage,
            MaxSpeed, MinSpeed) =
SELECT      A.UserId, A.Machine, SUM(A.Usage),
            MAX(P.Speed), MIN(P.Speed)
FROM        PrinterAuth A, Printer P
WHERE       A.PNo = P.PNo
GROUP BY    A.UserId, A.Machine
```

and

```
R'2 (UserId, Machine, UserName) =
SELECT      UserId, Machine, UserName
FROM        UserAccount U
WHERE       U.Machine = 'dragon'
```

The reader may have noticed that further optimization is possible. In particular, it is wasteful to perform the grouping for all users in PrinterAuth because we are only interested in those on machine dragon. Hence, we can add the predicate `A.Machine = 'dragon'` to the query computing R'_1. This type of optimization (predicate expansion) is routinely used but outside the scope of this paper. □

7 When is the transformation advantageous?

Example 4 : Figure 6 shows two access plans for a query. The two input tables, A and B, consist of 10000 and 100 rows, respectively. In Plan 1, the (10000×100) join yields only 50 rows, which are then grouped into 10 groups. In Plan 2, we first group the 10000 rows of A into 9000 groups and then perform a (9000 × 100) join. The input cardinalities of the join have not changed significantly but the input cardinality of the group-by operation increased from 50 to 9000. Most likely, Plan 2 is more expensive than Plan 1. □

This example may be somewhat contrived but it shows that the transformation does not always produce a better access plan. Ultimately, the choice is determined by the estimated cost of the two plans. However, we have some observation regarding the effect of the transformation:

- It cannot increase the input cardinality of the join.
- It may increase or decrease the input cardinality of the group-by operation. This depends on the selectivity of the join.
- It restricts the choice of join orders. We first have to perform all joins required to create R1 so we can perform the grouping. However, the join order of R1 with members of R2 is not restricted.
- In a distributed database, it may reduce the communication cost. Instead of transferring all of R1

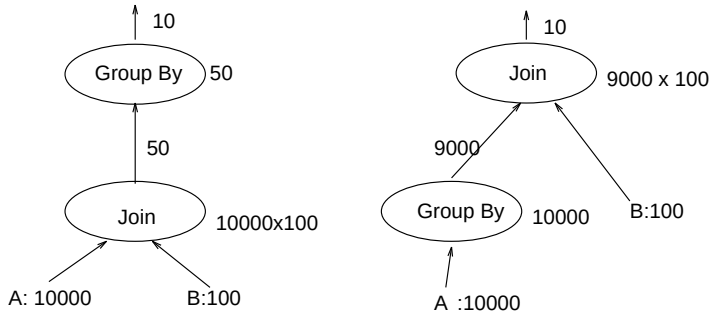


Figure 6: Is Plan 2 better than Plan 1?

to some other site to be joined with R2, we transfer only one row for each group of R2. Since communication costs often dominate the query processing cost, this may reduce the overall cost significantly.

- After the grouping and aggregation operation, the resulting table is normally sorted based on the grouping columns in most of the existing implementation of database systems. This fact can be exploited to reduce the cost of subsequent joins.

8 Performing join before group-by

Consider a query that involves one or more joins and where one of the tables mentioned in the from-clause is in fact an aggregated view. An aggregated view is a view obtained by aggregation on a grouped view. In a straightforward implementation, the aggregated view would first be materialized and the result then joined with other tables in the from-clause. In other words, group-by is performed before join. However, it may be possible (and beneficial) to reverse the order and first perform the joins and then the group-by. The theorems and algorithms developed in this paper allow us to determine whether the order can be reversed.

Example 5 : Assuming we have the same tables in Section 6, consider the same query: for each user on machine 'dragon', find the UserId, UserName, his/her total printer usage, and the maximum and minimum speeds of printers accessible to the user. In addition, we assume that there exists an aggregated view:

```
CREATE VIEW   UserInfo (UserId, Machine,
              TotUsage, MaxSpeed, MinSpeed)
AS SELECT     A.UserId, A.Machine, SUM(A.Usage),
              MAX(P.Speed), MIN(P.Speed)
FROM          PrinterAuth A, Printer P
WHERE         A.PNo = P.PNo
GROUP BY      A.UserId, A.Machine
```

on table `PrinterAuth` and `Printer`, which, for each user, lists the UserId, Machine, his/her total printer usage, and the maximum and minimum speeds of printers accessible to the user. Therefore, the query can be written as:

```
SELECT   UserId, UserName, TotUsage,
         MaxSpeed, MinSpeed
FROM     UserInfo I, UserAccount U
WHERE    I.UserId = U.UserId AND
         I.Machine = U.Machine AND
         U.Machine = "dragon"
```

The standard evaluation process for this query is to first materialize the view `UserInfo` by the join and aggregation and then join it with the `UserAccount` table. Using TestFD as we did in Section 6, we know that this query is equivalent to:

```
SELECT   U.UserId, U.UserName, SUM(A.Usage),
         MAX(P.Speed), MIN(P.Speed)
FROM     UserAccount U, PrinterAuth A, Printer P
WHERE    U.UserId = A.UserId and
         U.Machine = A.Machine
         and A.PNo = P.PNo and
         U.Machine = 'dragon'
GROUP BY U.UserId, U.UserName
```

Thus, the optimizer has two choices to consider for the query. It is possible that in the latter query expression, the number of rows resulting from the 3-table join is much smaller than the number of rows resulting from the 2-table join in the aggregated view. If this is the case, then the grouping operation will operate on a much smaller input in the latter query than in the former query, and the latter query can be better than the former query. Therefore, the reverse transformation can be beneficial.

9 Concluding remarks

We proposed a new strategy for processing SQL queries containing group-by, namely, pushing the group-by operation past one or more joins. This transformation may result in significant savings in query processing time. We derived conditions for deciding whether the transformation is valid and showed that they are both necessary and sufficient. The conditions were also shown to be sufficient for the more general transformation specified in Theorem 2. Because testing the full conditions may be expensive or even impossible, a fast algorithm was designed that tests a simpler, sufficient condition. The reverse of the transformation is also shown to be possible.

All queries considered in this paper were assumed not to contain a HAVING clause. Further work includes relaxing those conditions and finding necessary and sufficient conditions for the transformation specified in Theorem 2. Another important issue under study is how to partition all tables for a query into two sets of tables, R_1 and R_2, with R_1 containing aggregation columns and R_2 not. Some queries may not be transformable because: (a) no partitioning is possible, i.e., all tables contain some aggregation columns; or (b) it can be somehow partitioned but the testing algorithm returns NO. Column substitution can be used to improve the chance of a query being tested transformable. First, column substitution can be employed to obtain a set of equivalent queries. Based on this set, all possible partitions of the tables can be performed and the resulting queries can all be tested. This technique not only increases the chance of a query being tested transformable, but also provides the optimizer more choices of execution plans for a query. In addition, we are investigating algorithms for performing grouping and how to detect when the group-by operation can be pipelined with other operations [6, 2].

References

[1] C. J. Date and Hugh Darwen. *A Guide to the SQL Standard: a user's guide.* Addison-Wesley, Reading, Massachusetts, third edition, 1993.

[2] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 197–208, Brighton, England, August 1987. IEEE Computer Society Press.

[3] Richard A. Ganski and Harry K. T. Wong. Optimization of nested queries revisited. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 23–33, San Francisco, California, May 1987.

[4] Werner Kiessling. On semantic reefs and efficient processing of correlation queries with aggregates. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 241–249, Stockholm, Sweden, August 1985. IEEE Computer Society Press.

[5] Won Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[6] A. Klug. Access paths in the "abe" statistical query facility. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 161–173, Orlendo, Fla., June 2-4 1982.

[7] ISO/IEC SQL 92. *Information Technology - Database languages - SQL.* Reference number ISO/IEC 9075:1992(E), November 1992.

[8] Jim Melton and Alan R. Simon. *Understanding the new SQL: A Complete Guide.* Morgan Kaufmann, 1993.

[9] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 17(3):513–534, September 1991.

[10] Günter von Bültzingsloewen. Translating and optimizing SQL queries having aggregates. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 235–243, Brighton, England, August 1987. IEEE Computer Society Press.

[11] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. Technical Report CS 93-46, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, October 1993.