Performing group-by before join [query processing]

Conference Paper · March 1994

DOI: 10.1109/ICDE.1994.283001 · Source: IEEE Xplore

2 authors, including:

Per-Åke Larson
University of Waterloo
180 PUBLICATIONS **5,178** CITATIONS

Performing Group-By before Join

Weipeng P. Yan and Per-Åke Larson *

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

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 SQL2, 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.

*Authors' email addresses: {pwyan,palarson}@bluebox.uwaterloo.ca

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

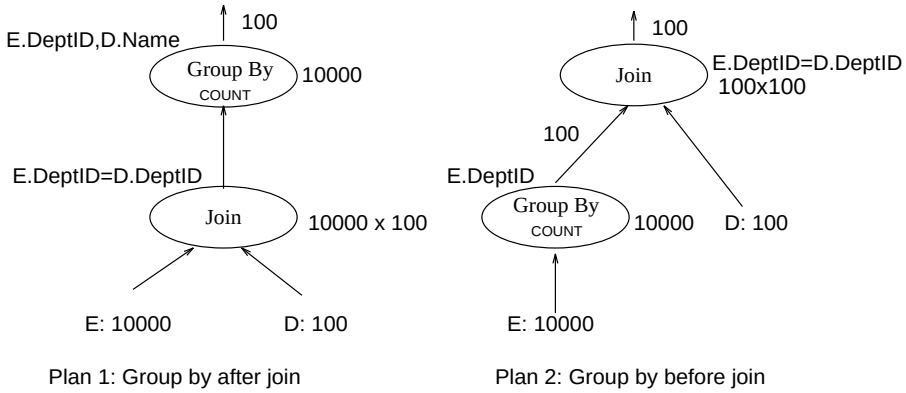


Figure 1: Two access plans for Example 1

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 4 presents the formalism that our results are based on. Section 3 defines the class of queries that we consider. 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 that 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[9] 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[3] 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 ([8, 7, 5, 12, 11]) 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[6], 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 aggregation functions and/or arithmetic aggregation 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.

- $\pi_d[B]R$, 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.

[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.

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[6, 10, 2] 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 val-

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

ues 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 NULLs 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

[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.

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 NULLs. In other words, SQL2 considers "NULL equal to NULL" when determining duplicates.

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.

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

4.3 Functional Dependencies

SQL2 [6] 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 interest. 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.

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

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 the "`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.

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

5 Theorems and Proofs

Theorem 1 (Main Theorem): *The expressions*

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

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, $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$*:*

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

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 $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ 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 "$(GA_1+, GA_2) \longrightarrow \texttt{RowID}(R_2)$ 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.

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.

Proof: The only difference between E_2 and E_2' is the change from $\pi_A[GA_2+]$ $\sigma[C_2]R_2$ to $\sigma[C_2]R_2$. Since the columns in R_2 other than GA_2+ do not participate in any of the operations in the expressions and the final projection is on columns (GA_1, GA_2, FAA), this change does not affect the result of the expression. Consequently the two expressions are equivalent. □

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: 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 $t[GA_1+] \stackrel{n}{\neq} t'[GA_1+]$. Clearly, t and t' are produced from the join of two sets, $S_1 = \{t[\alpha(R_1)], t'[\alpha(R_1)]\} \subseteq \sigma[C_1]r_1$ and $S_2 = \{t[\alpha(R_2)], t'[\alpha(R_2)]\} \subseteq \sigma[C_2]r_2$. Note that $t[\alpha(R_1)]$ and $t'[\alpha(R_1)]$ must be two different rows whereas $t[\alpha(R_2)]$ and $t'[\alpha(R_2)]$ might be the same row.

Consider $E_1(r_1, r_2)$ first. Since $t[GA_1, GA_2] \stackrel{n}{=} t'[GA_1, GA_2]$, t and t' will be grouped into the same group in $\mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$. All rows sharing the same value $t[GA_1, GA_2]$ in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ will be grouped into this group. In $E_1(r_1, r_2)$, there is therefore exactly one row whose value for columns $[GA_1, GA_2]$ is $t[GA_1, GA_2]$.

Now consider $E_2'(r_1, r_2)$. Since $t[GA_1+] \stackrel{n}{\neq} t'[GA_1+]$, $t[\alpha(R_1)]$ and $t'[\alpha(R_1)]$ will be grouped into two different groups in $\mathcal{G}[GA_1+]\sigma[C_1]r_1$. Denote these groups as g_1 and g_2 respectively. Therefore, $F[AA]\pi_A[GA_1+, AA]$ $\mathcal{G}[GA_1+]\sigma[C_1]r_1$ must contain the following two rows: $F[AA]\pi_A[GA_1+, AA]g_1$ and $F[AA]\pi_A[GA_1+, AA]g_2$, whose values for columns GA_1+ are $t[GA_1+]$ and $t'[GA_1+]$, respectively. Since t and t' are in the join result $\sigma[C_1 \wedge C_0 \wedge C_2]$

$(r_1 \times r_2)$, and $GA1+$ are the only columns of R_1 participating in the join, it follows that $(F[AA]\pi_A[GA_1+, AA]g_1) \circ t[\alpha(R_2)]$ and $(F[AA]\pi_A[GA_1+, AA]g_2) \circ t'[\alpha(R_2)]$ must be in the join result $\sigma[C_0](F[AA]\pi_A[GA_1+, AA]\ \mathcal{G}[GA_1+]\ \sigma[C_1]r_1 \times \sigma[C_2]r_2)$. Therefore, there are (at least) two rows, in $E_2'(r_1, r_2)$, with the same value $(t[GA1, GA2])$ for columns $[GA1, GA2]$. Since there is only one row in $E_1(r_1, r_2)$ with the value $(t[GA1, GA2])$ for columns $[GA1, GA2]$, $E_1(r_1, r_2)$ and $E_2(r_1, r_2)'$ cannot be equivalent. This proves the lemma. □

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: 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 $t[GA_1+, GA_2] \stackrel{\text{n}}{=} t'[GA_1+, GA_2]$ but $t[\alpha(R_2)] \stackrel{\text{n}}{\neq} t'[\alpha(R_2)]$. Clearly, t and t' are produced from the join of two sets, $S_1 = \{t[\alpha(R_1)], t'[\alpha(R_1)]\} \subseteq \sigma[C_1]r_1$ and $S_2 = \{t[\alpha(R_2)], t'[\alpha(R_2)]\} \subseteq \sigma[C_2]r_2$. Note that $t[\alpha(R_1)]$ and $t'[\alpha(R_1)]$ can be the same row but $t[\alpha(R_2)]$ and $t'[\alpha(R_2)]$ must be different rows.

First consider $E_1(r_1, r_2)$. Since $t[GA_1, GA_2] \stackrel{\text{n}}{=} t'[GA_1, GA_2]$, $t[\alpha(R_1)]$ and $t'[\alpha(R_1)]$ will be grouped into the same group in $\mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$. All rows sharing the same value $t[GA_1, GA_2]$ in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ will be grouped into this group. In $E_1(r_1, r_2)$ there is therefore exactly one row whose value for columns $[GA_1, GA_2]$ is $t[GA_1, GA_2]$.

Now consider $E_2'(r_1, r_2)$. Since $t[GA_1+] \stackrel{\text{n}}{=} t'[GA_1+]$, t and t' will be grouped into the same group in $\mathcal{G}[GA_1+]\sigma[C_1]r_1$. Therefore, there is exactly one row in $F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1$ having the value $t[GA_1+]$ for columns GA_1+. Denote this row by t_1. Since t and t' are in the join result $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$, and GA_1+ are the only columns of R_1 participating in the join, $t_1 \circ t[\alpha(R_2)]$ and $t_1 \circ t'[\alpha(R_2)]$ are in the join result $\sigma[C_0](F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1 \times \sigma[C_2]r_2)$. Therefore, there are (at least) two rows, $\pi_A[GA_1, GA_2, FAA]\ (t_1 \circ t[\alpha(R_2)])$ and $\pi_A[GA_1, GA_2, FAA]\ (t_1 \circ t'[\alpha(R_2)])$ in $E_2'(r_1, r_2)$, having the value $t[GA_1, GA_2]$ for columns $[GA_1, GA_2]$. Since there is only one row in $E_1(r_1, r_2)$ having the value $t[GA_1, GA_2]$ for columns $[GA_1, GA_2]$, $E_1(r_1, r_2)$ and $E_2'(r_1, r_2)$ cannot be equivalent. This proves the lemma. □

Lemma 2 and Lemma 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.*

Proof: Clearly, (GA_1, GA_2) is the key of the derived table resulting from applying E_1 to valid instances r_1 and r_2 of R_1 and R_2 respectively. Therefore there are no duplicate rows in E_1. □

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: 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 two cases to consider.

Case 1: Assume that $t_{21}[GA_1+] \stackrel{n}{\neq} t_{21}'[GA_1+]$. Clearly, $t_{21}[GA_1] \stackrel{n}{=} t_{21}'[GA_1]$ and $t_{22}[GA_2] \stackrel{n}{=} t_{22}'[GA_2]$. Since FD_1 holds in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$, (GA_1, GA_2) functionally determines GA_1+ in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$. Consider the grouping and aggregation in $F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1$, these operations only merge several rows with the same value for columns GA_1+ in $\sigma[C_1]r_1$ into one row, consequently the number of rows cannot increase and there is no new value for columns GA_1+ in all resulting rows. It follows that (GA_1, GA_2) must still functionally determine GA_1+ in $\sigma[C_0](F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1 \times \sigma[C_2]r_2)$. Since $t_1[GA_1, GA_2] \stackrel{n}{=} t_1'[GA_1, GA_2]$, $t_1[GA_1+] \stackrel{n}{=} t_1'[GA_1+]$ must hold. Therefore, $t_{21}[GA_1+] \stackrel{n}{=} t_{21}'[GA_1+]$, which is a contradiction.

Case 2: Assume that $t_{21}[GA_1+] \stackrel{n}{=} t_{21}'[GA_1+]$. Since the grouping in $\mathcal{G}[GA_1+]\sigma[C_1]r_1$ is on GA_1+, t_{21} and t_{21}' must be the same row, which is denoted by T_1. Since FD_2 holds in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$, (GA_1+, GA_2) functionally determines $\texttt{RowID}(R_2)$ in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$. Similarly due to the reasons above, (GA_1, GA_2) must still functionally determine $\texttt{RowID}(R_2)$ in $\sigma[C_0](F[AA]\pi_A[GA_1+, AA]\mathcal{G}[GA_1+]\sigma[C_1]r_1 \times \sigma[C_2]r_2)$. Since $t_1[GA_1, GA_2] \stackrel{n}{=} t_1'[GA_1, GA_2]$, t_{22} and t_{22}' must be the same row, which is denoted as T_2. The join between T_1 and T_2 can only generate one row. Therefore, t_1 and t_1' are the same row. Hence t and t' must be the same row, a contradiction.

The two cases above are the only possible cases and they both lead to contradictions. This proves the lemma. □

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: 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.

Case 1: $t \in E_1(r_1, r_2) \Rightarrow t \in E_2'(r_1, r_2)$. Consider a row $t \in E_1(r_1, r_2)$. There exists a group $g \in \mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ such that $t = F[AA]\pi_A[GA_1, GA_2, AA]g$. Since $(GA_1, GA_2) \longrightarrow \texttt{RowID}(R_2)$ (follows from FD_1 and FD_2) in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$, there is exactly one row $t_2 \in \sigma[C_2]r_2$ which joins with a subset g_1 of rows in $\sigma[C_1]r_1$ to form g. We can therefore write $g \equiv g_1 \times t_2$. Clearly, every row $t_p \in g_1$ has the property that $t_p[GA_1] \stackrel{\text{n}}{=} t[GA_1]$ and $C_0(t_p, t_2)$ is true. Furthermore, all rows in g_1 have the same values for columns GA_1+ because $(GA_1, GA_2) \longrightarrow (GA_1+)$ holds in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$. Therefore, for every row $t_o \in \sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$, if $t_o[GA_1+] \stackrel{\text{n}}{=} t[GA_1+]$, then $t_o[GA_1] \stackrel{\text{n}}{=} t[GA_1]$, and consequently $t_o \in g$, and $t_o[\alpha(R_1)] \in g_1$.

Now consider $E_2'(r_1, r_2)$. Clearly, there must exist a group $g_1' \in \mathcal{G}[GA_1+]$ $\sigma[C_1]r_1$ containing all rows in $\sigma[C_1]r_1$ having the value $t[GA_1+]$ for columns GA_1+. Therefore, $g_1 \subseteq g_1'$. The rows in g_1 and g_1' all have the same value for columns GA_1+ but may differ on other columns. In the same way as above, every row $t_q \in g_1'$ has the property that $t_q[GA_1] \stackrel{\text{n}}{=} t[GA_1]$ and $C_0(t_q, t_2)$ is true. (Recall that GA_1+ are the only columns of R_1 involved in C_0.) Consequently, g_1' consist of exactly those rows in $\sigma[C_1]r_1$ that satisfy C_0 when concatenate with t_2 and therefore $g_1 = g_1'$.

Therefore, the row $t' \equiv \pi_A[GA_1, GA_2, FAA]\sigma[C_0](F[AA]\pi_A[GA_1+, AA]g_1 \circ t_2)$ must then exist in $E_2'(r_1, r_2)$ and, since $g_1 = g_1'$, $t \stackrel{\text{n}}{=} t_1'$. In other words, $t \in E_2'(r_1, r_2)$.

Case 2: $t \in E_2'(r_1, r_2) \Rightarrow t \in E_1(r_1, r_2)$. Consider a row $t \in E_2'(r_1, r_2)$. There must exist a group $g_1 \in \mathcal{G}[GA_1+]\sigma[C_1]r_1$ such that $t \equiv (\pi_A[GA_1, GA_2, FAA]\sigma[C_0](F[AA]\pi_A[GA_1+, AA]g_1) \circ t_2)$. for some $t_2 \in r_2$. For every row $t_1 \in g_1$, $C_0(t_1, t_2)$ is true and consequently $(t_1 \circ t_2) \in \sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$. Since all such $(t_1 \circ t_2)$ rows have the same value of (GA_1, GA_2), they all belong to the same group $g \in \mathcal{G}[GA_1, GA_2]\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$. From the fact that $(GA_1, GA_2) \longrightarrow \texttt{RowID}(R_2)$ holds in $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$, it follows that there exists exactly one row in $\sigma[C_2]r_2$ that join with some set of rows in $\sigma[C_1]r_1$ to form g. Clearly this row must be t_2. In other words, there exists a subset $g_1' \subseteq \sigma[C_1]r_1$ such that $g = g_1' \times t_2$.

Now $g_1 \subseteq g_1'$ because

(a) for any row $t_p \in \sigma[C_1]r_1$, if $t_p[GA_1] \stackrel{\text{n}}{=} t[GA_1]$ and $C_0(t_p, t_2)$ is true, then t_p must be in g_1';

(b) if a row $t_p \in g_1$, then $t_p[GA_1] \stackrel{n}{=} t[GA_1]$ and $C_0(t_p, t_2)$ is true.

Since $(GA_1, GA_2) \longrightarrow (GA_1+)$, all rows in g_1' have the same value for columns (GA_1+). Therefore, the rows in g_1 and g_1' all have the same value for columns GA_1+ but may differ on other columns. Since g_1 contains all rows in $\sigma[C_1]r_1$ having the value $t[GA_1+]$ for columns (GA_1+), the rows in g_1' must all be in g_1. In other words, $g_1' \subseteq g_1$. Therefore $g_1 = g_1'$. It follows that the row $t' \equiv \pi_A[GA_1, GA_2, FAA]\ ((F[AA]\pi_A[GA_1+, AA]g_1') \circ t_2) \in E_2'(r_1, r_2)$. Since $g_1 = g_1'$, $t = t'$. In other words, $t \in E_1$. □

Proof of the Main Theorem:

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 $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$ 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 $\sigma[C_1 \wedge C_0 \wedge C_2](r_1 \times r_2)$(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. Since GA_1+ is empty, GA_1 and C_0 must be empty. Consequently the join must be a Cartesian product. But GA_2 cannot be empty because in that case the grouping columns in query E_1 is empty and the query does not belong to the class of queries we consider. Therefore, E_1 and E_2' degenerate to:

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

and

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

Similarly, 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 $\sigma[C_1 \wedge C_2](R_1 \times R_2)$.

Since there is no grouping operation in E_2, $F[AA]\pi_A[AA]\sigma[C_1]R_1$ can yield only one row of result. Therefore its Cartesian product with R_2 produces $|\sigma[C_2]R_2|$ rows. If FD_2 holds in $\sigma[C_1 \wedge C_2](R_1 \times R_2)$, then $(GA_2) \longrightarrow$ `RowID`(R_2) in $\sigma[C_2]r_2$ because the join is a Cartesian product. Therefore, the grouping in E_1 is actually based on every row of R_2. Therefore, E_1 and E_2 are equivalent. If FD_2 does not hold in $\sigma[C_1 \wedge C_2](R_1 \times R_2)$, then there must exist an instance of table R_2 in which GA_2 is not unique. It follows that E_1 must produce a table with cardinality less than $|R_2|$, and E_2 must produce a

table with cardinality equal to $|R_2|$. Therefore E_1 and E_2 cannot be equivalent. Therefore, if and only if FD_2 holds in $\sigma[C_1 \wedge C_2](R_1 \times R_2)$, E_1 is equivalent to E_2. Consequently our Main Theorem holds when GA_1+ is empty.

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:

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

and

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

respectively, and FD_1 and FD_2 degenerate to $(GA_1) \longrightarrow GA_1$ and $(GA_1) \longrightarrow$ `RowID`(R_2) respectively.

FD_1 always holds. Therefore, we only need to determine whether FD_2 is a necessary and sufficient condition. Since the join is merely a Cartesian product, this condition means that $\sigma[C_2]r_2$ can contain no more than one row.

Clearly, if FD_2 holds in $\sigma[C_1 \wedge C_2](r_1 \times r_2)$, then E_1 and E_2 are equivalent. If FD_2 does not hold in $\sigma[C_1 \wedge C_2](r_1 \times r_2)$, that is, $\sigma[C_2]r_2$ contains more than one row, then, for every $t_1 \in \sigma[C_1]r_1$, E_2 must contain more rows with the value $t_1[GA_1]$ for columns GA_1 than E_1 does. Hence E_1 and E_2 cannot be equivalent. Consequently, our main theorem holds also 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.

Proof: If FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$, then,

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

and

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

are equivalent according to our main theorem. Therefore,

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

and

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

are equivalent because $\pi_A[Att]R_a = \pi_A[Att]R_b$ provided that $R_a = R_b$, where Att are some common columns of R_a and R_b. Therefore, when $d = A$, the two expressions in the theorem are equivalent. Also,

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

and

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

are equivalent because $\pi_D[Att]R_a = \pi_D[Att]R_b$ provided that $\pi_A[Att]R_a = \pi_A[Att]R_b$, where Att are some common columns of R_a and R_b. Therefore, when $d = D$, the two expressions in the theorem are also equivalent. This proves the theorem. □

6 Algorithms to Test the Conditions

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` [6] 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.

6.1 Constraints in SQL2

In SQL2[6, 10, 2], a users can specify several kinds of semantic integrity constraints on tables and columns. For our purpose, we classify SQL2 constraints into five classes: column constraints, domain constraints, key constraints, referential integrity constraints and assertion constraints. We will use the table specified in Figure 5 as an example as we briefly explain these constraints.

```
CREATE DOMAIN DepIdType SMALLINT
      CHECK VALUE > 0 AND VALUE < 100
CREATE TABLE Department (
      EmpID       INTEGER   CHECK (EmpID > 0),
      EmpSID      INTEGER UNIQUE,
      LastName    CHARACTER(30) NOT NULL,
      FirstName   CHARACTER(30),
      DeptID      DepIdType    CHECK (DeptID>5),
         PRIMARY KEY (EmpID),
         FOREIGN KEY (DeptID) REFERENCES Dept)
```

Figure 5: SQL constraints

Column constraints include `NOT NULL` and `CHECK` constraints. A column can be specified as `NOT NULL`. A `CHECK` constraint can also be added to a column of a table. In Figure 5, the statement `LastName CHARACTER(30) NOT NULL` specifies that `LastName` cannot be `NULL`, and the statement `EmpID INTEGER (CHECK EmpID > 0)` specifies that `EmpID` must be positive.

A domain constraint specifies a constraint on a domain, and all columns defined on the domain must satisfy the constraint. In Figure 5, the statement `CREATE DOMAIN DepIdType SMALLINT CHECK VALUE > 0 AND VALUE < 100` specifies the domain name `DepIdType` and its constraint. Then the statement `DeptID DepIdType` specifies that `DepID` should satisfy the constraint. Note that domain constraints are equivalent to column constraints on the appropriate columns.

Key constraints include primary key and candidate key constraints. Primary key and candidate keys are defined by the statement `PRIMARY KEY` and `UNIQUE` respectively in a base table definition. A primary key cannot contain `NULL`, whereas a candidate key may contain `NULL`. In Figure 5, the statement `PRIMARY KEY (EmpID)` specifies that `(EmpID)` is the primary key, and the statement `EmpSID INTEGER UNIQUE` specifies that `EmpSID` is a candidate key.

A referential integrity constraint is a foreign key constraint which specifies a constraint between two tables. A foreign key is a list of columns in one table whose values must either be `NULL` or match the values of some candi-

date key or primary key in some table(may be the same as the original table). In Figure 5, the statement `FOREIGN KEY (DeptID) REFERENCES Dept` is an example of a referential integrity constraint.

An assertion constraint specifies a restriction which possibly several columns in possibly several tables must satisfy. It is defined by the statement `CREATE ASSERTION` outside of the table definition.

Observe that all constraints must be satisfied in every valid instance of the database. We can therefore add these constraints into the `WHERE` clause of a query without changing the result of the query. Therefore these constraints must be satisfied in the join result. Because each primary/candidate key functionally determines all columns in a table, we can use the notation defined in Section 4.3 to represent these conditions as Boolean expressions. `NOT NULL` and `CHECK` constraints on a column can also be easily represented as Boolean expressions. Each domain constraint can be treated as a `CHECK` constraint on a column defined over the domain. Referential integrity and assertion constraints can also be expressed as Boolean expressions.

The detailed method to translate domain, column, referential integrity and assertion constraints into Boolean expressions is not the focus of this paper and will not be discussed further here. We use T_1 and T_2 to denote the Boolean expressions representing domain, column, referential integrity and assertion constraints in table R_1 and R_2, respectively. We use H to denote the set of host variables in a query predicate, $K_i(R)$ to denote the ith candidate(primary) key of table R, and $|R|$ to denote the cardinality of a table R. These symbols are summarized in Figure 6.

Symbol	Definitions
$K_i(R)$	The ith candidate(primary) key of table R
T_1	Column, domain, referential integrity and assertion constraints on table R_1
T_2	Column, domain, referential integrity and assertion constraints on table R_2
H	The set of host variables in a query predicate
$\|R\|$	the cardinality of a table R

Figure 6: Summary of Symbols

6.2 Using Semantic Constraints to Test the Conditions

There can be many ways to test the conditions FD_1 and FD_2. The semantic constraints in `SQL` we discuss in Section 6.1 can be used to determine whether

FD_1 and FD_2 are true.

Theorem 3 : *FD_1 and FD_2 hold in $(\sigma[C_1]R_1 \times \pi_A[GA_2+]\sigma[C_2]R_2)$ if Condition (A):*

$$\begin{aligned}
&\forall h \in H, \forall t, t' \in Domain(R_1 \times R_2) \\
&\quad \{\lfloor C_1(t,h) \wedge C_1(t',h) \wedge C_0(t,t',h) \wedge C_0(t,t',h) \wedge C_2(t,h) \wedge C_2(t',h) \\
&\qquad \wedge T_1(t,h) \wedge T_2(t',h) \rfloor \\
&\quad \wedge (\bigwedge_{\forall i} (t[K_i(R_1)] \stackrel{n}{=} t'[K_i(R_1)] \Rightarrow t[\alpha(R_1)] \stackrel{n}{=} t'[\alpha(R_1)])) \\
&\quad \wedge (\bigwedge_{\forall i} (t[K_i(R_2)] \stackrel{n}{=} t'[K_i(R_2)] \Rightarrow t[\alpha(R_2)] \stackrel{n}{=} t'[\alpha(R_2)]))\} \\
&\Rightarrow \{(t[GA1, GA2] \stackrel{n}{=} t'[GA1, GA2] \Rightarrow t[GA1+] \stackrel{n}{=} t'[GA1+])\}
\end{aligned}$$

and Condition (B):

$$\begin{aligned}
&\forall h \in H, \forall t, t' \in Domain(R_1 \times R_2) \\
&\quad \{\lfloor C_1(t,h) \wedge C_1(t',h) \wedge C_0(t,t',h) \wedge C_0(t,t',h) \wedge C_2(t,h) \wedge C_2(t',h) \\
&\qquad \wedge T_1(t,h) \wedge T_2(t',h) \rfloor \\
&\quad \wedge (\bigwedge_{\forall i} (t[K_i(R_1)] \stackrel{n}{=} t'[K_i(R_1)] \Rightarrow t[\alpha(R_1)] \stackrel{n}{=} t'[\alpha(R_1)])) \\
&\quad \wedge (\bigwedge_{\forall i} (t[K_i(R_2)] \stackrel{n}{=} t'[K_i(R_2)] \Rightarrow t[\alpha(R_2)] \stackrel{n}{=} t'[\alpha(R_2)]))\} \\
&\Rightarrow \{(t[GA1+, GA2] \stackrel{n}{=} t'[GA1+, GA2] \Rightarrow t[\texttt{RowID}(R2)] \stackrel{n}{=} t'[\texttt{RowID}(R2)])\}
\end{aligned}$$

hold.

Condition (A) and (B) correspond to FD_1 and FD_2 respectively. The consequences of Condition (A) and (B), $(t[GA1, GA2] \stackrel{n}{=} t'[GA1, GA2] \Rightarrow t[GA1+] \stackrel{n}{=} t'[GA1+])$ and $(t[GA1+, GA2] \stackrel{n}{=} t'[GA1+, GA2] \Rightarrow t[\texttt{RowID}(R2)] \stackrel{n}{=} t'[\texttt{RowID}(R2)])$, are actually FD_1 and FD_2 according to our definition on functional dependency. There are three parts in each of the antecedents of Condition (A) and (B). In the Cartesian product of R_1 and R_2, part one, $\lfloor C_1(t,h) \wedge C_1(t',h) \wedge C_0(t,t',h) \wedge C_0(t,t',h) \wedge C_2(t,h) \wedge C_2(t',h) \wedge T_1(t,h) \wedge T_2(t',h) \rfloor$, states that all host variables and rows satisfy the join condition $C_1 \wedge C_0 \wedge C_2$ and all the semantic constraints except the key constraints of table R_1 and R_2; part two and three, $(\bigwedge_{\forall i}(t[K_i(R_1)] \stackrel{n}{=} t'[K_i(R_1)] \Rightarrow t[\alpha(R_1)] \stackrel{n}{=} t'[\alpha(R_1)]))$ and $(\bigwedge_{\forall i}(t[K_i(R_2)] \stackrel{n}{=} t'[K_i(R_2)] \Rightarrow t[\alpha(R_2)] \stackrel{n}{=} t'[\alpha(R_2)]))$, state that all rows satisfy the key constraints of table R_1 and R_2. Therefore, the proof of this theorem is straightforward.

Proof: Assume that the conditions stated in the theorem hold. In the join result $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$, all semantic constraints (key constraints, T_1

and T_2) and all join conditions C_1, C_0, C_2 must be satisfied, that is, the antecedents of both conditions are true. Therefore the two consequents:

$$t[GA1+, GA2] \stackrel{n}{=} t'[GA1+, GA2] \Rightarrow t[\texttt{RowID}(R2)] \stackrel{n}{=} t'[\texttt{RowID}(R2)]$$

and

$$t[GA1, GA2] \stackrel{n}{=} t'[GA1, GA2] \Rightarrow t[GA1+] \stackrel{n}{=} t'[GA1+]$$

are both true. According to our definition of functional dependency, this means that FD_1 and FD_2 hold in $\sigma[C_1 \wedge C_0 \wedge C_2](R_1 \times R_2)$. □

If we can design an efficient algorithm to test the satisfiability of the conditions in Theorem 3, we can use it to determine the validity of the transformation. Note that, when the algorithm returns true, the transformation is valid, but when the algorithm returns false, the transformation is not necessary invalid.

An example of such an algorithm is the satisfiability algorithm in [1]. This algorithm can be used to test the satisfiability of a **restricted class** of Boolean expressions. Hence we can simplify the conditions stated in Theorem 3 into a stronger condition which contains only Boolean expressions belonging to the restricted class. If the simplification cannot be done, it immediately returns false. If it can be done then we apply that satisfiability algorithm to test the simplified condition and if the algorithm returns true, the transformation is valid.

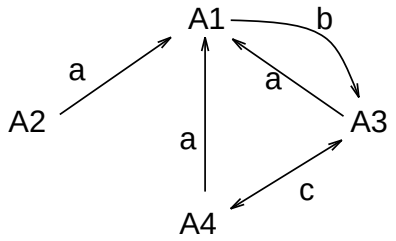
6.3 TestFD: A Fast Algorithm

In this section, we will present an efficient algorithm that handles 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. Therefore, we designed an algorithm that exploits only information about primary(candidate) keys and equality conditions in the `WHERE` clause, column and domain constraints. 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. 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`.



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 7: Illustration of Algorithm TestFD

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 7. 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 7. 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 7. $A_1 \longrightarrow A_3$ is also shown as a directed arc marked by b 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 $FD_2 : (GA_1+, GA_2) \longrightarrow \texttt{RowID}(R_2)$ 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 4 : *If the algorithm TestFD returns* `YES`, *FD_1 and FD_2 hold in* $\sigma[C_1 \wedge C_0 \wedge C2](R_1 \times R_2)$.

Proof: We prove the theorem by showing that Conditions (A) and (B) in Theorem 3 hold when algorithm TestFD returns YES. TestFD tests simpler and stronger conditions than Conditions (A) and (B). It drops the non-equality atomic conditions in C_1, C_0, C_2, T_1 and T_2 by deleting D_i's in C. This weakens the Boolean expression C and thus strengthens the whole conditions. Assuming that the algorithm TestFD returns YES, consider one iteration of Step 4. Since Step 4(d) for E_i returns true, the expression

$$\begin{aligned}
&\forall h \in H, \forall t, t' \in Domain(R_1 \times R_2) \\
&\quad \{\lfloor E_i(t,t') \rfloor \wedge (\bigwedge_{\forall i}(t[K_i(R_1)] \stackrel{n}{=} t'[K_i(R_1)] \Rightarrow t[\alpha(R_1)] \stackrel{n}{=} t'[\alpha(R_1)])) \\
&\quad \wedge (\bigwedge_{\forall i}(t[K_i(R_2)] \stackrel{n}{=} t'[K_i(R_2)] \Rightarrow t[\alpha(R_2)] \stackrel{n}{=} t'[\alpha(R_2)]))\} \\
&\Rightarrow \{t[GA_1+, GA_2] \stackrel{n}{=} t'[GA_1+, GA_2] \Rightarrow t[\texttt{RowID}(R_2)] \stackrel{n}{=} t'[\texttt{RowID}(R_2)]\}
\end{aligned}$$

is also true. Because Step(4) is true for all E_i, $i = i, ..., n$, we know that the above expression is true when E_i is replaced with C and thus Condition (B) is true. Condition (A) can be proved to be true in the same way. □

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: C remains unchanged.

Step 3: C is not empty so continue.

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 4: No more disjunctive components in C, go to Step 5;

Step 5: Return YES and stop.

Therefore, the query can be evaluated as follows:

```
    SELECT      UserId, UserName, TotUsage, MaxSpeed, MinSpeed
    FROM        R1', R2'
    WHERE       R1'.UserId = R2'.UserId and
                R1'.Machine = R2'.Machine
where
    R1' (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
    R2' (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 8 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. □

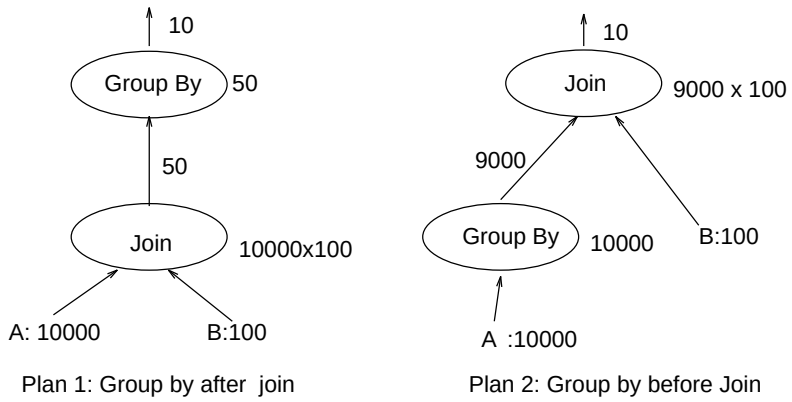


Figure 8: Is Plan 2 better than Plan 1?

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 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 6.3, 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 6.3, 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 [9, 3].

References

[1] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

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

[3] 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.

[4] Johann Christoph Freytag and Nathan Goodman. On the translation of relational queries into iterative programs. *ACM Transactions on Database Systems*, 14(1):1–27, March 1989.

[5] 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.

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

[7] 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.

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

[9] 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.

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

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

[12] 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.