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, 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
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, R1 and R2. Let R1 denote the table containing aggregation columns and R2 the table not containing any such columns.

The search conditions in the WHERE clause can be expressed as C1 \& C0 \& C2, where C1, C0, and C2 are in conjunctive normal form, C1 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 sub-queries are allowed.

The grouping columns mentioned in the \texttt{GROUP BY} clause may contain columns from $R_1$ and $R_2$, denoted by $GA_1$ and $GA_2$, respectively. According to SQL\texttt{2}[7], the selection columns in the \texttt{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 \texttt{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:

\begin{verbatim}
SELECT [ALL/DISTINCT] $SGA_1$, $SGA_2$, $F(AA)$ FROM $R_1$, $R_2$ WHERE $C_1 \land C_0 \land C_2$ GROUP BY $GA_1$, $GA_2$
\end{verbatim}

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 \texttt{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 \texttt{null} 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:

\begin{verbatim}
WHERE $C_0$
\end{verbatim}

\begin{verbatim}
WHERE $R_1(GA_1^+, F(AA))$
\end{verbatim}

\begin{verbatim}
SELECT ALL $GA_1^+, F(AA)$ FROM $R_1$
\end{verbatim}

where

or

\begin{verbatim}
WHERE $C_1$
\end{verbatim}

\begin{verbatim}
GROUP BY $GA_1^+$
\end{verbatim}

and

\begin{verbatim}
WHERE $R_2(GA_2^+) =
\end{verbatim}

\begin{verbatim}
SELECT ALL $GA_2^+$ FROM $R_2$
\end{verbatim}

In SQL\texttt{2}, $F(AA)$ transfers a group of rows into one single row, even when $F(AA)$ is empty. Therefore, throughout 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 \texttt{nulls} on comparisons, duplicate eliminations, and functional dependencies when using strict SQL\texttt{2} 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.

- $G[GA]$: Group table $R$ on grouping columns $GA = \{GA_1, GA_2, \ldots, GA_n\}$. This operation is defined by the query\footnote{Certainly, this query does more than \texttt{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.}

\begin{verbatim}
SELECT * FROM R ORDER BY GA.
\end{verbatim}

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 \texttt{SELECT * FROM R WHERE C}.

In this paper we will not consider the \texttt{EXCEPT} and \texttt{UNION} operations.
• \( \pi_d [B] R \), 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 \( \text{SELECT [ALL /DISTINCT] } E \text{ FROM } R \).

• \( F[AA] R \); \( F[AA] = (f_1(AA), f_2(AA), ..., f_n(AA)) \) 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 \( G \) of \( R \) and yields one value. An example of \( f_i(AA) \) is \( \text{COUNT}(A_1) + \text{SUM}(A_2 + A_3) \). Duplicates in the overall result are not eliminated. This operation is defined by the query \( \text{SELECT } G(A_1) \text{ FROM } R \text{ GROUP BY } G(A) \), where \( G(A) \) is the grouping columns of \( R \).

We also use \( \Rightarrow, \leftrightarrow, \land \) and \( \lor \) to represent logical implication, logical equivalence, logical conjunction and logical disjunction respectively. Then, the class of SQL queries we consider can be expressed as \(^3\):

\[
F[AA] \pi_d [SGA_1, SGA_2, AA] G [GA_1, GA_2] \\
\sigma[C_1 \land C_0 \land C_2 | (R_1 \times R_2)]
\]

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

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

where \( F AA \) 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 NULLs on duplicate operations is different. Duplicate operations include DISTINCT,

\[\begin{array}{|c|c|c|c|}
\hline
\text{AND} & \text{true} & \text{unknown} & \text{false} \\
\text{null} & \text{true} & \text{unknown} & \text{false} \\
\text{false} & \text{false} & \text{false} & \text{false} \\
\hline
\end{array}\]

\[\begin{array}{|c|c|c|c|}
\hline
\text{OR} & \text{true} & \text{unknown} & \text{false} \\
\text{null} & \text{true} & \text{true} & \text{false} \\
\text{false} & \text{true} & \text{true} & \text{false} \\
\text{false} & \text{false} & \text{false} & \text{false} \\
\hline
\end{array}\]

Figure 2: The semantics of AND and OR in SQL2

GROUP BY, \( \text{UNION} \), \( \text{EXCEPT} \) and \( \text{INTERSECT} \), 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. In other words, SQL2 considers "NULL equal to NULL" when determining duplicates.

Note that we do not include the \( \text{UNIQUE} \) predicate among the duplicate operations. SQL2 uses "NULL not equal to NULL" semantics when considering \( \text{UNIQUE} \).

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 \([P]\) and \([P]^\prime\) specified in Figure 3 for interpreting unknown to false and true respectively. In addition, a special equality operator, \( = \), 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-

\(^3\)There certainly exist other solutions to this problem. We just present the one we think is most appropriate for our purpose.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>P is a</td>
<td>P is true</td>
</tr>
<tr>
<td>predicate</td>
<td>P is unknown</td>
</tr>
<tr>
<td></td>
<td>P is false</td>
</tr>
<tr>
<td>P</td>
<td>true</td>
</tr>
<tr>
<td>P</td>
<td>unknown</td>
</tr>
<tr>
<td>P</td>
<td>false</td>
</tr>
<tr>
<td>X, Y are</td>
<td>X is NULL &amp; Y</td>
</tr>
<tr>
<td>variables</td>
<td>is NULL</td>
</tr>
<tr>
<td></td>
<td>Otherwise</td>
</tr>
</tbody>
</table>

**Figure 3:** The definition of interpretation operators

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 a column of R, we denote the value of $a$ in $t$ as $t[a]$.

**Definition 1:** (Row scheme) Consider a table scheme $R(\ldots, A, \ldots)$, where $A = \{a_1, a_2, \ldots, a_n\}$ is a set of columns and $B$ is a single column. Let $r$ be an instance of $R$. $A$ functionally determines $B$, denoted by $A \rightarrow B$, if the following condition holds:

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

which we also write as $t[A] \equiv t'[A]$.

**Definition 2:** (Functional Dependency) Consider a table scheme $R(A, B, \ldots)$, where $A = \{a_1, a_2, \ldots, a_n\}$ is a set of columns and $B$ is a single column. Let $r$ be an instance of $R$. $A$ functionally determines $B$, denoted by $A \rightarrow B$, in $r$ if the following condition holds:

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

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

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

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 "o" is also defined as the concatenation operator.

5 Theorems and proofs

**Theorem 1 (Main Theorem):** The expressions
De/initions

the concatenation of a group ed table $g$
the concatenation of two rows $A$ and $B$
shorthand for $A \circ B$

$\sigma_{g}$ notation /\( \sigma_{g} \)
The intuitive meaning of $\sigma_{g}$ ensures that each group in $G[GA_{1}, GA_{2}][\sigma(C_{1} \land C_{0} \land C_{2})(R_{1} \times R_{2})]$ (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 $G[GA_{1}][\sigma(C_{1})(R_{1})] \circ \sigma(C_{0})(R_{1} \times R_{2})$ (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 $F[AA][\pi_{A}[GA_{1}+, AA][G[GA_{1}+][\sigma(C_{1})]R_{1}] \circ \sigma(C_{2})R_{2}$ (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 $G[GA_{1}, GA_{2}][\sigma(C_{1} \land C_{0} \land C_{2})(R_{1} \times R_{2})]$ (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 $G[GA_{1}, GA_{2}][\sigma(C_{1} \land C_{0} \land C_{2})(R_{1} \times R_{2})]$ yields one row in the overall result of $E_{1}$, therefore, $E_{2}$ contains one row corresponding 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

\[ E_{1} : \pi_{A}[GA_{1}, GA_{2}, FAA][\sigma(C_{0})[F[AA][\pi_{A}[GA_{1}+, AA][G[GA_{1}+][\sigma(C_{1})]R_{1}]\times \sigma(C_{2})R_{2}] \]

is equivalent to $E_{2}$.

The difference between $E_{2}$ and $E_{3}$ is that $E_{3}$ 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 $E_{3}$ if and only if $FD_{1}$ and $FD_{2}$ hold in $\sigma(C_{1} \land C_{0} \land C_{2})(R_{1} \times R_{2})$. 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 $\sigma(C_{1} \land C_{0} \land C_{2})(R_{1} \times R_{2})$. 
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 $\sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$. 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 $t' \in \sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$ such that $t'[GA_1, GA_2] \neq t'[GA_1, GA_2]$ but $t[GA_1, GA_2] = t'[GA_1, GA_2]$. 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 $\sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$.

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 $\sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$. 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 $t' \in \sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$ such that $t[GA_1, GA_2] \neq t'[GA_1, GA_2]$ but $t[GA_1, GA_2] = t'[GA_1, GA_2]$. 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 $\sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$ 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.


Lemma 5: If $FD_1$ and $FD_2$ hold in $\sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$, 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 $\sigma[C_1 \land C_0 \land C_2](r_1 \times r_2)$, but there exist two different rows $t, t' \in E'_2(r_1, r_2)$ which are duplicates of each other, that is, $t \equiv t'$. Then there must exist two rows, $t_1, t_2 \in \sigma[C_0](F[AA][GA_1^+, AA]G[GA_1^+]\sigma[C_1]r_1 \times \sigma[C_1]r_2)$ such that $t = t_1[GA_1, GA_2, FAA]$ and $t' = t_2[GA_1, GA_2, FAA]$. $t_1$ and $t_2$ must be produced by the join between rows in $F[AA]GA_1[GA_1^+, AA]G[GA_1^+]\sigma[C_1]r_1$ and $\sigma[C_2]r_2$. Assume $t_1 = t_1[GA_1^+] \equiv t_2[GA_1^+]$, which can be shown to lead to $t_1[GA_1^+] = t_2[GA_1^+]$, which is a contradiction. Case 2 is when $t_2[GA_1^+] \equiv t_1[GA_1^+]$, 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 $\sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$, 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 $G[GA_1, GA_2]\sigma[C_1 \land C_0 \land C_2](r_1 \times r_2)$ 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 $G[GA_1^+]\sigma[C_1]r_1$. 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 $\sigma[C_2 \land C_0 \land C_1](r_1 \times r_2)$ 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 $\sigma[C_2 \land C_0 \land C_1](r_1 \times r_2)$ (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. Then, it can be proved that $E_1$ and $E_2$ degenerate to:

$$E_1 : F[AA] \pi_A [GA_2, AA] \sigma [GA_2] \sigma [C_1 \land C_2](R_1 \times R_2)$$

and

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

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 $\sigma [C_1 \land C_2](R_1 \times R_2)$. 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:

$$E_1 : F[AA] \pi_A [GA_1, AA] \sigma [GA_1] \sigma [C_1 \land C_2](R_1 \times R_2)$$

and

$$E_2 : \pi_A [GA_1, F.AA] \sigma [C_0](F[AA] \pi_A [GA_1, AA] \sigma [C_1] R_1 \times \sigma [C_2] R_2).$$

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. \hfill \Box

Theorem 2 : Consider the following two expressions:

$$F[AA] \pi_d [SGA_1, SGA_2, AA] \sigma [GA_1, GA_2] \sigma [C_1 \land C_0 \land C_2](R_1 \times R_2)$$

and

$$\pi_d [SGA_1, SGA_2, F.AA] \sigma [C_0](F[AA] \pi_A [GA_1+, AA] \sigma [C_1] R_1 \times \pi_A [GA_2+], \sigma [C_2] R_2).$$

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

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 $\sigma [C_1 \land C_0 \land C_2](R_1 \times R_2)$, 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 $(v_1 = v_2)$, where $v_1, v_2, v$ 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:

\begin{algorithm}
\textbf{Algorithm TestFD: determine whether group-by can be performed before join.}
\begin{itemize}
  \item \textbf{Input}: Predicates $C_1, C_0, C_2, T_1, T_2$; key constraints of $R_1$ and $R_2$.
  \item \textbf{Output}: YES or NO.
\end{itemize}
\end{algorithm}

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

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: $C = E_1 \lor E_2 \lor \ldots \lor E_n$.

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 $v_1 = v_2 \in C$ such that $v_1 \in S$ and $v_2 \notin S$) or $(\exists$ key $(R_1) \in S$ and $v_2 \in R_1$ and $v_2 \notin S$) or $(\exists$ key $(R_2) \in S$ and $v_2 \in R_2$ and $v_2 \notin S$), add $v_2$ 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 $\{a: A_1 = 25; b: A_1 \rightarrow A_3; c: A_3 = A_4\}$ 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. $A_1 \rightarrow A_3$ 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 $A_2 \rightarrow A_4$. Therefore, in TestFD, if $A_i \rightarrow A_j$ 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 $A_i \rightarrow A_j$ is true. This is essentially what one iteration of Step 4 does: determining whether $FD_1 : (GA_1, GA_2) \rightarrow GA_1$ and $FD_2 : (GA_1, GA_2) \rightarrow RowID(R_2)$ 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 $\sigma[C_1 \land C_0 \land C_2](R_1 \times R_2)$.


**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

```sql
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
```
Because \( AA = (A . Usage, P . Speed) \) we partition the tables in the FROM clause into: \( R_1 = (A, P) \) and \( R_2 = (U) \). Consequently, \( SGA_1 = GA_1 = 0, SGA_2 = GA_2 = (U . UserId, U . UserName), GA_1 + = (A . UserId, A . Machine), \)

\[

\( C_1 = 'A . PNo = P . PNo', \) and \( C_2 = 'U . Machine = 'dragon'. ' We now apply algorithm TestFD.

Step 1: \( C \iff \) (\( U . UserId = A . UserId \land U . Machine = A . Machine \land A . PNo = P . PNo \land U . Machine = 'dragon' \)).

Step 2-3: \( C \) remains unchanged.

Step 4: \( E_1 \iff \) (\( U . UserId = A . UserId \land U . Machine = A . Machine \land A . PNo = P . PNo \land U . Machine = 'dragon' \)).

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:


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:


Step h: \( S \) contains \( GA_1 + = (A . Machine, A . UserId) \).

Step 5: Return YES and stop.

Therefore, the query can be evaluated as follows:

\[
\text{SELECT} \quad \text{UserId, UserName, TotUsage, MaxSpeed, MinSpeed} \\
\text{FROM} \quad R_1 \cup R_2 \\
\text{WHERE} \quad R_1 . UserId = R_2 . UserId \land R_1 . Machine = R_2 . Machine \\
\text{where} \quad R_1 (UserId, Machine, TotUsage, MaxSpeed, MinSpeed) = \\
\text{SELECT} \quad A . UserId, A . Machine, SUM(A . Usage), MAX(P . Speed), MIN(P . Speed) \\
\text{FROM} \quad PrinterAuth A, Printer P \\
\text{WHERE} \quad A . PNo = P . PNo \\
\text{GROUP BY} \quad A . UserId, A . Machine \\
\text{and} \quad R_2 (UserId, Machine, UserName) = \\
\text{SELECT} \quad UserId, Machine, UserName \\
\text{FROM} \quad UserAccount U \\
\text{WHERE} \quad U . Machine = 'dragon' \]

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 \((10000 \times 100)\) 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 \((9000 \times 100)\) 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 \( R_1 \) so we can perform the grouping. However, the join order of \( R_1 \) with members of \( R_2 \) is not restricted.
- In a distributed database, it may reduce the communication cost. Instead of transferring all of \( R_1 \)
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


