Interchanging the Order of Grouping and Join [1]

Weipeng P. Yan and Per-Åke Larson [2]

Department of Computer Science

University of Waterloo

Waterloo, Ontario

Canada N2L 3G1

Abstract

Assume that we have an `SQL` *query containing joins, a GROUP-BY and possibly a HAVING predicate. 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. This may reduce the query processing cost by reducing the amount of data participating in joins. The reverse transformation, i.e., performing join before group-by, can also be beneficial because the join may greatly reduce the input to the group-by. We formally define the problem, adhering strictly to the semantics of* `SQL`*2, and prove necessary and sufficient conditions for deciding when the 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

1.1 Group-by Push Down

`SQL` queries containing joins and group-by are fairly common. The standard way of evalu-

[1]Technical Report CS 95-09, Department of Computer Science, University of Waterloo, Canada

[2]Authors' email addresses: {pwyan,palarson}@bluebox.uwaterloo.ca

ating 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.

The following example illustrates the basic idea. This and subsequent examples are all based on a subset of the TPC-D database[11]. The tables are defined in Appendix A.

Example 1 : *Find the number of orders for each customer. Output customer key, customer name and total number of orders.*

```
SELECT C_CUSTKEY, C_NAME,COUNT(O_CUSTKEY)
FROM   CUSTOMERS,  ORDERS
WHERE  C_CUSTKEY = O_CUSTKEY
GROUP  C_CUSTKEY, C_NAME
```

Plan 1 in Figure 1 illustrates the standard way of evaluating the query: fetch the rows in tables `CUSTOMER` and `ORDERS`, perform the join, and group the result by `C_CUSTKEY` and `C_NAME`, counting the number of rows in each group. Since there are 150K orders and 15K customers, the input to the join is 150K `ORDERS` rows and 15K `CUSTOMER` rows. The input to the group-by consists of 150K rows. Now consider plan 2 in Figure 1. We first group the `ORDERS` table on `O_CUSTKEY` and perform the `COUNT`, then join the resulting 15K rows to the 15K `CUSTOMER` rows. This reduces the join from $(150K \times 15K)$ to $(15K \times 15K)$. The input cardinality of the group-by remains the same. Normally, plan 2 would be faster than plan 1.

The reason why the group-by can be pushed down for this query is as follows. `C_NAME` is functionally determined by `C_CUSTKEY` so it can be dropped from the list of group-by columns. Since `C_CUSTKEY` is the key of the `CUSTOMER` table, each `O_CUSTKEY` matches at most one `C_CUSTKEY` value and thus joins with at most one `CUSTOMER` row. Therefore, if we first perform the group-by on `O_CUSTKEY` of the `ORDERS` table, every row of a group will join with the same `CUSTOMER` row. Consequently, we obtain the same result as for the the original query. □

The following example shows that we sometimes have several ways of pushing down a group-by operator.

Example 2 : *Find the total amount of orders for every supplier with a positive account balance and with total order value exceeding 7.3 millions. Output supplier key, order key, total price of the order and the total value of the items from the supplier.*

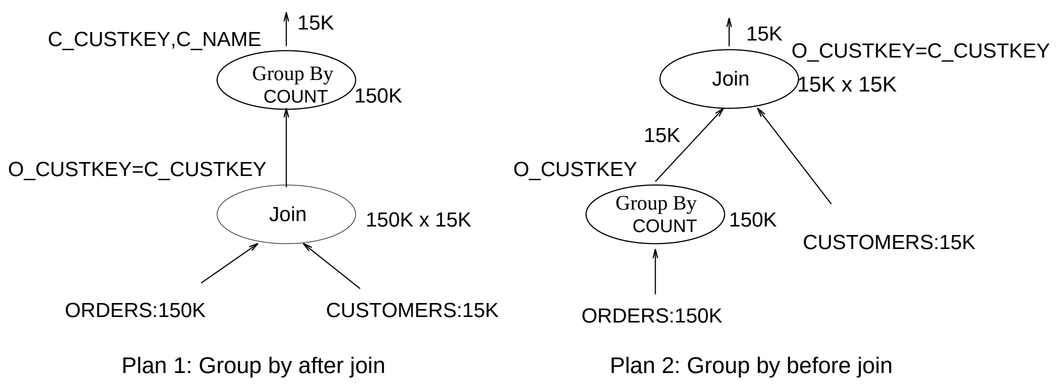


Figure 1: Two access plans for Example 1

```
SELECT S_SUPPKEY, L_ORDERKEY, O_TOTALPRICE, SUM(L_QUANTITY*L_EXTENDEDPRICE)
FROM SUPPLIERS, LINEITEM, ORDERS
WHERE S_SUPPKEY = L_SUPPKEY AND S_ACCTBAL > 0
      AND L_ORDERKEY= O_ORDERKEY
GROUP BY S_SUPPKEY, L_ORDERKEY, O_TOTALPRICE
HAVING SUM(L_QUANTITY*L_EXTENDEDPRICE) > 7300000;
```

We can push down the the tables `SUPPLIER` and `LINEITEM` and rewrite the query as

```
CREATE VIEW LINESUPP (V_ORDERKEY, V_SUPPKEY, V_TOTALVALUE) AS
(
  SELECT L_ORDERKEY, L_SUPPKEY, SUM(L_QUANTITY*L_EXTENDEDPRICE)
  FROM SUPPLIERS, LINEITEM
  WHERE S_SUPPKEY = L_SUPPKEY AND S_ACCTBAL > 0
  GROUP BY L_ORDERKEY, S_SUPPKEY
  HAVING SUM(L_QUANTITY*L_EXTENDEDPRICE)> 7300000
);

SELECT V_SUPPKEY, V_ORDERKEY, O_TOTALPRICE, V_TOTALVALUE
FROM LINESUPP, ORDERS
WHERE V_ORDERKEY= O_ORDERKEY;
```

Since each row in the view joins with at most one row from `ORDERS`, the transformation is intuitively correct. The `HAVING` predicate in the view is very selective, so it greatly reduces the number of rows output from the group-by. Therefore, the join between the view and ORDERS is very fast. In the original query we need to perform all the joins first, generating many rows that are later eliminated by the HAVING predicate. Much of the join effort is wasted. Therefore, it is very likely that the rewritten query can be evaluated faster.

It is also possible to push down only the LINEITEM table, and rewrite the query as

```
CREATE VIEW TPCD.LINEVALUE (V_ORDERKEY, V_SUPPKEY, V_TOTALVALUE) AS
(
  SELECT L_ORDERKEY, L_SUPPKEY, SUM(L_QUANTITY*L_EXTENDEDPRICE)
  FROM TPCD.LINEITEM
  GROUP BY L_ORDERKEY, L_SUPPKEY
  HAVING SUM(L_QUANTITY*L_EXTENDEDPRICE)> 7300000
);

SELECT V_SUPPKEY, V_ORDERKEY, O_TOTALPRICE, V_TOTALVALUE
FROM LINEVALUE, ORDERS, SUPPLIERS
WHERE S_SUPPKEY=V_SUPPKEY AND S_ACCTBAL > 0 AND V_ORDERKEY= O_ORDERKEY;
```

Since each row in the view joins with at most one row from the join of ORDERS and SUPPLIER, the transformation is intuitively correct. Again, the HAVING predicate in the view is very selective, so the number of groups output is small. There is an index on (L_ORDERKEY, L_SUPPKEY),see Appendix A, so no sort is needed to perform the group-by. Furthermore, since the second rewrite only groups on one table instead of the join of two tables, the second rewrite further reduces the join effort.

Database System	Tables Pushed Down	
	LINEITEM, SUPPLIER	LINEITEM
DB2/6000 V1	45%	65%
ORACLE V7	45%	67%

Figure 2: Reduction in Elapsed Time by Group-by Push-down for Example 2

Figure 2 shows the reduction in elapsed time obtained on two different database systems. In the above examples, it was both possible and beneficial to perform the group-by operation before the join. However, it is also easy to find examples where this is not possible, or possible but not beneficial.

1.2 Group-by Pull up

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 is first 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, as illustrated by the following example.

Example 3 : *For each supplier with an account balance greater than 9990, find the total value of items from this supplier among all orders.* Assume that the user writes this query using the aggregated view shown below.

```
CREATE VIEW SUPPVALUE (V_SUPPKEY, V_TOTALVALUE) AS
(
  SELECT L_SUPPKEY, SUM(L_QUANTITY*L_EXTENDEDPRICE)
  FROM LINEITEM
  GROUP BY L_SUPPKEY
);
```

```
   SELECT V_SUPPKEY, V_TOTALVALUE
   FROM SUPPVALUE, SUPPLIERS
   WHERE S_ACCTBAL > 9990 AND V_SUPPKEY = S_SUPPKEY;
```

The standard evaluation process for this query is to first materialize the view LINEITEM and then join it with the SUPPLIERS table. However, the group-by in the view can be pulled up and the query rewritten as

```
   SELECT L_SUPPKEY, SUM(L_QUANTITY*L_EXTENDEDPRICE)
   FROM LINEITEM, SUPPLIERS
   WHERE S_ACCTBAL > 9990 AND L_SUPPKEY = S_SUPPKEY
   GROUP BY L_SUPPKEY;
```

The reason why the transformation is correct is similar to the ones we stated for group-by push down in Section 1.1. For this query, the predicate `S_ACCTBAL > 9990` is very selective and it greatly reduces the number of rows participating in the join [3]. The rewritten query is very likely to perform better than the original one. This was confirmed by our experiments. Rewriting the query reduced the elapsed time by 99% on DB2/6000 V1 and by 98% on Oracle V7.

1.3 Outline of Paper

In Sections 1.1 and 1.2, we presented examples which show that interchanging the order of grouping and join can be beneficial. However, it is very easy to find examples where the transformation is (a) *not* possible or (b) possible but *not* beneficial. This raises the following general questions:

1. Exactly under what conditions is it possible to interchange the order of grouping and join?

2. Under what conditions does this transformation reduce the query processing cost? In other words, when should we perform group-by push down and group-by pull up.

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

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 5 introduces and proves the main theorem, which states necessary and sufficient conditions for performing the proposed transformation. Section 6 describes an efficient algorithm for deciding whether group-by can be interchanged with a join. In order to simplify the proofs, the first part of the paper does not allow queries with a `HAVING` clause. Section 7 considers the case when the `HAVING` clause is present and extends the theorems to handle this case. Section 8 proposes an efficient algorithm for partitioning the tables into pushed down tables and left-up tables. Section 9 illustrates that col-

[3] A typical database system would apply the local predicate first before the join.

umn substitution can be used to generate equivalent queries. Section 11 continues some observations about the trade-offs of the transformation. Section 12 summarizes the paper.

2 Related Work

We first proposed the idea of interchanging the order of group-by and join in [15]. However, we did not present proofs of the conditions and algorithms, and queries with `HAVING` were not considered. This paper presents the full proofs, extends the result to queries with a `HAVING` clause, and proposes an efficient algorithm for table partitioning.

The idea of interchanging the order of group-by and join can be generalized to partial grouping[14], which partially pushes a group-by past a join. In other words, for some queries which first perform a join and then group-by, we can perform a new group-by first, then the join, and finally the original group-by. The new "eager" group-by only partially groups the input so the original group-by is still needed after the join. Partial grouping reduces the input cardinality to the subsequent join and may reduce the overall processing time.

Chaudhuri and Shim independently generalized our idea of interchanging the order of group-by and join into a technique similar to partial grouping[2]. They also proposed a greedy conservative approach to modifying traditional System-R style optimizers to incorporate the transformation.

Klug[8] observed that in some cases, the result from a join is already grouped correctly. Nested-loop and sort merge joins, both have this property. In this case, explicit grouping is not needed and the join can be pipelined with aggregation. Dayal[4] stated, without proof, that the necessary condition for direct pipelining 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 [7, 6, 5, 13, 10] 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.* In `SQL2`, a value expression may include aggregation functions. We call a value expression that includes aggregation functions *an aggregation expression.* 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_d and R_u, where R_d denotes the table containing aggregation columns and R_u the table not containing any such columns.

The search condition in the `WHERE` clause can be expressed as $C_d \wedge C_0 \wedge C_u$, where C_d, C_0, and C_u are in conjunctive normal form, C_d only involves columns in R_d, C_u only involves columns in R_u, and each disjunctive component in C_0 involves columns from both R_d and R_u. Note that subqueries are allowed.

The *grouping columns* mentioned in the `GROUP BY` clause may contain columns from R_d and R_u, denoted by GA_d and GA_u, respectively. According to `SQL2`[12], the selection columns in the `SELECT` clause must be a subset of the grouping columns. We denote the selection columns as SGA_d and SGA_u, subsets of GA_d and GA_u, respectively. For the time being, we assume that the query does not contain a `HAVING` clause(relaxed in Section 7). The columns of R_d participating in the join and grouping is denoted by GA_d^+, and the columns of R_u participating in the join and grouping is denoted by GA_u^+.

In summary, we consider queries of the following form:

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

where:

GA_d:	grouping columns of table R_d;
GA_u:	grouping columns of table R_u; GA_d and GA_u cannot both be empty.
SGA_d:	selection columns, must be a subset of grouping columns GA_d;
SGA_u:	selection columns, must be a subset of grouping columns GA_u;
AA:	aggregation columns of table R_d;
C_d:	conjunctive predicates on columns of table R_d;
C_u:	conjunctive predicates on columns of table R_u;
C_0:	conjunctive predicates involving columns of both tables R_d and R_u, e.g., join predicates;
$\alpha(C_0)$:	columns involved in C_0;
F:	array of aggregation functions and/or aggregation expressions applied on AA (may be empty);
$F(AA)$:	application of aggregation functions and/or aggregation expressions F on aggregation columns AA;
GA_d^+:	$\equiv GA_d \cup \alpha(C_0) - R_u$, i.e., the columns of R_d participating in the join and grouping;
GA_u^+:	$\equiv GA_d \cup \alpha(C_0) - R_d$, i.e., the columns of R_u participating in the join and grouping

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

```
    SELECT [ALL/DISTINCT]      $SGA_d$, $SGA_u$, $FAA$
    FROM                       $R'_d$, $R'_u$
    WHERE                      $C_0$
```

where

```
    $R'_d(GA_d^+, FAA)$ ==
    SELECT ALL                 $GA_d^+, F(AA)$
    FROM                       $R_d$
    WHERE                      $C_d$
    GROUP BY                   $GA_d^+$
```

and

$R'_u(GA_u^+) ==$

SELECT ALL	GA_u^+
FROM	R_u
WHERE	C_u

Group-by generates one row per group[12], even when there are no aggregation columns (AA is empty) and no aggregation functions(F 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 effects of NULLs on comparisons, duplicate eliminations, and functional dependencies 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 [4] `SELECT * FROM R ORDER BY GA`. The result of this operation is a *grouped table.*

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

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

- $\pi_d[B]R$, where $d = A$ or D: Project table R on columns B, without eliminating duplicates when $d = A$ and with duplicate elimination when $d = D$. This operation is defined by the query `SELECT [ALL /DISTINCT] B FROM R`.

- $F[AA]R$: $F[AA] = (f_1(AA), f_2(AA), ..., f_n(AA))$, where $AA = \{A_1, A_2, ..., A_n\}$, and $F = \{f_1, f_2, ..., f_n\}$, AA are aggregation columns of grouped table R and F are aggregation expressions operating on AA. We must emphasize the requirement that table R is grouped by some grouping columns GA. All rows of table R must agree on the values of all columns except AA columns. For $i = 1, 2, ..., n$, f_i is an aggregation expression(which maybe just 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,A, F(AA) FROM R GROUP BY GA`, where GA is the grouping columns of R, and A is a set of columns that are functionally determined by GA (A may be empty). Note that this is not a syntactically valid `SQL2` statement since the columns A in the `SELECT` clause are not mentioned in the `GROUP BY` clause. However, since $GA \longrightarrow A$, from a query processing point of view, this is semantically sound.

We also use $\Rightarrow$, $\Longleftrightarrow$, $\wedge$ and $\vee$ to represent logical implication, logical equivalence, logical conjunction and logical disjunction respectively. The class of `SQL` queries we consider can be expressed as [5]:

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

where $d = A$ or $d = D$, and FAA are the aggregation columns after applying $F[AA]$ on each group. The last projection simply projects the rows on the columns wanted, and may eliminate duplicates. The last projection may be omitted when it is the same as the projection before the group-by operation $F[AA]$. Our objective is to determine under what conditions this expression is equivalent to

$$\begin{aligned}&\pi_d[SGA_d, SGA_u, FAA]\\&\quad\sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d]\sigma[C_d]R_d \times \pi_A[GA_u^+]\sigma[C_u]R_u).\end{aligned}$$

[5]In the case that there exists $f_i(A_i) \equiv$ `COUNT(*)` $\in F(AA)$, we can replace it with `COUNT`(A), where A is any column in GA_d, without changing the result of the query.

4.2 The Semantics of `NULL` in `SQL2`

`SQL2`[12, 9, 3] 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 3 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 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 3: The semantics of `AND` and `OR` in `SQL2`

ues is `NULL` or both values are `NULL`. A row qualifies only if the condition in the `WHERE` clause evaluates to `true`, that is, `unknown` is interpreted as `false`.

However, the effect of `NULL`s on duplicate operations is different. Duplicate operations include `DISTINCT`, `GROUP BY`, `UNION`, `EXCEPT` and `INTERSECT`, which all involve the detection of duplicate rows. Two rows are defined to be *duplicates* when every pair of corresponding column values are duplicates. Two column values are defined to be *duplicates* when they are equal and both not `NULL` or when they are both `NULL`s. In other words, `SQL2` adopts "`NULL` equal to `NULL`" semantics 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[6] 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 $\lceil P \rceil$ spec-

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

ified in Figure 4 for interpreting `unknown` to `false` and `true` respectively. These operators were first used in [10]. In addition, a special equality operator, $\overset{n}{=}$, 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
$X \overset{n}{=} Y$	`true`		$\lfloor X = Y \rfloor$

Figure 4: The definition of interpretation operators

4.3 Functional Dependencies

`SQL2` [12] provides facilities for defining (primary) keys of base tables. 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.

Example 4 *:* 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. 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 $\{a_1, a_2, ..., a_n\}$. Let r be an instance of R. Two rows t, $t' \in r$ are *equivalent with respect to A* if

$$\bigwedge_{i=1,\ldots,n} (t[a_i] \overset{\text{n}}{=} t'[a_i]),$$

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

Definition 2: (*Functional Dependency*) Consider a table $R(A, B, ...)$, where A 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:

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

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

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

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 $E(r_d, r_u)$ to denote the result generated by an `SQL` expression E when evaluated on instances r_d and r_u of tables R_d and R_u, respectively. We summarize all symbols defined in Section 4.2 and this section in Figure 5.

Symbol	Definitions
r_d, r_u	Instances of table R_d and R_u
$A \circ B$	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 a new grouped table. Each row in the new grouped table is the result of a row in g concatenated 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.
$E(r_d, r_u)$	the result from applying E on instances r_d and r_u.
`RowID`(R)	the `RowID` of table R

Figure 5: Summary of Symbols

5 Theorems and Proofs

Theorem 1 (Main Theorem): *The expressions*

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

and

$$E_2 : \pi_A[GA_d, GA_u, FAA] \\ \sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]R_d \times \pi_A[GA_u^+]\sigma[C_u]R_u)$$

are equivalent if and only if the following two functional dependencies hold in the join of R_d and R_u, $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$:

$$FD_1 : \; (GA_d, GA_u) \longrightarrow GA_d^+$$
$$FD_2 : \; (GA_d^+, GA_u) \longrightarrow \texttt{RowID}(R_u)$$

FD_2 means that for all valid instances r_d and r_u of R_d and R_u, respectively, if two different rows in $\sigma[C_d \wedge C_0 \wedge C_u]$ $(r_d \times r_u)$ have the same value for columns (GA_d^+, GA_u), then the two rows must be produced from the join of one row in $\sigma[C_u]r_u$ and two rows (could be duplicates) in $\sigma[C_d]r_d$.

Note that R_u does not necessarily have to include a column `RowID`. The notation "$(GA_d^+, GA_u) \longrightarrow \texttt{RowID}(R_u)$ in the join of R_d and R_u" is simply a shorthand for the requirement that (GA_d^+, GA_u) uniquely identifies a row of R_u in the join of R_d and R_u.

The intuitive meaning of FD_1 and FD_2 is as follows. FD_1 ensures that each group in $\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$ corresponds to exactly one group in $\mathcal{G}[GA_d^+]\sigma[C_d]R_d$ Exact correspondence means that there is an one to one matching between rows in the two groups. This condition guarantees that two matching groups will produce the same aggregation values in E_1 and E_2 respectively. Note that the aggregation functions and aggregation expressions operate only on columns of R_d.

FD_2 ensures that each row in $F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]R_d$ contributes at most one row in the overall result of E_2 by joining with at most one row from $\sigma[C_u]R_u$. In other words, FD_2 prevents such a row from contributing two or more rows in the overall result of E_2. If FD_2 is not true, there may exist a row t_d in $F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]R_d$ which contributes two or more rows after the join in E_2. Then, since

- in E_1, after the join and before the aggregation, the rows, which correspond to those rows produced by the join between t_d and some rows of R_u in E_2, will belong to the same group in $\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$, and
- every group in $\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$ yields one row in the overall result of E_1,

E_1 contains one row corresponding to more than one rows in E_2, and consequently the transformation cannot be valid.

Lemma 1 : *The expression*

$$E_2' : \pi_A[GA_d, GA_u, FAA]$$
$$\sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]R_d \times \sigma[C_u]R_u)$$

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_u^+ of table $\sigma[C_u]R_u$ 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 $\pi_A[GA_u^+]\ \sigma[C_u]R_u$ to $\sigma[C_u]R_u$. Since the columns in R_u other than GA_u^+ do not participate in any of the operations in the expressions and the final projection is on columns (GA_d, GA_u, FAA), this change does not affect the result of the expression. Consequently the two expressions are equivalent. $\square$

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 $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$. Lemmas 2 - 6 essentially prove the Main Theorem for the case when GA_d^+ and GA_u^+ are both non-empty. The proof is divided into several steps: Lemmas 2 and 3 show the necessity of FD_1 and FD_2; Lemmas 4 and 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 using these lemmas.

5.1 Necessity

Lemma 2 : *If the two expressions E_1 and E_2' are equivalent, and GA_d^+ and GA_u^+ are both non-empty, then FD_1 holds in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$.*

Proof: We prove the lemma by contradiction. Assume that E_1 and E_2' are equivalent, and GA_d^+ and GA_u^+ are both non-empty, but FD_1 does not hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$. Then there must exist two valid instances r_d and r_u of R_d and R_u, 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 $t' \in \sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$ such that $t[GA_d, GA_u] \overset{\text{n}}{=} t'[GA_d, GA_u]$ but $t[GA_d^+] \overset{\text{n}}{\neq} t'[GA_d^+]$. Clearly, t and t' are produced from the join of two sets, $S_1 = \{t[\alpha(R_d)], t'[\alpha(R_d)]\} \subseteq \sigma[C_d]r_d$ and $S_2 = \{t[\alpha(R_u)], t'[\alpha(R_u)]\} \subseteq \sigma[C_u]r_u$. Note that $t[\alpha(R_d)]$ and $t'[\alpha(R_d)]$ must be two different rows whereas $t[\alpha(R_u)]$ and $t'[\alpha(R_u)]$ might be the same row.

Consider $E_1(r_1, r_2)$ first. Since $t[GA_d, GA_u] \overset{\text{n}}{=} t'[GA_d, GA_u]$, t and t' will be grouped into the same group in $\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$. All rows sharing the same value $t[GA_d, GA_u]$ in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$ will be grouped into this group. In $E_1(r_1, r_2)$, there is therefore exactly one row whose value for columns $[GA_d, GA_u]$ is $t[GA_d, GA_u]$.

Now consider $E_2'(r_1, r_2)$. Since $t[GA_d^+] \overset{\text{n}}{\neq} t'[GA_d^+]$, $t[\alpha(R_d)]$ and $t'[\alpha(R_d)]$ will be grouped into two different groups in $\mathcal{G}[GA_d^+]\sigma[C_d]r_d$. Denote these groups as g_1 and g_2 respectively. Therefore, $F[AA]\pi_A[GA_d^+, AA]\ \mathcal{G}[GA_d^+]\sigma[C_d]r_d$ must contain the following two rows: $F[AA]\pi_A[GA_d^+, AA]g_1$ and $F[AA]\pi_A[GA_d^+, AA]g_2$, whose values for columns GA_d^+ are $t[GA_d^+]$ and $t'[GA_d^+]$, respectively. Since t and t' are in the join result $\sigma[C_d \wedge C_0 \wedge C_u]$ $(r_d \times r_u)$, and GA_d^+ are the only columns of R_d participating in the join, it follows that $(F[AA]\pi_A[GA_d^+, AA]g_1) \circ t[\alpha(R_u)]$ and $(F[AA]\pi_A[GA_d^+, AA]g_2) \circ t'[\alpha(R_u)]$ must be in the join result $\sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\ \mathcal{G}[GA_d^+]\ \sigma[C_d]r_d \times \sigma[C_u]r_u)$. Therefore, there are (at least) two rows, in $E_2'(r_1, r_2)$, with the same value $(t[GA_d, GA_u])$ for columns $[GA_d, GA_u]$. Since there is only one row in $E_1(r_1, r_2)$ with the value $(t[GA_d, GA_u])$ for columns $[GA_d, GA_u]$, $E_1(r_d, r_u)$ and $E_2(r_d, r_u)'$ cannot be equivalent. This proves the lemma. $\Box$

Lemma 3 : *If the two expressions E_1 and E_2' are equivalent, and GA_d^+ and GA_u^+ are both non-empty, then FD_2 holds in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$.*

Proof: We prove the lemma by contradiction. Assume that E_1 and E_2' are equivalent, and GA_d^+ and GA_u^+ are both non-empty, but FD_2 does not hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$. Then, there must exist two valid instances r_d and r_u of R_d and R_u, respectively, with the following properties: (a) $E_1(r_d, r_u) = E_2'(r_1, r_2)$, and (b) there exist two rows t and $t' \in \sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$ such that $t[GA_d^+, GA_u] \overset{\text{n}}{=} t'[GA_d^+, GA_u]$ but $t[\alpha(R_u)] \overset{\text{n}}{\neq} t'[\alpha(R_u)]$. Clearly, t and t' are produced from the join of two sets, $S_1 = \{t[\alpha(R_d)], t'[\alpha(R_d)]\} \subseteq \sigma[C_d]r_d$

and $S_2 = \{t[\alpha(R_u)], t'[\alpha(R_u)]\} \subseteq \sigma[C_u]r_u$. Note that $t[\alpha(R_d)]$ and $t'[\alpha(R_d)]$ can be the same row but $t[\alpha(R_u)]$ and $t'[\alpha(R_u)]$ must be different rows.

First consider $E_1(r_1, r_2)$. Since $t[GA_d, GA_u] \overset{\text{n}}{=} t'[GA_d, GA_u]$, $t[\alpha(R_d)]$ and $t'[\alpha(R_d)]$ will be grouped into the same group in $\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$. All rows sharing the same value $t[GA_d, GA_u]$ in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$ will be grouped into this group. In $E_1(r_1, r_2)$ there is therefore exactly one row whose value for columns $[GA_d, GA_u]$ is $t[GA_d, GA_u]$.

Now consider $E_2'(r_1, r_2)$. Since $t[GA_d^+] \overset{\text{n}}{=} t'[GA_d^+]$, t and t' will be grouped into the same group in $\mathcal{G}[GA_d^+]\sigma[C_d]r_d$. Therefore, there is exactly one row in $F[AA]\pi_A[GA_d^+, AA]$ $\mathcal{G}[GA_d^+]\sigma[C_d]r_d$ having the value $t[GA_d^+]$ for columns GA_d^+. Denote this row by t_1. Since t and t' are in the join result $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$, and GA_d^+ are the only columns of R_d participating in the join, $t_1 \circ t[\alpha(R_u)]$ and $t_1 \circ t'[\alpha(R_u)]$ are in the join result $\sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]r_d \times \sigma[C_u]r_u)$. Therefore, there are (at least) two rows, $\pi_A[GA_d, GA_u, FAA]$ $(t_1 \circ t[\alpha(R_u)])$ and $\pi_A[GA_d, GA_u, FAA]$ $(t_1 \circ t'[\alpha(R_u)])$ in $E_2'(r_1, r_2)$, having the value $t[GA_d, GA_u]$ for columns $[GA_d, GA_u]$. Since there is only one row in $E_1(r_1, r_2)$ having the value $t[GA_d, GA_u]$ for columns $[GA_d, GA_u]$, $E_1(r_1, r_2)$ and $E_2'(r_1, r_2)$ cannot be equivalent. This proves the lemma. $\square$

Lemma 2 and Lemma 3 prove that FD_1 and FD_2 must hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$ if E_1 and E_2' are equivalent and GA_d^+ and GA_u^+ are both non-empty.

5.2 Distinctness

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

Proof: Clearly, (GA_d, GA_u) is the key of the derived table resulting from applying E_1 to valid instances r_d and r_u of R_d and R_u respectively. Therefore there are no duplicate rows in E_1. $\square$

Lemma 5 : *If FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$, and GA_d^+ and GA_u^+ 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_d and r_u of R_d and R_u, respectively, such that, FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$, but there exist two different rows $t, t' \in E_2'(r_1, r_2)$ which are duplicates of each other, that is, $t \overset{\text{n}}{=} t'$. Then there must exist two rows, $t_d, t_d' \in \sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]r_d \times$

$\sigma[C_u]r_u)$, such that $t = t_d[GA_d, GA_u, FAA]$, and $t' = t'_d[GA_d, GA_u, FAA]$. t_d and t'_d must be produced by the join between rows in $F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]r_d$ and $\sigma[C_u]r_u$. Assume $t_d = t_{21} \circ t_{22}$ and $t'_d = t'_{21} \circ t'_{22}$, where $t_{21}, t'_{21} \in F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]r_d$ and $t_{22}, t'_{22} \in \sigma[C_u]r_u$. There are two cases to consider.

Case 1: Assume that $t_{21}[GA_d^+] \overset{n}{\neq} t'_{21}[GA_d^+]$. Clearly, $t_{21}[GA_d] \overset{n}{=} t'_{21}[GA_d]$ and $t_{22}[GA_u] \overset{n}{=} t'_{22}[GA_u]$. Since FD_1 holds in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$, (GA_d, GA_u) functionally determines GA_d^+ in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$. Consider the grouping and aggregation in $F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]r_d$, these operations only merge several rows with the same value for columns GA_d^+ in $\sigma[C_d]r_d$ into one row, consequently the number of rows cannot increase and there is no new value for columns GA_d^+ in all resulting rows. It follows that (GA_d, GA_u) must still functionally determine GA_d^+ in $\sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]r_d \times \sigma[C_u]r_u)$. Since $t_d[GA_d, GA_u] \overset{n}{=} t'_d[GA_d, GA_u]$, $t_d[GA_d^+] \overset{n}{=} t'_d[GA_d^+]$ must hold. Therefore, $t_{21}[GA_d^+] \overset{n}{=} t'_{21}[GA_d^+]$, which is a contradiction.

Case 2: Assume that $t_{21}[GA_d^+] \overset{n}{=} t'_{21}[GA_d^+]$. Since the grouping in $\mathcal{G}[GA_d^+]\sigma[C_d]r_d$ is on GA_d^+, t_{21} and t'_{21} must be the same row, which is denoted by T_d. Since FD_2 holds in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$, (GA_d^+, GA_u) functionally determines $\texttt{RowID}(R_u)$ in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$. Similarly due to the reasons above, (GA_d, GA_u) must still functionally determine $\texttt{RowID}(R_u)$ in $\sigma[C_0](F[AA]\pi_A[GA_d^+, AA]\mathcal{G}[GA_d^+]\sigma[C_d]r_d \times \sigma[C_u]r_u)$. Since $t_d[GA_d, GA_u] \overset{n}{=} t'_d[GA_d, GA_u]$, t_{22} and t'_{22} must be the same row, which is denoted as T_u. The join between T_d and T_u can only generate one row. Therefore, t_d and t'_d are the same row. Hence t and t' must be the same row, which is a contradiction.

The two cases above are the only possible cases and both lead to contradictions. This proves the lemma. $\square$

5.3 Sufficiency

Lemma 6 : *If FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$, and GA_d^+ and GA_u^+ 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 produce duplicate rows if GA_d^+ and GA_u^+ are both non-empty. Let r_d and r_u be valid instances of R_d and R_u respectively. All we need to prove is that, if $t \in E_1(r_d, r_u)$, then $t \in E'_2(r_d, r_u)$ and vice versa.

Case 1: $t \in E_1(r_d, r_u) \Rightarrow t \in E'_2(r_d, r_u)$. Consider a row $t \in E_1(r_d, r_u)$. There exists a group $g \in \mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$ such that $t = F[AA]\pi_A[GA_d, GA_u, AA]g$.

Since $(GA_d, GA_u) \longrightarrow \texttt{RowID}(R_u)$ (follows from FD_1 and FD_2) in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$, there is exactly one row $t_u \in \sigma[C_u]r_u$ which joins with a subset g_d of rows in $\sigma[C_d]r_d$ to form g. We can therefore write $g \equiv g_d \times t_u$. Clearly, every row $t_p \in g_d$ has the property that $t_p[GA_d] \overset{\text{n}}{=} t[GA_d]$ and $C_0(t_p, t_u)$ is true. Furthermore, all rows in g_d have the same values for columns GA_d^+ because $(GA_d, GA_u) \longrightarrow (GA_d^+)$ holds in $\sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$. Therefore, for every row $t_o \in \sigma[C_d \wedge C_0 \wedge C_u](r_d \times r_u)$, if $t_o[GA_d^+] \overset{\text{n}}{=} t[GA_d^+]$ then $t_o[GA_d] \overset{\text{n}}{=} t[GA_d]$, and consequently $t_o \in g$, and $t_o[\alpha(R_d)] \in g_d$.

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

Therefore, the row $t' \equiv \pi_A[GA_d, GA_u, FAA]\sigma[C_0](F[AA]\pi_A[GA_d^+, AA]g_d \circ t_u)$ must then exist in $E_2'(r_d, r_u)$ and, since $g_d = g_d'$, $t \overset{\text{n}}{=} t_d'$. In other words, $t \in E_2'(r_d, r_u)$.

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

Now $g_d \subseteq g_d'$ because

(a) for any row $t_p \in \sigma[C_d]r_d$, if $t_p[GA_d] \overset{\text{n}}{=} t[GA_d]$ and $C_0(t_p, t_u)$ is true, then t_p must be in g_d';

(b) if a row $t_p \in g_d$, then $t_p[GA_d] \overset{\text{n}}{=} t[GA_d]$ and $C_0(t_p, t_u)$ is true.

Since $(GA_d, GA_u) \longrightarrow (GA_d^+)$, all rows in g_d' have the same value for columns (GA_d^+). Therefore, the rows in g_d and g_d' all have the same value for columns GA_d^+) but may differ on other columns. Since g_d contains all rows in $\sigma[C_d]r_d$ having the value $t[GA_d^+]$ for columns (GA_d^+), the rows in g'_d must all be in g_d. In other words, $g'_d \subseteq g_d$. Therefore $g_d = g'_d$. It follows that the row $t' \equiv \pi_A[GA_d, GA_u, FAA]\ ((F[AA]\pi_A[GA_d^+, AA]g'_d) \circ t_u) \in E'_2(r_1, r_2)$. Since $g_d = g'_d$, $t = t'$. In other words, $t \in E_1$. $\square$

Proof of the Main Theorem:

For the case that GA_d^+ and GA_u^+ are both non-empty, Lemma 2 and Lemma 3 prove that FD_1, FD_2 must hold in $\sigma[C_d \wedge C_0 \wedge C_u]\ (r_d \times r_u)$ if E_1 and E'_2 are equivalent(necessity). Lemma 6 shows that E_1 and E'_2 are equivalent if FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u]$ $(r_d \times r_u)$(sufficiency). Lemma 1 ensures that $E_2 = E'_2$. These lemmas together prove the theorem for the case that GA_d^+ and GA_u^+ are both non-empty. GA_d^+ and GA_u^+ cannot both be empty because in that case (GA_d, GA_u) 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_d^+ is empty but GA_u^+ is not empty. Since GA_d^+ is empty, GA_d and C_0 must be empty. Consequently the join must be a Cartesian product. But GA_u 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:

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

and

$$E_2 :\ \pi_A[GA_u, FAA](F[AA]\pi_A[AA]\sigma[C_d]R_d \times \pi_A[GA_u^+]\sigma[C_u]R_u).$$

Similarly, FD_1 and FD_2 degenerate to $(GA_u) \longrightarrow \phi$ and $(GA_u) \longrightarrow \texttt{RowID}(R_u)$ respectively. FD_1 is always true. Thus the necessary and sufficient condition is that FD_2 holds in $\sigma[C_d \wedge C_u](R_d \times R_u)$.

Since there is no grouping operation in E_2, $F[AA]\pi_A[AA]\sigma[C_d]R_d$ can yield only one row of result. Therefore its Cartesian product with R_u produces $|\sigma[C_u]R_u|$ rows. If FD_2 holds in $\sigma[C_d \wedge C_u](R_d \times R_u)$, then $(GA_u) \longrightarrow \texttt{RowID}(R_u)$ in $\sigma[C_u]r_u$ because the join is a Cartesian product. Therefore, the grouping in E_1 is actually based on every row of R_u. Therefore, E_1 and E_2 are equivalent. If FD_2 does not hold in $\sigma[C_d \wedge C_u](R_d \times R_u)$, then there must exist an instance of table R_u in which GA_u is not unique. It follows that E_1 must produce a table with cardinality less than $|R_u|$, and E_2 must produce a table with cardinality equal to $|R_u|$. Therefore E_1 and E_2 cannot be equivalent. Therefore, if and only if FD_2 holds in

$\sigma[C_d \wedge C_u](R_d \times R_u)$, E_1 is equivalent to E_2. Consequently our Main Theorem holds when GA_d^+ is empty.

Case 2: GA_u^+ is empty but GA_d^+ is not empty. Since GA_u^+ is empty, GA_u and C_0 must be empty. Therefore the join is a Cartesian product. Since C_0 is empty, GA_d^+ must be the same as GA_d.

Hence, E_1 and E_2 degenerate to:

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

and

$$E_2: \; \pi_A[GA_d, FAA]\sigma[C_0](F[AA]\pi_A[GA_d, AA]\mathcal{G}[GA_d]\sigma[C_d]R_d \times \sigma[C_u]R_u)$$

respectively. FD_1 and FD_2 degenerate to $(GA_d) \longrightarrow GA_d$ and $(GA_d) \longrightarrow \texttt{RowID}(R_u)$ 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 $\sigma[C_u]r_u$ can contain no more than one row.

Clearly, if FD_2 holds in $\sigma[C_d \wedge C_u](r_d \times r_u)$, then E_1 and E_2 are equivalent. If FD_2 does not hold in $\sigma[C_d \wedge C_u](r_d \times r_u)$, that is, $\sigma[C_u]r_u$ contains more than one row, then, for every $t_1 \in \sigma[C_d]r_d$, E_2 must contain more rows with the value $t_1[GA_d]$ for columns GA_d than E_1 does. Hence E_1 and E_2 cannot be equivalent. Consequently, our main theorem holds also when GA_u^+ is empty. $\square$

Corollary 1 : *The two expressions*

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

and

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

where d is either A or D, are equivalent if FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$.

The Main Theorem assumes that the final selection columns are the same as the grouping columns(GA_d, GA_u) and the final projection must be an `ALL` projection. This corollary

relaxes these two restrictions, i.e., the final selection columns may be a subset(SGA_d, SGA_u) of the grouping columns(GA_d, GA_u), and the final projection can be a `DISTINCT` projection. The two conditions FD_1 and FD_2 are still sufficient but not necessary.
Proof: If FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$, then

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

and

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

are equivalent according to our main theorem. Then

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

and

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

are also equivalent because they project on the same columns in the last projection. Therefore, the two expressions in the corollary are equivalent when $d = A$.

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

and

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

are also equivalent because they both eliminate duplicates at the last projection. Therefore, the two expressions in the corollary are also equivalent when $d = D$. $\square$

Corollary 2 *:(***Key Joins***)* *Consider a query where the join is an equi-join between columns of tables R_d and key columns of table R_u. The two expressions in Corollary 1 are equivalent if GA_d contain all R_d columns participating in the joins with R_u.*

Proof: The condition requires that $GA_d \Longleftrightarrow GA_d^+$. Therefore, $FD_1, (GA_d, GA_u) \longrightarrow GA_d^+$, clearly holds. Since the join contains an equi-join between columns of tables R_d and key columns of table R_u, the join columns of R_d functionally determine the $RowId$ of R_u tables. Therefore, $GA_d \longrightarrow RowId(R_u)$ holds. Therefore FD_2 holds. $\Box$

It is easy to see that, when (1) the joins are all foreign key joins (all join columns of R_d tables are foreign keys to R_u tables), and (2) the R_d grouping columns GA_d contain all columns involving the joins with R_u, the two expressions in Corollary 1 are equivalent.

A more special case of the above observation is when (1) the joins are foreign key joins with all joins columns of R_d tables are foreign keys to R_u tables; (2) the R_d grouping columns GA_d are all joins columns of the joins with R_u tables; and (3) GA_u is empty. These conditions are identical to the conditions presented in [2]. The advantage of these conditions is that it does not need any functional dependency checking, but of course, it is very restricted and can only catch simple cases.

6 Algorithms to Test the Conditions

To apply the transformation in the Main Theorem, 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_d and R_u. To achieve this, we can make use of semantic integrity constraints and the conditions specified in the query. SQL2 [12] allows users to specify integrity constraints on the valid state of SQL data and these constraints are enforced by the SQL implementation.

There are two types of constraints, immediate constraints and deferred constraints. *Immediate constraints* are immediately enforced after every update within a transaction. *Deferred constraints* are enforced at the end of a transaction. A deferred constraints may not hold in the database in the middle of a transaction, Therefore, for a query that is within one of the following two types of transactions:

- a transaction with no updates, or
- a transaction containing updates and all the constraints affected by updates before the query are immediate constraints,

we can assume that all integrity constraints hold in the join result of R_d and R_u and can add these constraints into the WHERE clause of a query without changing the result of the query.

In addition, the predicates in the `WHERE` clause and `HAVING` clause 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. For the rest of this paper, without loss of generality, we assume that all constraints are immediate constraints.

6.1 Constraints in SQL2

In SQL2[12, 9, 3], 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 in Figure 6 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 6: SQL constraints

Column constraints include `NOT NULL` and `CHECK` constraints. In Figure 6, 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 6, 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` must satisfy the constraint. A domain constraint is 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 6, 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 (foreign key constraint) 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 candidate key or primary key in some table(may be the same as the original table). In Figure 6, the statement `FOREIGN KEY (DeptID) REFERENCES Dept` is an example of a referential integrity constraint.

An assertion constraint specifies a restriction involving multiple tables. It is defined by the statement `CREATE ASSERTION` outside of the table definition.

We can add all immediate constraints into the `WHERE` clause of a query without changing the result of the query. 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.

Within a query block, the values of host variables, columns referencing outer tables and parameter markers are constants. We can regard these values as *input parameters* to the query block. We denote the set of input parameters to a query block by I, and the Cartesian product of the domains of all the input parameters by H. We call H the *domain* of I.

We use T_d and T_u to denote the Boolean expressions representing domain, column, referential integrity and assertion constraints in table R_d and R_u, respectively. We use $K_i(R)$ 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 7.

Symbol	Definitions
$K_i(R)$	The ith candidate(primary) key of table R
T_d	Column, domain, referential integrity and assertion constraints on table R_d
T_u	Column, domain, referential integrity and assertion constraints on table R_u
I	The set of input parameters for a query block
H	The domain of the set of input parameters I.
$\|R\|$	the cardinality of a table R

Figure 7: Summary of Symbols

6.2 Using Semantic Constraints to Test the Conditions

There are many ways to test the conditions FD_1 and FD_2. The semantic constraints in SQL discussed in Section 6.1 can be used to determine whether FD_1 and FD_2 are true.

Theorem 2 : *(***Exploiting Semantic Constraints***) FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u]$ $(R_d \times R_u)$ if the following two conditions are true.*
Condition A:

$$\begin{aligned}
&\forall h \in H, \forall t, t' \in Domain(R_d \times R_u) \\
&\quad \{\lfloor C_d(t,h) \wedge C_d(t',h) \wedge C_0(t,t',h) \wedge C_0(t,t',h) \wedge C_u(t,h) \wedge C_u(t',h) \\
&\qquad\qquad \wedge T_1(t,h) \wedge T_2(t',h)\rfloor \\
&\quad \wedge(\bigwedge_{\forall i}(t[K_i(R_d)] \overset{n}{=} t'[K_i(R_d)] \Rightarrow t[\alpha(R_d)] \overset{n}{=} t'[\alpha(R_d)])) \\
&\quad \wedge(\bigwedge_{\forall i}(t[K_i(R_u)] \overset{n}{=} t'[K_i(R_u)] \Rightarrow t[\alpha(R_u)] \overset{n}{=} t'[\alpha(R_u)]))\} \\
&\Rightarrow \{(t[GA_d, GA_u] \overset{n}{=} t'[GA_d, GA_u] \Rightarrow t[GA_d^+] \overset{n}{=} t'[GA_d^+])\}
\end{aligned}$$

Condition B:

$$\begin{aligned}
&\forall h \in H, \forall t, t' \in Domain(R_d \times R_u)\\
&\quad \{\lfloor C_d(t,h) \wedge C_d(t',h) \wedge C_0(t,t',h) \wedge C_0(t,t',h) \wedge C_u(t,h) \wedge C_u(t',h)\\
&\qquad\qquad \wedge T_1(t,h) \wedge T_2(t',h)\rfloor\\
&\quad \wedge(\bigwedge_{\forall i}(t[K_i(R_d)] \overset{\text{n}}{=} t'[K_i(R_d)] \Rightarrow t[\alpha(R_d)] \overset{\text{n}}{=} t'[\alpha(R_d)]))\\
&\quad \wedge(\bigwedge_{\forall i}(t[K_i(R_u)] \overset{\text{n}}{=} t'[K_i(R_u)] \Rightarrow t[\alpha(R_u)] \overset{\text{n}}{=} t'[\alpha(R_u)]))\}\\
&\Rightarrow \{(t[GA_d^+, GA_u] \overset{\text{n}}{=} t'[GA_d^+, GA_u] \Rightarrow t[\texttt{RowID}(R2)] \overset{\text{n}}{=} t'[\texttt{RowID}(R2)])\}
\end{aligned}$$

Condition A and B correspond to FD_1 and FD_2 respectively. The consequents of Condition A and B, $(t[GA_d, GA_u] \overset{\text{n}}{=} t'[GA_d, GA_u] \Rightarrow t[GA_d^+] \overset{\text{n}}{=} t'[GA_d^+])$ and $(t[GA_d^+, GA_u] \overset{\text{n}}{=} t'[GA_d^+, GA_u] \Rightarrow t[\texttt{RowID}(R2)] \overset{\text{n}}{=} t'[\texttt{RowID}(R2)])$, 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_d and R_u, part one, $\lfloor C_d(t,h) \wedge C_d(t',h) \wedge C_0(t,t',h) \wedge C_0(t,t',h) \wedge C_u(t,h) \wedge C_u(t',h) \wedge T_1(t,h) \wedge T_2(t',h)\rfloor$, requires that all input parameters and rows satisfy the join condition $C_d \wedge C_0 \wedge C_u$ and all the semantic constraints except the key constraints of table R_d and R_u; part two and three, $(\bigwedge_{\forall i}(t[K_i(R_d)] \overset{\text{n}}{=} t'[K_i(R_d)] \Rightarrow t[\alpha(R_d)] \overset{\text{n}}{=} t'[\alpha(R_d)]))$ and $(\bigwedge_{\forall i}(t[K_i(R_u)] \overset{\text{n}}{=} t'[K_i(R_u)] \Rightarrow t[\alpha(R_u)] \overset{\text{n}}{=} t'[\alpha(R_u)]))$, require that all rows satisfy the key constraints of table R_d and R_u. The proof of this theorem is straightforward.

Proof: Assume that the conditions stated in the theorem hold. In the join result $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$, all semantic constraints (key constraints, T_1 and T_2) and all join conditions C_d, C_0, C_u must be satisfied, that is, the antecedents of both conditions are true. Therefore the two consequents:

$$t[GA_d^+, GA_u] \overset{\text{n}}{=} t'[GA_d^+, GA_u] \Rightarrow t[\texttt{RowID}(R2)] \overset{\text{n}}{=} t'[\texttt{RowID}(R2)]$$

and

$$t[GA_d, GA_u] \overset{\text{n}}{=} t'[GA_d, GA_u] \Rightarrow t[GA_d^+] \overset{\text{n}}{=} t'[GA_d^+]$$

are both true. This means that FD_1 and FD_2 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$. □

If we can design an efficient algorithm to test the satisfiability of the conditions in Theorem 2, we can use it to determine the validity of the transformation. 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 2 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 TestGroupJoin: A Fast Algorithm

In this section, we 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 $\sigma[C_d \wedge C_0 \wedge C2](R_d \times R_u)$, and returns `NO` when it cannot.

Atomic conditions not involving '=' are seldom useful for generating new functional dependencies. The algorithm 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 $(v_1 = v_2)$, where v_1, v_2, v are columns and c is a constant or an input parameter. An input parameter can be treated as a constant because its value is fixed when evaluating the query block. We first present an algorithm that finds the columns functionally determined by a set of columns.

Algorithm FindClosure: *find the columns functionally determined by a set of columns*

Inputs: PRD, set of Predicates;
FDs, set of functional dependencies;
S, set of columns.

Output: CS, set of columns functionally determined by S

```
1   Convert PRD into conjunctive normal form: $C = D_1 \wedge D_2 \wedge ... \wedge D_m$.
2   for each $D_i$ do
3       if $D_i$ contains an atomic condition not of Type 1 or Type 2 then delete $D_i$ from $C$.
4   end for
5   convert $C$ into disjunctive normal form: $C = E_1 \vee E_2 \vee ... \vee E_n$.
```

$$\begin{tikzcd}
 & A1 \arrow[dr, bend left, "b"] & \\
A2 \arrow[ur, "a"] & & A3 \arrow[ul, "a"'] \arrow[dl, leftrightarrow, "c"] \\
 & A4 \arrow[uu, "a"] &
\end{tikzcd}$$

Known conditions and constraints:

$a : A_d = 25;\ b : A_d \longrightarrow A_3;\ c : A_3 = A_4$

Conclusion: $A_u \longrightarrow A_4$

Figure 8: Illustration of Algorithm FindClosure

```
6     for each $E_i$ do
7         $CS_i = S$;
8         for each atomic condition of Type 1 ($v = c$) in $E_i$ do add $v$ into $CS_i$.
          — Compute the transitive closure of $CS_i$ based on
          — Type 2 atomic conditions and key constraints.
9         while ($\exists$ a Type 2 condition $(v_1 = v_2) \in C$ such that $v_1 \in CS_i$ and $v_2 \notin CS_i$) or
                ($\exists$ a functional dependency $(d \longrightarrow v_2) \in$ FDs such that $d \subseteq CS_i$ and $v_2 \notin CS_i$)
10            add $v_2$ to $CS_i$.
11        end while
12    end for
13    $CS = CS_1 \cap CS_2 \cap ... \cap CS_n$;
14    Return CS.
```

End Algorithm FindClosure

The basic idea of the algorithm is quite simple. Lines 1 to 4 discard all non-equality conditions in join conditions and semantic constraints. Then we convert the conditions into disjunctive normal form in Line 5. For each conjunct E_i, we then compute the closure on S as illustrated by Figure 8. Assume that an E_i contains the following constraints: $\{a : A_1 = 25; c : A_3 = A_4\}$, and functional dependency $\{b : A_1 \longrightarrow A_3\}$ holds. These assertions are satisfied in the join result. Since A_1 is a constant in the join result, every column functionally determines A_1, which is represented by the directed edges marked by a in Figure 8. These FDs are generated by Line 8 in the algorithm. Furthermore, since A_3 equals to A_4, they functionally determine each another. This is illustrated by a bi-directional edge marked by c in Figure 8. The FD $A_1 \longrightarrow A_3$ is also shown as a directed edge marked by b in the figure.

Due to the transitive property of functional dependencies, we can draw the conclusion that $A_2 \longrightarrow A_4$. If $S = \{A_2\}$, the algorithm generates the set $CS_i = \{A_1, A_2, A_3, A_4\}$. Therefore CS_i contains the columns functionally determined by S given that E_i is true and the input FDs are true. Since the input PRD is the disjunction of all E_i, the intersection of $CS_i^{\prime}s$ contains the set of columns functionally determined by S given that PRD is true and the input FDs are true.

Therefore, if $A_i \longrightarrow A_j$ is to be tested, where A_i and A_j are some sets of columns, we can see whether the CS set for A_i contains A_j. If so, then $A_i \longrightarrow A_j$ is true. The following algorithm can then be used to determine whether grouping and join can be interchanged:

Algorithm TestGroupJoin: *determine whether grouping can be interchanged with join*

Inputs: Predicates PRD C_d, C_0, C_u, T_d, T_u; key constraints of R_d and R_u.

Output: `YES` or `NO`.

```
— Convert all key constraints into functional dependencies.
$FDs = empty$;
for each key $K$ do
    for each column $A$ in the same table as $K$ do
        $FDs = FDs \cup (K \longrightarrow A)$
    end for
end for
— Check whether $FD_1, (GA_d, GA_u) \longrightarrow GA_d^+$, holds
$CS = FindClosure(PRD = C_d \wedge C_0 \wedge C_u \wedge T_d \wedge T_u;\ FDs; S = GA_d \cup GA_u)$;
if $(GA_d^+ \not\subseteq CS)$ Return NO
— Check whether $FD_2, (GA_d^+, GA_u) \longrightarrow \texttt{RowID}(R_u)$, holds
$CS = FindClosure(PRD = C_d \wedge C_0 \wedge C_u \wedge T_d \wedge T_u;\ FDs; S = GA_d^+ \cup GA_u)$;
for each table $T \in R_d$ do
    if ( $Key(T) \not\subseteq CS$) return NO
end for
Return YES
```

End Algorithm TestGroupJoin

If the joins satisfy the conditions in Corollary 2, the transformation can always be done. Therefore, a very simple algorithm would consist of checking whether: (1) the joins between R_u tables and R_d tables are equi-joins mentioning all keys of R_u and (2) the R_d grouping columns GA_d contain all R_d columns participating in the joins with R_u.

7 Queries Including HAVING

7.1 Class of Queries Considered

In this section, we consider SQL queries containing both GROUP BY and HAVING clauses. We assume that the query excluding the HAVING clause belongs to the class of queries described in Section 3. The HAVING clause operates on the grouped table resulting from the GROUP BY. It applies the search conditions specified in the HAVING clause to each group and produces a grouped table. Only the groups for which the evaluation of the search condition is true remain in the result from the HAVING clause. The search condition may contain aggregation functions referencing some aggregation columns. The difference between a HAVING predicate and a WHERE predicate is that the former operates on a grouped table, while the later operates on a regular table. Our objective is to find out when both GROUP BY and HAVING can be pushed past joins and vice versa.

Assume that the predicate in the HAVING clause of a query block is in conjunctive normal form. According to SQL 92, a HAVING clause may contain conjunctive terms on the grouping columns and aggregation functions operating on some columns [7]. We also call any column occurring as an operand of an aggregation function in a HAVING clause *an aggregation column.* If we push down the GROUP BY, each group will be aggregated into one row and no aggregation functions can be performed on the rows after the joins. Therefore, if there are any aggregations functions in the HAVING clause, these functions must be pushed down along with the GROUP BY in order to be evaluated. The grouping columns are still available after the join. Hence, after the join, the predicate in the HAVING clause can still be evaluated.

[7]Actually, SQL 92 also allows the HAVING clause to reference a column, called an outer reference, in a table outside the scope of the query block being considered. This column can be treated as a constant with respect to this query block.

Therefore, in addition to the assumptions made in Section 3, we have the following two assumptions:

- The tables in the FROM clause can be partitioned into two groups, R_d and R_u. R_d contains the aggregation columns, denoted by AA, in both the SELECT clause and the HAVING clause, while R_u does not. R_d and R_u are both non-empty. If we cannot perform this partitioning, the transformation cannot be done.
- Since conjunctive terms in the HAVING clause involving only grouping columns can be moved to the WHERE clause without changing the result of the query, we assume that this has been done and the original query does not contain any such conjunctive terms in the HAVING clause.

Therefore, the predicate on aggregation expressions in the HAVING clause can be expressed in the following conjunctive form: $H_d \wedge H_0$, where H_d contains columns exclusively from R_d, and H_0 contains aggregation columns from R_d and non-aggregation columns from R_u. The aggregation expressions mentioned in H_0 and H_d can be denoted as $F_0(AA)$ and $F_d(AA)$ respectively, which are arrays of aggregation functions and/or aggregation expressions operating on AA. Since all columns in AA belong to R_d, i.e., there is no aggregation columns from R_u, there is no corresponding H_u conjunctive terms.

Clearly, when performing group-by push down, H_d can be pushed down along with the group-by. That is, we push down group-by along with HAVING predicate past a join. This is often beneficial because it reduces the input cardinality to the join. When performing group-by pull up, we also pull up the HAVING predicate in the aggregated view with the group-by. This is often beneficial when we have a selective join. In summary, the class of queries we consider is defined as follows:

SELECT [ALL/DISTINCT]	$GA_d,\ GA_u,\ F(AA)$
FROM	$R_d,\ R_u$
WHERE	$C_d \wedge C_0 \wedge C_u$
GROUP BY	GA_d, GA_u
HAVING	$H_0(F_0(AA)) \wedge H_d(F_d(AA))$

The alternative way to write the query is:

SELECT [ALL/DISTINCT]	$GA_d,\ GA_u,\ FAA$

```
    FROM                  $R'_d$, $R_u$
    WHERE                 $C_u \wedge C_0 \wedge H_0(F_0AA)$
where
    $R'_d(GA_d^+, FAA, F_0AA) ==$
    SELECT ALL            $GA_d^+, F(AA), F_0(AA)$
    FROM                  $R_d$
    WHERE                 $C_d$
    GROUP BY              $GA_d^+$
    HAVING                $H_d(F_d(AA))$
```

where GA_d^+ is $GA_d \cup (\alpha(C_0) \cap R_d)$, i.e., the grouping columns belonging to R_d, and the join columns of R_d mentioned in the WHERE clauses of the two join queries. Note that, the R_d join columns in the HAVING clause must be grouping columns so those columns have been included in GA_d^+. In the above queries, we use $H_0(F_0AA)$ to denote the predicate obtained by replacing the aggregation expressions in $H_0(F_0(AA))$ by F_0AA respectively.

Note that, GA_d^+ defined here includes the grouping columns belonging to tables containing the aggregation columns in both the WHERE clause and the HAVING clause. The GA_d^+ defined in Section 3 contains only grouping columns belonging to tables containing the aggregation columns in the WHERE clause, which is a special case of GA_d^+ defined here since the class of queries we considered before does not contain a HAVING clause.

As a shorthand notation, we add the notation $\mathcal{H}[P]R$ to our SQL algebra, representing the operation of selecting the groups in the grouped table R satisfying predicate P. The result of this operation is also a grouped table. Also, we use the shorthand notation $\pi_A[GA, F[AA]]$ to represent $\pi_A[GA, FAA]F[AA]\pi_A[GA, AA]$, where FAA are the columns produced by $F[AA]$. The above two queries can then be expressed as

$$\begin{aligned} E_3 :\ & \pi_A[GA_d, GA_u, F[AA]]\ \ \mathcal{H}[H_d[F_d[AA]] \wedge H_0[F_0[AA]]] \\ & \mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u) \end{aligned}$$

and

$$\begin{aligned} E_4 :\ & \pi_A[GA_d, GA_u, FAA]\sigma[C_0 \wedge C_u \wedge H_0(F_0AA)] \\ & (\{\pi_A[GA_d^+, F(AA), F_0(AA)]\ \ \mathcal{H}[H_d(F_d[AA])]\ \ \mathcal{G}[GA_d^+]\sigma[C_d]R_d\} \times R_u) \end{aligned}$$

7.2 Main Theorem with HAVING

Theorem 3 (Main Theorem with HAVING): *The expressions*

$$E_3 : \; \pi_A[GA_d, GA_u, F[AA]] \;\; \mathcal{H}[H_d[F_d[AA]] \wedge H_0[F_0[AA]]]$$
$$\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$$

and

$$E_4 : \; \pi_A[GA_d, GA_u, FAA]\sigma[C_0 \wedge C_u \wedge H_0(F_0AA)]$$
$$(\{\pi_A[GA_d^+, F(AA), F_0(AA)] \;\; \mathcal{H}[H_d(F_d[AA])] \;\; \mathcal{G}[GA_d^+]\sigma[C_d]R_d\} \times R_u)$$

are equivalent if and only if the following two functional dependencies hold in the join of R_d and R_u, $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$:

$$FD_3 : \;\; (GA_d, GA_u) \longrightarrow GA_d^+$$
$$FD_4 : \;\; (GA_d^+, GA_u) \longrightarrow \texttt{RowID}(R_u)$$

As will be shown in the proof, an SQL query with a HAVING clause can be transformed into a query without a HAVING clause. We prove the theorem by transforming E_3 into an equivalent query without a HAVING clause, then applying the transformation in our Main Theorem with known sufficient and necessary conditions, and finally converting the sub-query into a query with a HAVING clause.

Proof:

When applying the HAVING predicate to the grouped table resulting from the GROUP BY operation, we first evaluate the aggregation functions, then apply the predicate on these aggregation functions. Therefore, E_3 is equivalent to:

$$\pi_A[GA_d, GA_u, FAA]\sigma(H_d[F_dAA] \wedge H_0[F_0AA])$$
$$\pi_A[GA_d, GA_u, F[AA], F_d[AA], F_0[AA]]\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u) \qquad (1)$$

Now consider the last part of the above expression:

$$\pi_A[GA_d, GA_u, F[AA], F_d[AA], F_0[AA]]$$
$$\mathcal{G}[GA_d, GA_u]\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u) \qquad (2)$$

It belongs to the class of queries defined in Section 3. Therefore, according to the Main Theorem, *if and only if* FD_3 and FD_4 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$, (2) is equivalent to

$$\begin{aligned}&\pi_A[GA_d, GA_u, FAA, F_dAA, F_0AA]\sigma[C_0 \wedge C_u]\\ &\qquad (\{\pi_A[GA_d^+, F[AA], F_d[AA], F_0[AA]] \;\; \mathcal{G}[GA_d^+]\sigma[C_d]R_d\} \times R_u)\end{aligned} \tag{3}$$

Hence, *if and only if* FD_3 and FD_4 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$, (1) is equivalent to

$$\begin{aligned}&\pi_A[GA_d, GA_u, FAA]\sigma(H_d[F_dAA] \wedge H_0[F_0AA])\\ &\quad (\pi_A[GA_d, GA_u, FAA, F_dAA, F_0AA]\sigma[C_0 \wedge C_u]\\ &\qquad \{\pi_A[GA_d^+, F[AA], F_d[AA], F_0[AA]] \;\; \mathcal{G}[GA_d^+]\sigma[C_d]R_d\} \times R_u)\end{aligned} \tag{4}$$

Since

- $H_d[F_dAA]$ mentions only columns of R_d;
- the columns F_dAA are deleted in the final projection; and
- the columns F_dAA are not used in the join between R_d and R_u,

we can move the predicate $H_d[F_dAA]$ to the sub-query and remove F_dAA from the outer query. Therefore the above expression is equivalent to

$$\begin{aligned}&\pi_A[GA_d, GA_u, FAA]\sigma(H_0[F_0AA])\\ &\quad \pi_A[GA_d, GA_u, FAA, F_0AA]\sigma[C_0 \wedge C_u]\\ &\qquad (\{\pi_A[GA_d^+, FAA, F_0AA]\sigma(H_d[F_dAA])\\ &\qquad\quad \pi_A[GA_d^+, FAA, F_d[AA], F_0[AA]] \;\; \mathcal{G}[GA_d^+]\sigma[C_d]R_d\} \times R_u)\end{aligned} \tag{5}$$

Clearly, $\sigma(H_d[F_dAA])$ is equivalent to a `HAVING` clause, which gives us

$$\begin{aligned}&\pi_A[GA_d, GA_u, FAA]\sigma(H_0[F_0AA])\\ &\quad \pi_A[GA_d, GA_u, FAA, F_0AA]\sigma[C_0 \wedge C_u]\\ &\qquad (\{\pi_A[GA_d^+, F[AA], F_0[AA]] \;\; \mathcal{H}(H_d[F_d[AA]])\\ &\qquad \mathcal{G}[GA_d^+]\sigma[C_d]R_d\} \times R_u)\end{aligned} \tag{6}$$

The selection $\sigma(H_0[F_0AA])$ is a join condition and columns F_0AA do not appear in the final result. Therefore, the above query is equivalent to

$$\begin{aligned}
&\pi_A[GA_d, GA_u, FAA]\sigma[C_0 \wedge C_u \wedge H_0[F_0AA]] \\
&\qquad (\{\pi_A[GA_d^+, F[AA], F_0[AA]] \ \ \mathcal{H}(H_d[F_d[AA]]) \\
&\qquad \mathcal{G}[GA_d^+]\sigma[C_d]R_d\} \times R_u)
\end{aligned} \tag{7}$$

Expression (7) is E_4. Since

- E_3 is equivalent to (1);
- E_4 is equivalent to (4); and
- Expressions (1) and (4) are equivalent if and only if FD_3 and FD_4 hold in $\sigma[C_d \wedge C_0 \wedge C_u](R_d \times R_u)$,

the theorem holds. $\square$

8 Partitioning of Tables

8.1 Table Partitioning

In Sections 3 and 7, we assumed that tables in the `FROM` clause can be partitioned into two groups, R_d and R_u. R_u is the Cartesian product of tables containing aggregation columns, and R_u is the Cartesian product of the rest of the tables. We call the tables in R_d *pushed down* tables, and the tables in R_u *pulled up* tables. If some partitioning of the tables into R_d and R_u satisfies the conditions of our theorems, the conditions would still hold if we add a subset of R_u into R_d and subtract the subset from R_u. In other words, even though all tables in R_d which contain the aggregation columns must be pushed down, only a subset of the R_u tables need to be pulled up. If a subset of R_u is pushed down along with R_d, the assumption that "aggregation columns must belong to the tables pushed down" is still satisfied. Therefore, there can be multiple valid partitionings of tables. Example 2 shows a case with three valid table partitionings(see Section 10). In this section, we will investigate how to efficiently obtain all valid partitionings.

8.2 Rewriting of Necessary and Sufficient Conditions

First, let us prove the following lemma:

Lemma 7 (Rewritten FDs): *The two conditions in Theorem 3:*

$$FD_3: \ (GA_d, GA_u) \longrightarrow GA_d^+$$
$$FD_4: \ (GA_d^+, GA_u) \longrightarrow \texttt{RowID}(R_u)$$

are equivalent to

$$FD_5: \ (GA_d, GA_u) \longrightarrow JA_d$$
$$FD_6: \ (GA_d, GA_u) \longrightarrow \texttt{RowID}(R_u)$$

where $JA_d = \alpha(R_d) \cap \alpha(C_0)$).

Observe that (GA_d, GA_u) is the set of all grouping columns, and JA_d is the set of *join columns* of R_d tables.

Proof:

We only need to prove that $FD_3, FD_4 \Rightarrow FD_5, FD_6$, and $FD_5, FD_6 \Rightarrow FD_3, FD_4$.

$FD_3, FD_4 \Rightarrow FD_5, FD_6$: Since $JA_d \subseteq GA_d^+$, FD_5 holds when FD_3 is true. Since FD_3 holds, we have:

$$(GA_d, GA_u) \longrightarrow GA_d^+ \cup GA_u$$

Combining the above expression with FD_4, we obtain FD_6.

$FD_5, FD_6 \Rightarrow FD_3, FD_4$: FD_4 is clearly implied by FD_6. From FD_5, we have:

$$(GA_d, GA_u) \longrightarrow JA_d \cup GA_d.$$

Since $GA_d^+ \equiv JA_d \cup GA_d$ by its definition, it follows that FD_3 also holds.

□

8.3 Algorithm for Finding All Valid Partitionings

According to Lemma 7, to test whether the transformation is valid, we can simply perform a closure on the grouping columns. Then if both JA_d and $\texttt{RowID}(R_u)$ are in the closure, the transformation is valid. Note that, this closure is partitioning independent. Therefore, we only need to compute the closure once. We use the notation $FD(R_d, R_u)$ to denote that the table partitioning of R_d and R_u satisfies the functional dependency FD.

We can also easily prove the following corollaries:

Corollary 3 :
If $FD_6(R_d, R_u)$ holds, then, $FD_6(R_d + R', R_u - R')$, where $R' \subseteq R_u$, also holds.

Corollary 4 :
If $FD_5(R_d + R', R_u - R')$ holds, then, $FD_5(R_d, R_u)$, where $R' \subseteq R_u$, also holds.

Since all aggregation columns must belong to R_d, the minimum subset of tables that can be pushed down is the set of tables containing the aggregation columns. However, according to FD_6, all R_u tables must also be functionally determined by the grouping columns. Consequently, all tables not functionally determined by the grouping columns have to be pushed down. The minimum set of tables that can be pushed down is the set of tables containing the aggregation columns, plus the set of tables not functionally determined by the grouping columns. The remaining tables are the maximum set of tables that can be pulled up.

We can then start from the minimum R_d set and maximum R_u set, and try to add a subset of R_u to R_d to find all possible valid partitionings. This way we will find all valid partitionings. The algorithm follows.

Algorithm Find Partitionings: *find all valid table partitionings*

Inputs: Predicates PRD C_d, C_0, C_u, T_d, T_u;
key constraints of all table R

Output: *Valid table partitionings for interchanging grouping and join*

```
    — Convert all key constraints into functional dependencies $FDs$.
1   $FDs = empty$;
2   for each key $K$ do
3       for each column $A$ in the same table as $K$ do
4           $FDs = FDs \cup (K \longrightarrow A)$
5       end for
6   end for
    — Find the closure of all grouping columns
7   $S = FindClosure(PRD = C_d \wedge C_0 \wedge C_u \wedge T_d \wedge T_u;\ FDs; S = GA_d \cup GA_u)$;
```

```
      — every table containing an aggregation column or not
      — functionally determined by the grouping columns must be pushed down
8     $R_d = \{R_i | R_i \in R, \exists A \in AA$ such that $A \in \alpha(R_i) or \ \nexists Key(R_i)$ such that $Key(R_i) \subseteq S\}$;
      — The rest of the tables can either be pushed down or pulled up
9     $R_u = R - R_d$;
10    if $JA_d \subseteq S$ then
          — we have a valid partitioning (with minimal $R_d$)
11        OUTPUT( $R_d, R_u$) ;
          — try to push down additional tables from $R_u$
12        for each subset $R' \subset R_u$ do
13            $R'_u = R_u - R'$; $R'_d = R_d + R'$;
14            if $JA'_d \subseteq S$ then OUTPUT($R'_d, R'_u$);
15        end for
16    end if
```

End Algorithm Find Partitionings

Note that, at line 10, we try to find more valid partitionings only when $JA_d \subseteq S$, according to Corollary 4. Also, FD_6 holds for the new partitioning at line 13, according to Corollary 3.

There are two interesting special cases:

- If `RowID`(all tables)$\subseteq S$ then each group has only one row. Therefore, the grouping operation can be eliminated. The aggregation expressions can then be modified so that no `GROUP BY` clause is needed for this query. For example, COUNT (*) can be modified as 1, and SUM (COLUMN) can be modified as COLUMN.

- If α(all join predicates)$\subseteq S$, then FD_5 is always true. We then only need to check FD_6. Let $R_{AA} = \{R_i | R_i \in R, \exists A \in AA$ such that $A \in \alpha(R_i)\}$, i.e., all tables containing the aggregation columns; and let $T = \{R_i | RowID(R_i) \in S\} - R_{AA}$, i.e., all tables except R_{AA} with `RowID` in S. Then, $R_u =$ any subset R' of T and $R_d = R - R'$ will form a valid partition.

9 Column Substitution

The algorithm in the previous section finds all valid partitionings of the tables in the FROM clause. The tables are partitioned into two groups, one to be pushed down and one to be pulled up. However, some queries may not be transformable because: (a) no partitioning is possible, e.g., every table contains some aggregation columns; or (b) it can be somehow partitioned but the testing algorithm returns NO. Column substitution, that is substituting a column with another equivalent column, can be used to generate additional equivalent queries. First, column substitution can be employed to obtain a set of equivalent queries. We can then find all possible partitionings of tables for each query in the set. Theoretically, all possible partitionings of the tables for all queries can be considered and the best one, if beneficial, can be picked for the transformation.

Two columns are equivalent if they functionally determine each other in the result of the current query block. That is, they always agree on their value in every row in the result. Sometimes we are able to substitute a column by its equivalent column in a query block without changing the result of the query block. For example, we can always perform the substitution in the SELECT clause. We cannot always perform the substitution in the WHERE clause since it may change a predicate to a true constant. A typical source of equivalent columns is an equi-join. After the equi-join, the values of the join columns are always the same in every row. Therefore, in any operation after the join, e.g., selection, one column can be substituted by another.

Example 5 *:* Assume that we have the following query generated by some automatic tool:

```
SELECT L_ORDERKEY, MIN(O_ORDERDATE)
FROM LINEITEM, ORDERS
WHERE L_ORDERKEY = O_ORDERKEY AND
        O_ORDERDATE = L_SHIPDATE
GROUP BY L_ORDERKEY
```

Since the aggregation column is O_ORDERDATE, only ORDERS can be pushed past the join based on our previous algorithms. However, this query cannot be transformed into a query which first performs group-by on ORDERS and then the join. Based on the predicate O_ORDERDATE = L_SHIPDATE in the WHERE clause, we can first rewrite the query as

```
SELECT L_ORDERKEY, MIN(L_SHIPDATE)
FROM LINEITEM, ORDERS
WHERE L_ORDERKEY = O_ORDERKEY AND
        O_ORDERDATE = L_SHIPDATE
GROUP BY L_ORDERKEY
```

This query can then be rewritten as

```
CREATE VIEW V(V_ORDERKEY,V_SHIPDATE, V_MIN_SHIPDATE)
SELECT L_ORDERKEY, L_SHIPDATE, MIN(L_SHIPDATE)
FROM LINEITEM
GROUP BY L_ORDERKEY, L_SHIPDATE

SELECT L_ORDERKEY, V_MIN_SHIPDATE
FROM LINEITEM, V
WHERE V_ORDERKEY = O_ORDERKEY AND
        O_ORDERDATE = V_SHIPDATE
```

according to our algorithms.

In the original query, since O_ORDERKEY $\longrightarrow$ O_ORDERDATE in the ORDERS table, after the join, L_ORDERKEY $\longrightarrow$ O_ORDERDATE. Then, in each group based on L_ORDERKEY, the values of O_ORDERDATE are the same, and so are the values of L_SHIPDATE due to the join predicate. Therefore, in the original query, the grouping on L_ORDERKEY generates the same groups as grouping on L_ORDERKEY and L_SHIPDATE.

The user may actually have submitted the original query as

```
SELECT L_ORDERKEY, L_SHIPDATE
FROM LINEITEM, ORDERS
WHERE L_ORDERKEY = O_ORDERKEY AND
        O_ORDERDATE = L_SHIPDATE
GROUP BY L_ORDERKEY, L_SHIPDATE
```

Based on the above analysis, the original query is more likely to perform better because it has fewer grouping column and thus may require one less sort.

Since Algorithm **Find Partitionings** and Algorithm **FindClosure** have taken column equivalence into account for grouping columns and join columns, we only need to perform column substitution on the aggregation columns(AA). Therefore, we can first perform column substitution on AA, then call Algorithm **Find Partitioning** for each query generated to obtain a set of validly transformed queries.

10 Examples

Example 2: (continued)

We apply Algorithm **Find Partitionings** to the original query in the following steps. The number(s) in front of each statement represent(s) the line number(s) in the algorithm.

7: Since the set of the grouping columns is {`S_SUPPKEY`, `L_ORDERKEY`, `O_TOTALPRICE`}, its closure is S = {`S_SUPPKEY`, `O_ORDERKEY`, `L_SUPPKEY`, `L_ORDERKEY`, all `ORDERS` columns, all `SUPPLIER` columns}.

8: Since AA = (`L_QUANTITY`, `L_EXTENDEDPRICE`), the table containing AA is $LINEITEM$. Since S contains the the key `S_SUPPKEY` for the `SUPPLIER` table, and the key `O_ORDERKEY` for the `ORDERS` table, no more tables have to be pushed down. Therefore, R_d = {`LINEITEM`}.

9: $R_u = R - R_d$ = {`SUPPLIER`, `ORDERS`}.

10: The join columns of R_d tables are: JA_d = {`L_SUPPKEY`, `L_ORDERKEY`}. Therefore, $JA_d \subseteq S$ holds, and we proceed into the **if** block.

11: The first valid partitioning is (R_d = {`LINEITEM`}, R_u = {`SUPPLIER`, `ORDERS`}). We then try to find more valid partitionings in the following steps.

12: There are two non-empty subsets of R_u: {`SUPPLIER`} and {`ORDERS`}.

13: Let R' = {`SUPPLIER`}. Then, $R'_u = R_u - R'$ = {`ORDERS`}, and $R'_d = R_d + R'$ = {`LINEITEM`, `SUPPLIER`}.

14: The join columns of R'_d tables are: JA'_d = {`L_SUPPKEY`, `L_ORDERKEY`, `S_SUPPKEY`}. Since $JA'_d \subseteq S$ holds, the next valid partitioning is (R'_d = {`LINEITEM`, `SUPPLIER`}, R'_u = {`ORDERS`}).

13: Let R' = {`ORDERS`}. Then, $R'_u = R_u - R'$ = {`SUPPLIER`}, and $R'_d = R_d + R'$ = {`LINEITEM`, `ORDERS`}.

14: The join columns of R'_d tables are: JA'_d = {`L_SUPPKEY`, `L_ORDERKEY`, `O_ORDERKEY`}. Since $JA'_d \subseteq S$ holds, the next valid partitioning is (R'_d = {`LINEITEM`, `ORDERS`}, R'_u = {`SUPPLIER`}).

Therefore, there are three different ways to push down the group-by.

To illustrate the transformation, we consider the option of pushing down LINEITEM and SUPPLIER. Clearly, $GA_d^+ = \{$L_ORDERKEY, S_SUPPKEY$\}$, $C_d = \{$S_SUPPKEY = L_SUPPKEY AND S_ACCTBAL $> 0\}$; P_d is empty; and H_d is SUM(L_QUANTITY$*$L_EXTENDEDPRICE) $>$ 7300000. Therefore, the view can be defined as:

```
CREATE VIEW LINESUPP (V_ORDERKEY, V_SUPPKEY, V_TOTALVALUE) AS
(
  SELECT L_ORDERKEY, L_SUPPKEY, SUM(L_QUANTITY*L_EXTENDEDPRICE)
  FROM SUPPLIERS, LINEITEM
  WHERE S_SUPPKEY = L_SUPPKEY AND S_ACCTBAL > 0
  GROUP BY L_ORDERKEY, S_SUPPKEY
  HAVING SUM(L_QUANTITY*L_EXTENDEDPRICE)> 7300000
);
```

and the join after the group-by is

```
SELECT V_SUPPKEY, V_ORDERKEY, O_TOTALPRICE, V_TOTALVALUE
FROM LINEVALUE, ORDERS, SUPPLIERS
WHERE S_SUPPKEY=V_SUPPKEY AND S_ACCTBAL > 0 AND V_ORDERKEY= O_ORDERKEY;
```

11 When Is the Transformation Beneficial?

The best way for the optimizer to choose whether to perform group-by first or later is to cost the two alternative plans. However, when costing the two plans is too expensive, or at the stage of deciding whether or not to perform group-by first or later the cost function is unavailable, or there is no cost function at all in the optimizer, we may need some heuristic to decide whether to perform group-by first or later. The following list some observation regarding the effect of the transformation:

- Group-by push down cannot increase the input cardinality of the join.
- Group-by push down may increase or decrease the input cardinality of the group-by operation. This depends on the selectivity of the join.
- Group-by push down may restrict the choice of join orders. We first have to perform all joins required to create R_d so we can perform the grouping. However, the join order of

R_d with members of R_u is not restricted. On the other hand, group-by pull up provides additional choices for join ordering. However, if we decide whether to perform the transformation as part of the join enumeration process(using dynamic programming), the join order is not restricted by group-by push down.

- In a distributed database, group-by push down may reduce the communication cost. Instead of transferring all of R_d to some other site to be joined with R_u, we transfer only one row for each group of R_u. Since communication costs often dominate the query processing cost, this may reduce the overall cost significantly.

- When group-by is pushed down, the resulting table will normally be sorted based on the grouping columns. This fact can be exploited to reduce the cost of subsequent joins.

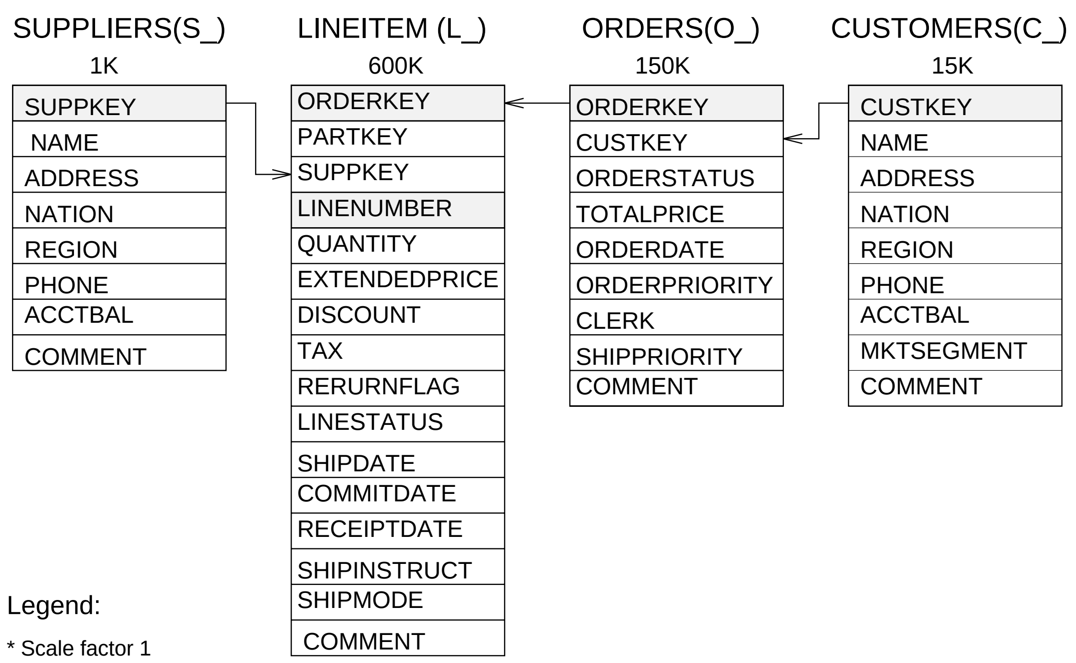
- When a `HAVING` predicates is very selective, group-by push down is a good choice since it reduce the input cardinality of the join. On the other hand, when the join predicates are very selective, or the predicates on the pulled up tables are very selective, it is more likely that group-by pull up would perform better since it can save most of the grouping effort.

12 Summary

We proposed a new strategy for processing `SQL` queries containing group-by, namely, interchanging the order of grouping and join. 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. `HAVING` clause is also considered in our algorithms. Since testing the full conditions may be expensive or even impossible, a fast algorithm was designed that tests a simpler, sufficient condition. We also designed an efficient algorithm for finding all valid partitionings of the tables into two sets, the pushed down tables and the pulled up tables.

A The TPCD Database

TPCD is a Decision support benchmark proposed by the the *Transaction Processing Performance Council(TPC)*. It is a suite of business oriented queries to be executed against a database that allows continuous access as well as concurrent updates[11]. The size of the database is scalable adjusted by a *scale factor*. The scale factor for a 100MB database is 0.1. Figure 9 shows the subset of the TPCD database we used through out this paper. The size of the database we used through out this paper is 100MB. We ran the queries in this paper on Oracle V7 and DB2/6000 V1 on an IBM RISC/6000 machine using the AIX operating system.



Legend:

* Scale factor 1

* The parentheses following each table name contain the prefix of the column names for that table.

* The highlighted column names in each table form the primary key of each table.

* The arrows are pointing in the direction of one-to-many relationships between tables.

* The number below the table name represents the number of rows (cardinality) of the table.

* Table SUPPLIERS has an index on (S_SUPPKEY),Table ORDERS has an index on (O_ORDERKEY)
Table CUSTOMERS has an index on (C_CUSTKEY)
Table LINEITEM has an index on (L_ORDERKEY,L_SUPPKEY), an index on (L_ORDERKEY) and an index on (L_PARTKEY)

Figure 9: Subset of the TPCD Database (Scale Factor 1)

Acknowledgements

The authors gratefully acknowledge financial support for this work from the Natural Science and Engineering Research Council and IBM Canada Ltd..

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] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 354–366, Santiago, Chile, September 1994.

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

[4] 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.

[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] 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.

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

[8] 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.

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

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

[11] Francois Raab, editor. *TPC Benchmark(tm) D (Decision Support), Working Draft 9.1*. Transaction Processing Performance Council, Administered by Shanley Public Relations, San Jose CA, 95112-6311, USA, February 1995.

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

[13] 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.

[14] Weipeng P. Yan. Query optimization techniques for aggregation queries. Research Proposal(unpublished), University of Waterloo, April 1994.

[15] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, pages 89–100, Houston, Texas, February 1994. IEEE Computer Society Press.