View Matching for Outer-Join Views

Per-Åke Larson Jingren Zhou
{palarson, jrzhou}@microsoft.com

August 2005

Technical Report
MSR-TR-2005-78

Prior work on computing queries from materialized views has focused on views defined by expressions consisting of selection, projection, and inner joins, with an optional aggregation on top (SPJG views). This paper provides the first view matching algorithm for views that may also contain outer joins (SPOJG views). The algorithm relies on a normal form for SPOJ expressions and is not based on bottom-up syntactic matching of expressions. It handles any combination of inner and outer joins, deals correctly with SQL bag semantics and exploits not-null constraints, uniqueness constraints and foreign key constraints.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Appropriately selected materialized views can speed up query processing greatly but only if the query optimizer can determine whether a query or part of query can be computed from existing materialized views. This is the view matching problem. Most work on view matching has focused on views defined by expressions consisting of selection, projection, and inner joins, possibly with a single group-by on top (SPJG views). In this paper we introduce the first view matching algorithm for views where some of the joins may be outer joins (SPOJG views).

The simplest approach to view matching is syntactic; essentially bottom-up matching of the operator trees of query and view expressions. However, algorithms of this type are easily fooled by expressions that are logically equivalent but syntactically different. A more robust approach is based on logical equivalence of expressions, which requires converting the expressions into a common normal form. SPJ expressions can be converted to a normal form consisting of a Cartesian product of all operand tables, followed by a selection and projection. More recently, Galindo-Legaria [5] showed that SPOJ expressions also have a normal form, called join-disjunctive normal form, which is the basis for our algorithm.

Example 1. Suppose we create the view shown below against tables in the TPC-R database.

```
create view oj_view as
select o_orderkey, o_custkey, l_linenumber,
  l_quantity, l_extendedprice, p_partkey,
  p_name, p_brand, p_retailprice
from part left outer join
  (orders left outer join lineitem
    on (l_orderkey=o_orderkey))
  on (p_partkey=l_partkey)
```

The following query asks for total quantity sold for each part with $partkey < 100$, including parts with no sales. Can this query be computed from the view?

```
select p_partkey, p_name, sum(l_quantity)
from (select * from parts where p_partkey < 100) p
  left outer join lineitem
  on (l_partkey=p_partkey)
group by p_partkey, p_name
```

The two expressions look very different but the query can in fact be computed from the view. The join between *Orders* and *Lineitem* will retain all *Lineitem* tuples because the join matches a foreign key declared between *l_orderkey* and *o_orderkey*. If the *Orders* table contains some orders without matching *Lineitem* tuples, they would occur in the result null-extended on all *Lineitem* columns. The outer join with *Part* will retain all real $\{Lineitem, Order\}$ tuples because this join is also a foreign-key join but it will eliminate all tuples that are null-extended on *Lineitem* columns. *Part* tuples that did not join with anything will also be retained in the result because the join is an outer join. Hence, the view will contain one complete tuple for each *Lineitem* tuple and also some *Part* tuples null-extended on columns from *Lineitem* and *Orders*. Hence, the view contains all required tuples and that the query can be computed from the view as follows.

```
select p_partkey, p_name, sum(l_quantity)
from oj_view
where p_partkey < 100
group by p_partkey, p_name
```

Now consider the following query. Can this query be computed from the view?

```
select o_orderkey, l_linenumber, l_quantity
from orders left outer join lineitem
  on (l_orderkey=o_orderkey)
```

The answer is no. The constraints defined on the TPC-R database allow *Orders* tuples without matching *Lineitem* tuples. If such an orphaned *Orders* tuple occurs in the database, it will be retained in the query result because the join is an outer join. It will also be retained in the result of the first join of the view because it is null-extended on all *Lineitem* columns, but it will be eliminated by the predicate of the second join of the view. Hence, the orphaned *Orders* tuple will not occur in the result of the view and the query cannot be computed from the view.

The rest of the paper is organized as follows. Section 2 introduces the notation used in the rest of the paper. In Section 3, we describe the join-disjunctive form of outer-join expressions and give an algorithm for computing the normal form. Section 4 shows how to determine containment of SPOJ expressions. We describe when and how the required tuples can be extracted from a SPOJ view in Section 5. Section 6 ties it all together by showing how to determine whether a SPOJ expression can be computed from a SPOJ view and how to construct the substitute expression. Aggregation views are discussed in Section 8. Initial experimental results are presented in Section 9. Finally, we survey related work in Section 10 and conclude in Section 11.

2 Definitions and Notations

The selection operator will be denoted in the normal way as σ_p where p is a predicate. Projection (without

duplicate elimination) will be denoted by π_c where c is a list of columns. We use the notation π_c^{null} to denote projection with null substitution. This operator outputs the columns in c unchanged but substitutes null for all other columns of its input. Borrowing from SQL, we also use the shorthand $T.*$ where T is a single table or a set of tables. $T.*$ denotes all columns of table(s) T that are available in the input.

The group-by (aggregation) operator is denoted by γ_G^A where G is a set of grouping expressions, normally just columns, and A is a set of aggregation expressions. The operator outputs all expressions in G and A. We also need an operator that removes duplicates (similar to SQL's `select distinct`), which we denote by δ.

A predicate p referencing some set $\mathcal{S}$ of columns is said to be *strong* or *null-rejecting* if it evaluates to false or unknown as soon as one of the columns in $\mathcal{S}$ is null. We will also use a special predicate $null(T)$ that evaluates to true if a tuple is null-extended on table T. $null(T)$ and $\sim null(T)$can be implemented in SQL as "$T.c$ is null" and "$T.c$ is not null", respectively, where c is any non-nullable column of T. The notation $null(\mathcal{T})$, where $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$, is a shorthand for $null(T_1) \wedge \ldots \wedge null(T_n)$ and $\overline{null}(\mathcal{T})$ is a shorthand for $\sim null(T_1) \wedge \ldots \wedge \sim null(T_n)$.

A schema $\mathcal{S}$ is a set of attributes (column names). Let T_1 and T_2 be tables with schemas $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. The *outer union*, denoted by $T_1 \uplus T_2$, first null-extends (pads with nulls) the tuples of each operand to schema $\mathcal{S}_1 \cup \mathcal{S}_2$ and then takes the union of the results (without duplicate elimination). Outer union has lower precedence than join.

A tuple t_1 is said to *subsume* a tuple t_2 if they are defined on the same schema, t_1 agrees with t_2 on all columns where they both are non-null, and t_1 contains fewer null values than t_2. The operator *removal of subsumed tuples* of T, denoted by $T\downarrow$, returns the tuples of T that are not subsumed by any other tuple in T.

The *minimum union*$(\oplus)$ of tables T_1 and T_2 is defined as $T_1 \oplus T_2 = (T_1 \uplus T_2)\downarrow$. Minimum union has lower precedence than join. It can be shown that minimum union is both commutative and associative.

Let T_1 and T_2 be tables with disjoint schemas $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, and p a predicate referencing some subset of the columns in $(\mathcal{S}_1 \cup \mathcal{S}_2)$. The *(inner) join* of the tables is defined as $T_1 \bowtie_p T_2 = \{(t_1, t_2) | t_1 \in T_1, t_2 \in T_2, p\ (t_1, t_2)\}$. The *left outer join* can then be defined as $T_1 \ltimes_p T_2 = (T_1 \bowtie_p T_2) \oplus T_1$. The *right outer join* is $T_1 \rtimes_p T_2 = T_2 \ltimes_p T_1$. The *full outer join* is $T_1 \times_p T_2 = (T_1 \bowtie_p T_2) \oplus T_1 \oplus T_2$.

We assume that base tables contain no subsumed tuples. This is usually the case in practice because base tables typically contain a unique key. We also assume that predicates are null-rejecting on all columns that they reference.

3 Join-Disjunctive Normal Form

To reason about equivalence and containment of SPOJ expressions we convert them into the join-disjunctive normal form introduced by Galindo-Legaria [5]. We extend Galindo-Legaria's definition of join-disjunctive normal form by allowing selection operators and incorporating the effects of primary keys and foreign keys. In addition, we provide an algorithm to compute the normal form.

We introduce the idea of join-disjunctive normal form by an example. Throughout this paper we will use the following database, modeled on the tables *Customer*, *Orders*, *Lineitem* of the TPC-R database.

```
C(ck, cn, cnk),
O(ok, ock, od, otp),
L(lok, ln, lpk, lq, lp)
```

Nulls are not allowed for any of the columns. Underlined columns form the primary key of each table. Two foreign key constraints are defined: $O.ock$ references $C.ck$ and $L.lok$ references $O.ok$.

Example 2. Suppose we have the following query

$$Q = C \rtimes_{ck=ock} (O \rtimes_{ok=lok} (\sigma_{lp>50K} L)).$$

The result will contain tuples of three types.

1. COL tuples, that is, tuples formed by concatenating a tuple from C, a tuple from O and a tuple from L. There will be one COL tuple for every L tuple that satisfies the predicate $lp > 50K$.
2. CO tuples, that is, tuples composed by concatenation a tuple from C, a tuple from O and nulls for all columns of L. There will be one such tuple for every O tuple that does not join with any L tuple satisfying $lp > 50K$.
3. C tuples, that is, tuples composed of a tuple from C with nulls for all columns of O and L. There will be one such tuple for every C tuple that does not join with any tuple in O.

The result contains all tuples of $C \bowtie_{ck=ock} O \bowtie_{ok=lok} (\sigma_{lp>50K} L)$, all tuples of $C \bowtie_{ck=ock} O$, and also all tuples in C. Each of the three sub-results is represented in the result in a minimal way. For example, if a tuple $(c_1, null, null)$ appears in the result, then there exists a tuple c_1 in C but there is no tuple o_1 in O such that (c_1, o_1) appears in $C \bowtie_{ck=ock} O$.

We can rewrite the expression as the minimum union of three join terms comprised solely of inner joins, which is the join-disjunctive form of the original SPOJ expression.

$$Q = (C \bowtie_{ck=ock} O \bowtie_{ok=lok} (\sigma_{lp>50K} L)) \oplus (C \bowtie_{ck=ock} O) \oplus (C)$$

3.1 Transformation Rules

The following transformation rules are used for converting SPOJ expression to join-disjunctive form.

$$T_1 \rightarrow_p T_2 = T_1 \bowtie_p T_2 \oplus T_1; \quad \text{if } T_1 = T_1{\downarrow} \text{ and } T_2 = T_2{\downarrow} \tag{1}$$

$$T_1 \times_p T_2 = T_1 \bowtie_p T_2 \oplus T_1 \oplus T_2; \quad \text{if } T_1 = T_1{\downarrow} \text{ and } T_2 = T_2{\downarrow} \tag{2}$$

$$(T_1 \oplus T_2) \bowtie_p T_3 = T_1 \bowtie_p T3 \oplus T_2 \bowtie_p T3; \quad \text{if } T_3 = T_3{\downarrow} \tag{3}$$

$$\sigma_{p(1)}(T_1 \bowtie_p T_2 \oplus T_2) = (\sigma_{p(1)}T_1) \bowtie_p T_2; \quad \text{if } p(1) \text{ is strong and references only } T_1 \tag{4}$$

$$\sigma_{p(1)}(T_1 \bowtie_p T_2 \oplus T_1) = (\sigma_{p(1)}T_1) \bowtie_p T_2 \oplus (\sigma_{p(1)}T_1); \quad \text{if } p(1) \text{ references only } T_1 \tag{5}$$

The proofs of the correctness of the first three transformation rules can be found in [5]. The fourth rule follows from the observation that all tuples originating from the term T_2 in $(T_1 \bowtie_p T_2 \oplus T_2)$ will be null-extended on all columns of T_1. All those tuples will be discarded if $p(1)$ is strong on T_1. The last rule follows from the obvious rule $\sigma_{p(1)}(T_1 \rightarrow_p T_2) = (\sigma_{p(1)}T_1) \rightarrow_p T_2$ by expanding the two outer joins.

Example 3. This example illustrates conversion of a SPOJ expression using the above rules. $p(i,j)$ denotes a predicate that references columns in tables T_i and T_j.

$$\begin{aligned}
&(\sigma_{p(1)}(T_1 \ltimes_{p(1,2)} T_2)) \times_{p(2,3)} T_3 \\
&= (\sigma_{p(1)}(T_1 \bowtie_{p(1,2)} T_2 \oplus T_2)) \times_{p(2,3)} T_3 && \text{by rule (1)} \\
&= ((\sigma_{p(1)}T_1) \bowtie_{p(1,2)} T_2)) \times_{p(2,3)} T_3 && \text{by rule (4)} \\
&= (T_1 \bowtie_{p(1)\wedge p(1,2)} T_2) \times_{p(2,3)} T_3 \\
&\quad \text{by including selection predicate in join} \\
&= (T_1 \bowtie_{p(1)\wedge p(1,2)} T_2 \bowtie_{p(2,3)} T_3) \oplus \\
&\quad (T_1 \bowtie_{p(1)\wedge p(1,2)} T_2) \oplus T_3 && \text{by rule (2)}
\end{aligned}$$

3.2 Join-Disjunctive Normal Form

In this section, we show that a SPOJ expression can always be converted to join-disjunctive form and that two SPOJ expressions are equivalent if they have the same join-disjunctive form. The main theorem is due to Galindo-Legaria [5] but our proofs are different and slightly more general.

Lemma 1. *Let Q_1 and Q_2 be SPOJ expressions in join-disjunctive form. Then the expressions $\sigma_p(Q_1)$, $Q_1 \bowtie_p Q_2$, $Q_1 \rightarrow_p Q_2$, and $Q_1 \times_p Q_2$ can all be rewritten in join-disjunctive form.*

Proof. We assume that expression Q_1 operates on tables in the set $\mathcal{T}_1$. We write Q_1 in the form $\sigma_{p11}(\mathcal{T}_{11}) \oplus \sigma_{p12}(\mathcal{T}_{12}) \oplus \ldots \oplus \sigma_{p1n}(\mathcal{T}_{1n})$ where $\mathcal{T}_{11}, \mathcal{T}_{12}, \ldots, \mathcal{T}_{1n}$ are subsets of $\mathcal{T}_1$. The notation $\sigma_{p1i}(\mathcal{T}_{1i})$ means a selection with predicate p_{1i} over the Cartesian product of the tables in $\mathcal{T}_{1i}$, that is, the normal form of a SPJ expression. Q_2 is expressed in the same way but over the base set $\mathcal{T}_2$ and containing m terms.

For the case $\sigma_p(Q_1)$, first reorder the SPJ terms of Q_1 into two groups. Reordering is allowed because minimum union is commutative. The first group contains all terms that are null-extended on at least one table referenced by predicate p and the second group contains all remaining terms. By rule (4), all terms in the first group will be completely eliminated by the selection. By rule (5), the selection can be applied separately to each term in the second group. Hence, the result is in join-disjunctive form.

For the case $Q_1 \bowtie_p Q_2$, repeated application of rule (3) converts the expression into

$$\sigma_{p11}(\mathcal{T}_{11}) \bowtie_p \sigma_{p21}(\mathcal{T}_{21}) \oplus \sigma_{p12}(\mathcal{T}_{12}) \bowtie_p \sigma_{p21}(\mathcal{T}_{21}) \oplus \ldots \oplus \sigma_{p1n}(\mathcal{T}_{1n}) \bowtie_p \sigma_{p2m}(\mathcal{T}_{2m})$$

In essence, we are *multiplying* the two input expressions producing an output expression containing nm terms. This expression can be converted into $\sigma_{p11\wedge p21\wedge p}(\mathcal{T}_{11} \cup \mathcal{T}_{21}) \oplus \sigma_{p12\wedge p21\wedge p}(\mathcal{T}_{12} \cup \mathcal{T}_{21}) \oplus \ldots \oplus \sigma_{p1n\wedge p2m\wedge p}(\mathcal{T}_{1n} \cup \mathcal{T}_{2m})$, which is in join-disjunctive form. The result may actually contain fewer than nm terms. Suppose predicate p references tables in a subset $\mathcal{S}$ of the tables in $\mathcal{T}_1 \cup \mathcal{T}_2$. Because p is strong on $\mathcal{S}$, each term that is null-extended on one or more tables in $\mathcal{S}$, i.e. $\mathcal{S} \not\subset (\mathcal{T}_{1i} \cup \mathcal{T}_{2j})$, will return an empty result when applying predicate p.

The outerjoin cases, $Q_1 \rightarrow_p Q_2$ and $Q_1 \times_p Q_2$, follow immediately from the join case by first applying rule (1) or (2), respectively. □

Lemma 2. *Let $\sigma_{p1}(\mathcal{T}_1)$ and $\sigma_{p2}(\mathcal{T}_2)$ be two terms in the join-disjunctive form of a SPOJ expression. If $\mathcal{T}_1 \subset \mathcal{T}_2$, then $p_2 \Rightarrow p_1$.*

Proof. Because the two terms are part of the same SPOJ expression, the term $\sigma_{p2}(\mathcal{T}_2)$ must have been created by joining in the additional tables $\mathcal{T}_2 - \mathcal{T}_1$ to the term $\sigma_{p1}(\mathcal{T}_1)$ through some sequence of joins, including possibly also some selects. Suppose the predicates of these joins and selections are $q_1, q_2, \ldots, q_n$. Consequently, p_2 must be of the form $p_2 = p_1 \wedge q_1 \wedge \ldots \wedge q_n$. Any tuple that satisfies $p_1 \wedge q_1 \wedge \ldots \wedge q_n$ must also satisfy p_1. □

Theorem 1 (Galindo-Legaria). *The join-disjunctive form of a SPOJ expression Q is a normal form for Q.*

Proof. To prove the theorem we must show that two SPOJ expressions Q_1 and Q_2 produce the same result for all database instances if and only if their join-disjunctive forms are equivalent. We denote their join-disjunctive forms by Q_1' and Q_2', respectively.

Each term in the join-disjunctive form of an expression produces tuples with a unique null-extension pattern. Suppose the complete set of operand tables for the expression is $\mathcal{T}$. A term in the join-disjunctive form is defined over a subset $\mathcal{S}$ of $\mathcal{T}$ and hence produces tuples that are null extended on $\mathcal{T} - \mathcal{S}$. No two terms operate on the same set of tables so the sub-results produced by different terms have unique null-extension patterns. It follows that to prove equivalency of two join-disjunctive forms we only need to prove pair-wise equivalency of terms that are defined over the same set of tables.

Suppose $Q_1' = Q_2'$. Clearly, Q_1' and Q_2' produce the same result for all database instances. Because $Q_1 = Q_1'$, Q_1 produces the same result as Q_1' for all database instances and similarly for the pair Q_2 and Q_2'. It follows that Q_1 and Q_2 produce the same result for all database instances, that is, $Q_1 = Q_2$.

Now suppose $Q_1' \neq Q_2'$. To prove the theorem we must show that $Q_1 \neq Q_2$. If $Q_1' \neq Q_2'$ then there must exist a term $\sigma_{p1}(\mathcal{S})$ in one of the expression Q_1' or Q_2' that is not equivalent to any term in the other expression. Suppose the other expression contains a term $\sigma_{p2}(\mathcal{S})$ defined over the same set of tables. If it does not, we are done because we only need to consider pair-wise equivalence. If the terms are not equivalent then $p_1 \neq p_2$. We construct a database instance D consisting of one tuple $t_i \in T_i$ for each table in T_i in $\mathcal{T}$ such that the tuple $t = \{t_1, t_2, \ldots, t_n\}$ satisfies p_1 but not p_2 (or vice versa). This is always possible given that the predicates are not equivalent.

Evaluating Q_1 on this database instance will produce one tuple t'. t' will be produced either by the term $\sigma_{p1}(\mathcal{S})$ or a term defined on a superset of $\mathcal{S}$. In other words, t' will be null-extended on, at most, the tables in $\mathcal{T} - \mathcal{S}$. However, evaluating Q_2 on the same instance cannot produce the same result. It follows from Lemma 2 that, because t does not satisfy predicate p_2, t cannot satisfy the predicate of any term in Q_2' that is defined on $\mathcal{S}$ or a superset of $\mathcal{S}$. Hence, the result of Q_2 cannot contain a tuple with the same null-extension pattern as t'. The result will either be empty or contain a tuple t'' with a different null-extension pattern. Consequently, $Q_1 \neq Q_2$. □

These lemmas and the theorem imply that deciding equivalence of two SPOJ expressions can be reduced to the well-understood problem of deciding equivalence of SPJ terms with matching source tables in the join-disjunctive forms of the expressions. If there are constraints on the database, two SPOJ expressions may still be equivalent even if their normal forms differ, because they may not produce different results on any valid database instances. The same is true for the normal form of SPJ expressions.

3.3 Computing the Normal Form

Theorem 1 guarantees that every SPOJ expression has a unique normal form but we also need an algorithm for it. Lemma 1 and its proof provide the basis for an algorithm. It shows how to construct an output expression in normal form from inputs in normal form. Hence, we can compute the normal form of an expression by traversing its operator tree bottom-up.

The algorithm exploits transformation rule (4) to discard terms that are eliminated by null-rejecting predicates. Additional terms can be eliminated by exploiting foreign keys. A term $\sigma_{p1}(\mathcal{T}_1)$ can be eliminated from the normal form if there exists another term $\sigma_{p2}(\mathcal{T}_2)$ such that $\mathcal{T}_1 \subset \mathcal{T}_2$ and $\sigma_{p1}(\mathcal{T}_1) \subseteq \pi_{\mathcal{T}_1.*}\sigma_{p2}(\mathcal{T}_2)$. This may happen if the additional tables $(\mathcal{T}_2 - \mathcal{T}_1)$ in $\sigma_{p2}(\mathcal{T}_2)$ are joined in through foreign key joins. This is an important simplification because, in practice, most joins correspond to foreign keys. Since terms are SPJ expressions, establishing whether the subset relationship holds is precisely the containment problem for SPJ expression. The containment testing algorithm in [8] can be used for this purpose.

We now have all the pieces needed to design an algorithm for computing the normal form of a SPOJ expression. Algorithm 1, recursively applies rules (1) - (3) bottom-up to expand joins and simplifies the resulting expressions by applying rules (4) and (5) and the containment rule described above. It returns a set of terms ($TermSet$) corresponding to the normal form of the input expression. Each term is represented by a structure consisting of a set of tables ($Tables$) and a predicate ($Pred$).

The parameter $Flag$ determines whether or not to discard terms that are completely subsumed. For reasons that will become clear later, we normally wish to discard subsumed terms when normalizing a query expression but not when normalizing a view expression.

Example 4. We compute the normal form of the view (with $Flag = false$)

$$V = C \rtimes_{ock=ck} (O \times_{ok=lok} (\sigma_{lp<20} L)))$$

The algorithm recursively descends the operator tree. When applied to the innermost join, it produces

$$V = C \rtimes_{ock=ck} (\sigma_{lp<20 \wedge ok=lok}(O, L) \oplus \sigma_{lp<20} L \oplus O)$$

Algorithm 1: Normalize(E, $Flag$)

Input: *Expression* E, *Boolean* $Flag$
Output: $TermSet$
/* A term represents a SPJ expression and consists */
/* of a set of tables and a predicate. */
$Node$ = top node of E;
switch *type of Node.Operator* **do**
 case *base table* R*:*
  $TermSet\ BT = \{\{R\}, true\}$;
  return BT;
 /* Select has an input expression IE and a predicate SP */
 case `select` *operator* (IE, SP)*:*
  $TermSet\ IT = Normalize(IE)$;
  foreach $Term\ t$ *in* IT **do**
   if SP *rejects nulls on a table not in* $t.Tables$ **then**
    $IT = IT - \{t\}$; /*apply rule (4) */
   else
    $t.Pred = t.Pred \wedge SP$; /*apply rule (5) */
  end
  return IT;
 /* Join has two input expressions (LE, RE), */
 /* a predicate (JP) and a join type */
 case `join` *operator* $(LE, RE, JP, JoinType)$*:*
  $TermSet\ LT = Normalize(LE)$;
  $TermSet\ RT = Normalize(RE)$;
  $TermSet\ JT = \emptyset$; /*terms after join */
  $TermSet\ EL = \emptyset$; /*terms eliminated by subsumption */
  /* Multiply the two input sets (rule (3)) */
  foreach $Term\ l \in LT$ **do**
   foreach $Term\ r \in RT$ **do**
    $Term\ t = \{(l.Tables \cup r.Tables), l.Pred \wedge r.Pred \wedge JP\}$;
    /* Apply rule (3) to eliminate terms */
    if *!(*JP *rejects nulls on a table not in* $t.Tables$*)* **then**
     $JT = JT \cup \{t\}$;
     /* Check whether all tuples in input term are sub- */
     /* sumed by the result term by testing containment */
     /* of SPJ expressions, see algorithm in [8]. */
     if $Flag \wedge \sigma_{l.Pred}(l.Tables) \subseteq \sigma_{t.Pred}(t.Tables)$ **then**
      $EL = EL \cup \{l\}$;
     end
     if $Flag \wedge \sigma_{r.Pred}(r.Tables) \subseteq \sigma_{t.Pred}(t.Tables)$ **then**
      $EL = EL \cup \{r\}$;
     end
    end
   end
  end
  /* Add inputs from preserved side(s) (rules (1) and (2)) */
  switch $JoinType$ **do**
   case *full outer:*
    $JT = JT \cup LT \cup RT$; break;
   case *left outer:*
    $JT = JT \cup LT$; break;
   case *right outer:*
    $JT = JT \cup RT$; break;
  end
  /* Discard terms eliminated by subsumption */
  $JT = JT - EL$;
  return JT
end

Next, the algorithm is applied to the left outer join and produces the normal form.

$$V = \sigma_{lp<20 \wedge ok=lok \wedge ock=ck}(C, O, L) \oplus \sigma_{ck=ock}(C, O) \oplus C$$

The term $\sigma_{lp<20 \wedge ock=ck}(C, L)$ was eliminated because the predicate $ock = ck$ is null-rejecting on O and O is not a member of (C, L).

3.4 The Subsumption Graph

The minimum union operators in the normal form are required because a term may produce redundant tuples, that is, tuples that are subsumed by tuples produced by other terms. However, the subsumption patterns are not arbitrary as shown in this section.

Each term in the join-disjunctive form of a SPOJ expression produces tuples with a unique null-extension pattern. Suppose the complete set of operand tables for the expression is $\mathcal{U}$. A term in the join-disjunctive form is defined over a subset $\mathcal{T}$ of $\mathcal{U}$ and hence produces tuples that are null extended on $\mathcal{U} - \mathcal{T}$.

A tuple produced by a term with source table set $\mathcal{T}$ can only be subsumed by tuples produced by terms whose source table set is a superset of $\mathcal{T}$. The subsumption relationships among terms can be modeled by a DAG (directed acyclic graph), which we call the *subsumption graph* of the SPOJ expression. It follows immediately from the definition that the graph has a single root (maximal node) with source set $\mathcal{U}$.

Definition 3.1. *Let* $E = E_1 \oplus \cdots \oplus E_n$ *be the join-disjunctive form of an SPOJ expression. The* subsumption graph *of* E *contains a node* n_i *for each term* E_i *in the normal form and the node is labeled with the source table set* $\mathcal{T}_i$ *of* E_i*. There is an edge from node* n_i *to node* n_j*, if* $\mathcal{T}_i$ *is a minimal superset of* $\mathcal{T}_j$*.* $\mathcal{T}_i$ *is a minimal superset of* $\mathcal{T}_j$ *if there does not exist a node* n_k *in the graph such that* $\mathcal{T}_j \subset \mathcal{T}_k \subset \mathcal{T}_i$*.*

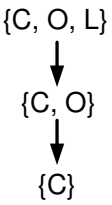

Figure 1: Subsumption graph for view V

Figure 1 shows an example of a subsumption graph. The graph for view V is extremely simple but that is not always the case.

Because of the one-to-one correspondence between terms and nodes in the subsumption graph, we take the liberty of referring to parents, ancestors and children of a term instead of the more precise but cumbersome phrase "the terms corresponding to the parent nodes of the node corresponding to the current term".

The following lemma shows that if a tuple of term is subsumed by any tuple, it is also subsumed by a tuple in one of its parent terms.

Lemma 3. *Let t be a tuple produced by a term E_i in the join-disjunctive form of an SPOJ expression. If t is subsumed, then t is subsumed by some tuple produced by a parent term of E_i.*

Proof. Denote the source table set of E_i by $\mathcal{S}_i$, its selection predicate by q_i, and its set of output columns by c_i. We need only consider ancestors of E_i because those are exactly the terms with source table sets that are supersets of $\mathcal{S}_i$.

Suppose t is subsumed by a tuple t_a. t_a must be from some ancestor E_a of E_i, that is, $\mathcal{S}_i \subset \mathcal{S}_a$. If E_a is a parent of E_i, we are done. If E_a is not a parent of E_i, it is reachable through one or more of the parents of E_i, $E_{p_1}, E_{p_2}, \cdots, E_{p_k}$. Without loss of generality, assume that E_a is reachable through E_{p_1}. The source tables set of E_{p_1} satisfies the relationship $\mathcal{S}_i \subset \mathcal{S}_{p_1} \subset \mathcal{S}_a$. By Lemma 2, the predicates of the terms then satisfy the relationship $q_a \Rightarrow q_{p_1} \Rightarrow q_i$. t_a is produced by term E_a so it satisfies predicate q_a. But t_a also satisfies predicate q_{p_1} because $q_a \Rightarrow q_{p_1}$. It follows that the tuple $t_{p_1} = \pi_{c_{p_1}}(t_a)$ is produced by term E_{p_1}. t_{p_1} also subsumes t because $c_i \subseteq c_{p_1}$ and $\pi_{c_i} t_{p_1} = t$. Tuple t_{p_1} subsumes t and is produced by a parent node, which proves the lemma. □

4 Containment of SPOJ Expressions

When computing a query from a view, the issue arises as to what operations one is willing to apply to the view. In the context of SPJ views, the operations are normally restricted to selection, projection and duplicate elimination so that each result tuple is computed from a single view tuple. When restricted to this set of operations, a query cannot be computed from the view unless the query is contained in the view.

We retain the same restriction in the context of SPOJ views, namely, we consider only transformations where a result tuple is computed from a single view tuple, but with a slight generalization. We also allow null substitution, that is, changing a column value to null. Given this generalization, we need a way to decide whether a view contains "enough" tuples.

Definition 4.1. *Let T_1 and T_2 be two tables with the same schema. T_1 is subsumption-contained in T_2, denoted by $T_1 \subset_s T_2$, if for every tuple $t_1 \in T_1$ there exists a tuple $t_2 \in T_2$ such that $t_1 = t_2$ or t_1 is subsumed by t_2. An expression Q_1 is subsumption-contained in an expression Q_2 if the result of Q_1 is subsumption-contained in the result of Q_2 for every valid database instance.*

Lemma 4. *Let Q_1 and Q_2 be two SPOJ expressions. If $Q_1 \not\subset_s Q_2$ then Q_1 cannot be computed from the result of Q_2 using a combination of selection, projection, null substitution, and removal of subsumed tuples for every database instance.*

Proof. The proof is straightforward. If $Q_1 \not\subset_s Q_2$ then, for some database instance, there exists a tuple t_1 in the result of Q_1 that is not subsumed by any tuple in the result of Q_2. It is obvious that no combination of selection, projection, null substitution, and removal of subsumed tuples applied to the result of Q_2 can generate t_1. □

The following theorem reduces the problem of testing containment of SPOJ expressions to the known problem of testing containment of SPJ expressions. This can be done using, for example, the containment testing algorithm in [8].

Theorem 2. *Let Q_1 and Q_2 be two SPOJ expressions on a database with no constraints and Q_1' and Q_2' their join-disjunctive forms, resfpectively. Then $Q_1 \subset_s Q_2$ if and only if the following condition holds: for every term $\sigma_{p_1}(\mathcal{S})$ in Q_1', there exists a matching term $\sigma_{p_2}(\mathcal{S})$ in Q_2' such that $\sigma_{p_1}(\mathcal{S}) \subset \sigma_{p_2}(\mathcal{S})$.*

Proof. To prove sufficiency, assume that the condition holds. If $\sigma_{p_1}(\mathcal{S}) \subset \sigma_{p_2}(\mathcal{S})$ holds for a pair of terms, then trivially $\sigma_{p_1}(\mathcal{S}) \subset_s \sigma_{p_2}(\mathcal{S})$ also holds. Because this holds for every term of Q_1, it immediately follows that $Q_1 \subset_s Q_2$.

To prove necessity, assume that the condition does not hold, that is, there exists a term $\sigma_{p_1}(\mathcal{S})$ in Q_1' with the property that no term in Q_2' satisfies the required conditions. There are two cases to consider: a) Q_2' does not contain a term with base $\mathcal{S}$ and b) Q_2' does contain a matching term $\sigma_{p_2}(\mathcal{S})$ but $\sigma_{p_1}(\mathcal{S}) \not\subset \sigma_{p_2}(\mathcal{S})$

To prove case a) we construct a database instance containing one tuple for each table in $\mathcal{S}$ such that their concatenation t satisfies predicate p_1. All other tables are empty. As there are no constraints on the database this is a valid database instance.

The result of Q_1 then contains the single tuple t. However, the result of Q_2 is either empty or contains tuples that are subsumed by but not subsuming t. Any term in Q_2' defined over a set containing a table T not in $\mathcal{S}$ must produce an empty result because T is empty. If a term Q_2' contains only tables in $\mathcal{S}$, it is defined over some subset $\mathcal{S}'$ of $\mathcal{S}$. Hence, it outputs either no tuples or tuple t null-extended on $\mathcal{S} - \mathcal{S}'$. The output tuple contains more nulls than t so it cannot subsume t. As none of the possible output tuples of Q_2 subsume t, $Q_1 \not\subset_s Q_2$ for this database instance.

Case b) can be proven in a similar way. We construct a database instance in the same way but add the requirement that t not satisfy predicate p_2 so that

t is not output by term $\sigma_{p_2}(\mathcal{S})$. Such a tuple must exist because $\sigma_{p_1}(\mathcal{S}) \not\subset \sigma_{p_2}(\mathcal{S})$. The rest of the argument is the same as for case a). □

The theorem assumes that there are no constraints on the database. If there are constraints, the conditions are still sufficient but they may not be necessary. In particular, foreign-key constraints may make it possible for a term $\sigma_{p_1}(\mathcal{S})$ to be contained by a term $\sigma_{p_2}(\mathcal{T})$ even when $\mathcal{S} \subset \mathcal{T}$.

A view term with extra tables must satisfy certain criteria to be worth considering. The additional tables, $\mathcal{T} - \mathcal{S}$, in the view must all be joined in through equijoin predicates matching foreign-key constraints and not have any further restrictions. We can compute the term as $\sigma_{p_2}(\mathcal{T}) = \sigma_{p_2'}(\mathcal{S}) \bowtie_{p_2''} \sigma(\mathcal{T} - \mathcal{S})$ where $p_2 = p_2' \wedge p_2''$. That is, we first join the tables $\mathcal{S}$ matching the query term and then join in the additional tables of the view term. Because p_2'' contains only foreign-key joins, every tuple in $\sigma_{p_2'}(\mathcal{S})$ is guaranteed to join with at least one tuple in $\sigma(\mathcal{T} - \mathcal{S})$. How to verify that the criteria are satisfied and to test containment for this case is described in [8].)

The normal form of the view may contain several terms such that $\mathcal{S} \subset \mathcal{T}$. If follows from Lemma 2 that we only need to consider a term $\sigma_{p_2}(\mathcal{T})$ if T is a minimal superset of S, that is, there are no other terms $\sigma_{p_3}(\mathcal{T}')$ in the view such that $\mathcal{S} \subset \mathcal{T}' \subset \mathcal{T}$. Normally, there is at most one such minimal term in the view but if there are more than one, each minimal term should be checked for containment. Candidate minimal terms can easily be found from the view's subsumption graph.

5 Recovering All Tuples of a Term

The result tuples of a term in the normal form of an SPOJ view are implicitly contained in the result of the view. A tuple t of a term $\sigma_{p1}(\mathcal{S}_1)$ may occur explicitly in the result of the view or it may be subsumed by another tuple t' generated by a wider term $\sigma_{p2}(\mathcal{S}_2)$, i.e. a term with the property $\mathcal{S}_1 \subset \mathcal{S}_2$. In fact, there may be many tuples in the result that subsume t. Suppose we have a query term $\sigma_{p3}(\mathcal{S}_1)$ and we have shown that all tuples needed by the query term are contained in the view term $\sigma_{p1}(\mathcal{S}_1)$. To compute the query from the view, we first *recover* the result of $\sigma_{p1}(\mathcal{S}_1)$ from the view result. The following example illustrates the steps necessary.

Example 5. Consider the following view.

$$\begin{aligned} V &= (\sigma_{cn<5} C) \rtimes_{ock=ck} (O \rtimes_{ok=lok} (\sigma_{lp<20} L)) \\ &= \sigma_{cn<5 \wedge lp<20 \wedge ok=lok \wedge ock=ck}(C, O, L) \oplus \\ &\quad \sigma_{cn<5 \wedge ck=ock}(C, O) \oplus \sigma_{cn<5} C \end{aligned}$$

Its normal form shows that the view consists of three types of tuples: COL tuples without null extension, CO tuples null extended on L, and C tuples null extended on O and L. Suppose we want to recover the tuples generated by the term $\sigma_{cn<5 \wedge ck=ock}(C, O)$. All the desired tuples are composed of a real C tuple and a real O tuple, i.e. they are not null-extended on C and O. We first apply the selection $\sigma_{\sim null(C) \wedge \sim null(O)} V$ to eliminate all tuples that do not satisfy this requirement. The selection can be simplified to $\sigma_{\sim null(O)} V$ because no tuples of V are null-extended on C.

The predicate $\sim null(C)$ can be implemented in SQL as "*C.col* is not null" where *col* is any C column guaranteed to be non-null in the result of $\sigma_{cn<5 \wedge ck=ock}(C, O)$. A column is guaranteed to be non-null if it is either declared with `not null` or occurs in a null-rejecting predicate. In our case, we can use *cn* or *ck* because of the predicate $(cn < 5 \wedge ck = ock)$.

We also have to make sure that we get tuples with the correct duplication factor. A CO tuple (t_c, t_o) that satisfies the predicate $(cn < 5 \wedge ck = ock)$ may have joined with one or more L tuples. Hence, if we simply project V onto the columns of C and O without duplicate elimination, the result may contain multiple duplicates of tuple (t_c, t_o) and the result is not correct according to SQL bag semantics. Duplicate elimination will eliminate all such duplicates, but it may also remove legitimate duplicates. It will work correctly only if the result of $\sigma_{cn<5 \wedge ck=ock}(C, O)$ has a unique key. In our case, *ok* is a unique key for the term so we can safely apply duplicate elimination. Consequently, we can recover the result of the term from the view as follows

$$\sigma_{cn<5 \wedge ck=ock}(C, O) = \delta(\pi_{C.*, O.*}(\sigma_{ck \neq null} V))$$

The following two theorems show how to recover the tuples of a SPJ term from a SPOJ view. Theorem 3 deals with the case when duplicate elimination is needed and Theorem 4 states under what conditions duplicate elimination is not necessary

Theorem 3. *Let $\sigma_P(\mathcal{R})$ be an SPJ term of a view V. If $\sigma_P(\mathcal{R})$ outputs a unique key, then $\sigma_P(\mathcal{R}) = \delta(\pi_{\mathcal{R}.*} \sigma_{\overline{null}(\mathcal{R})} V)$.*

Proof. The expression $\pi_{\mathcal{R}.*} \sigma_{\overline{null}(\mathcal{R})} V$ selects all tuples of the right form, that is, tuples not null-extended on any tables in $\mathcal{R}$, and projects them on the correct set of tables. Consider a tuple t in $\sigma_P(\mathcal{R})$. Because $\sigma_P(\mathcal{R})$

outputs a unique key, the result of $\sigma_P(\mathcal{R})$ does not contain any duplicates of t. We know that one or more copies of t must exist in $\pi_{\mathcal{R}.*}\sigma_{\overline{null}(\mathcal{R})}V$. After duplicate elimination, a single copy of t remains. It follows that the two expressions are equal. □

Next we consider how to recover the tuples of a SPJ term when a key is not available. In this case, we cannot apply duplicate elimination.

Example 6. Consider the following view.

$$\begin{aligned} V &= (\sigma_{lp<20}O) \rtimes_{ock=ck} (\sigma_{cn<5}C) \\ &= \sigma_{cn<5 \wedge ock=ck \wedge lp<20}(C, O) \oplus \sigma_{lp<20}O \end{aligned}$$

Suppose that a unique key of O is not available in the view output. If so, can we still recover the term $\sigma_{lp<20}O$ from the view? The answer is yes. We cannot apply duplicate elimination but it is not needed. Consider a tuple t_o in the result of $\sigma_{lp<20}O$. The tuple may not join with any tuple in $\sigma_{cn<5}C$, in which case it will occur once in the view result (null extended on C). If the tuple joins with a tuple t_c, the combined tuple (t_o, t_c) will occur in the view result. However, because the join condition $ock = ck$ corresponds to a foreign key constraint, we know that it cannot join with more than one C tuple. In other words, every tuple in $\sigma_{lp<20}O$ will occur exactly once in the view result. Hence, no duplicate elimination is needed and the tuples can be recovered by $\sigma_{lp<20}O = \pi_{O.*}\sigma_{\sim null(O)}V$.

The example illustrates a case with a single extension join. An extension join is an equijoin matching a foreign key constraint where the foreign key columns are declared non-null and reference a unique key. An extension join merely extends each input tuple with a additional columns. Reference [8] introduced the notion of the hub of a SPJ expression and gave a procedure for computing the hub. The hub of a term $\sigma_P(\mathcal{R})$ is the smallest subset $\mathcal{S}$ of $\mathcal{R}$ such that every table in $\mathcal{R}-\mathcal{S}$ is joined in through a sequence of extension joins. The following theorem shows how to exploit this idea to recover additional terms. [1]

Theorem 4. *Let $\sigma_P(\mathcal{R})$ be an SPJ term of a view V. Then $\sigma_P(\mathcal{R}) = \pi_{\mathcal{R}.*}\sigma_{\overline{null}(\mathcal{R})}V$ if every term $\sigma_q(\mathcal{T})$ in the normal form of V such that $\mathcal{R} \subset \mathcal{T}$, has a hub equal to the hub of $\sigma_P(\mathcal{R})$.*

Note that the condition is trivially satisfied for the maximal term of V (the term with the maximal set of tables) because there are no terms with a larger set of tables.

Proof. Consider a tuple t in $\sigma_P(\mathcal{R})$. The tuple may be represented in V either explicitly (null extended on tables not in $\mathcal{R}$) or it may have been subsumed by another tuple originating from a term $\sigma_q(\mathcal{T})$ such that $\mathcal{R} \subset \mathcal{T}$. Because the hub of $\sigma_q(\mathcal{T})$ is equal to the hub of $\sigma_P(\mathcal{R})$, the joins bringing in the extra tables $\mathcal{T} - \mathcal{R}$ are extension joins and create at most one tuple subsuming t. All terms that could generate tuples subsuming t have the same hub as $\sigma_P(\mathcal{R})$. Consequently, there is a unique tuple t' in V that subsumes t or t is represented explicitly in V. Hence, t will occur exactly once in the result of $\pi_{\mathcal{R}.*}\sigma_{\overline{null}(\mathcal{R})}V$ and no duplicate elimination is needed. □

So far we have assumed that the view outputs at least one non-null column for every table in $\mathcal{R}$. We now relax this assumption and consider what can be done if the view outputs a non-null column for only a subset of the tables. The following theorem states under what conditions we can still correctly extract the desired tuples.

Theorem 5. *Let $\sigma_P(\mathcal{R})$ be an SPJ term of a view V and $\mathcal{S}$ a subset of $\mathcal{R}$ such that the view outputs at least one non-null column for each table in $\mathcal{S}$. Then $\sigma_{\overline{null}(\mathcal{R})}V = \sigma_{\overline{null}(\mathcal{S})}V$ if, for every term $\sigma_q(\mathcal{T})$ in the normal form of V such that $\mathcal{T} \subset \mathcal{R}$, the set $(\mathcal{R}-\mathcal{T})\cap\mathcal{S}$ is non-empty.*

Proof. The purpose of the predicate $\overline{null}(\mathcal{R})$ is to reject all tuples that are null-extended on any table of $\mathcal{R}$, that is, tuples originating from any term $\sigma_q(\mathcal{T})$ where $\mathcal{T} \subset \mathcal{R}$. Tuples originating from a term $\sigma_q(\mathcal{T})$ with $\mathcal{T} \subset \mathcal{R}$ will be null-extended on tables in $(\mathcal{R} - \mathcal{T})$. If $\mathcal{S}$ overlaps with $(\mathcal{R} - \mathcal{T})$ then the reduced predicate $\overline{null}(\mathcal{S})$ will reject all tuples originating from $\sigma_q(\mathcal{T})$. Consequently, if the condition holds for every term with $\mathcal{T} \subset \mathcal{R}$, the reduced predicate will reject exactly the same tuples as the original predicate. □

6 Computing a Query from a View

We now have the main tools needed to decide whether a SPOJ query can be computed from a SPOJ view. This section pulls them together into a decision procedure and describes how to construct the substitute expression. Here are the high-level steps of the view matching algorithm; the steps are described in more detail in separate sections.

Algorithm SPOJ-View-Matching:

[1] The conditions in the theorem are stricter than absolutely necessary. It suffices that the additional tables, $\mathcal{T} - \mathcal{S}$, of the view term $\sigma_{p_i}(\mathcal{T})$ are joined in through equijoins against unique keys. In equijoin of tables R and S using a unique key of S, each tuple of R can join with at most one tuple of S so no additional duplicates are needed. This is true even if a selection has been applied to S.

1. Convert both the query Q and the view V to join-disjunctive normal form.
2. Check whether Q is subsumption-contained in V.
3. Check whether all terms in Q can be recovered from V.
4. Determine residual predicates, that is, query predicates that must be applied to the view.
5. Check whether all columns required by residual predicates and output expressions are available in the view output.
6. If the view passes all tests above, construct the substitute expression.

We will illustrate the algorithm using the following view and query.

$$\begin{aligned} V_1 =& \pi_{lok,ln,lq,lp,ok,od,otp,ck,cn,cnk}(\sigma_{cnk<10}(C) \\ & \ltimes_{ock=ck} (\sigma_{otp>50}(O) \times_{ok=lok} \sigma_{lq<100}(L))) \\ Q_1 =& \pi_{lok,lq,lp,od,otp}(\sigma_{otp>150}(O) \ltimes_{ok=lok} \sigma_{lq<100}(L)) \end{aligned}$$

6.1 Converting to Normal Form

Conversion to join-disjunctive normal form is simply a matter of applying algorithm *Normalize* described in Section 3.3. Applying the algorithm to our example view (with $Flag = false$) and query (with $Flag = true$) produces the following expressions.

$$\begin{aligned} V_1 =& \pi_{lok,ln,lq,lp,ok,od,otp,ck,cn,cnk} \\ & (\sigma_{cnk<10 \wedge ck=ock \wedge otp>50 \wedge ok=lok \wedge lq<100}(C,O,L) \oplus \\ & \sigma_{cnk<10 \wedge ck=ock \wedge otp>50}(C,O) \oplus \sigma_{otp>50}(O) \oplus \\ & \sigma_{otp>50 \wedge ok=lok \wedge lq<100}(O,L) \oplus \sigma_{lq<100}(L)) \\ Q_1 =& \pi_{lok,lq,lp,od,otp}(\sigma_{otp>150 \wedge ok=lok \wedge lq<100}(O,L) \oplus \\ & \sigma_{lq<100}(L)) \end{aligned}$$

6.2 Checking Containment

To check that the view contains all tuples required by the query we check containment of each term of the query (Theorem 2). That is, for every term $\sigma_{p1}(\mathcal{S})$ in the query, we try to find a term $\sigma_{p2}(\mathcal{T})$ in the view such that $\mathcal{S} \subseteq \mathcal{T}$ and $p_1 \Rightarrow p_2$.

The query term with base (O, L) has the same base as the fourth term in the view. To ensure containment the following condition must hold

$$\begin{aligned} (otp > 150 \ \wedge \ ok = lok \ \wedge \ lq < 100) \Rightarrow \\ (otp > 50 \ \wedge \ ok = lok \wedge \ lq < 100). \end{aligned}$$

The condition can be simplified to $(otp > 150) \Rightarrow (otp > 50)$, which trivially holds. Hence, the view contains all tuples required by the first term.

The second term of the query matches the last term of the view. In this case, the condition equals $(lq < 100) \Rightarrow (lq < 100)$, which of course holds. Hence, all tuples required by this term of the query are contained in the view. We conclude that the view contains all tuples required by the query.

6.3 Checking Recovery

Checking whether the tuples of a term can be recovered from the view consists of the following steps:

1. Check whether duplicate elimination is required by comparing hubs (Theorem 4).
2. If duplicate elimination is required, find a unique key of the term (Theorem 3) and check whether the view outputs the required columns.
3. Check whether the view outputs sufficient non-null columns (Theorem 5).

Our example view references tables C, O, and L and outputs at least one non-null column from each table. We can use $C.ck$, $O.otp$, and $L.ok$ as non-null columns. $C.ck$ is a primary key and as such must be non-null, $O.otp$ and $L.lq$ are referenced by null-rejecting predicates.

The first term of the query matches the fourth term of the view. The hub of the fourth term of the view is $\{L\}$ because the join between L and O matches a foreign key constraint and the foreign key column $L.lok$ is declared non-null. The COL term (the first term) of the view is the only term whose base is a superset of $\{O, L\}$. The hub of the COL term is also $\{L\}$ because the join between O and C is also a foreign key join. Hence, the conditions of Theorem 4 are satisfied and no duplicate elimination is needed.

The fourth term of the view references tables O and L, so we have $\mathcal{R} = \{O, L\}$ and $\mathcal{S} = \{O, L\}$. The third and the fifth terms of the view have bases that are subsets of $\mathcal{R}$. The third term has base $\mathcal{T} = \{O\}$. Consequently, the set $(\mathcal{R} - \mathcal{T}) \cap \mathcal{S} = (\{O, L\} - \{O\}) \cap \{O, L\} = \{L\}$ is non-empty. The fifth term has base $\mathcal{T} = \{L\}$ and, again, the set $(\mathcal{R} - \mathcal{T}) \cap \mathcal{S} = (\{O, L\} - \{L\}) \cap \{O, L\} = \{O\}$ is non-empty. If follows that we can extract the tuples of the fourth term of the view using the predicate $O.otp \neq null$ and $L.lq \neq null$.

The second term of the query matches the last term of the view. The hub of the last term is obviously $\{L\}$. We already determined that the hub of the first and the fourth terms of the view is also $\{L\}$. Those are the only terms whose base is a superset of $\{L\}$. Consequently, no duplicate elimination is required for this term either.

The last term of the view has base $\{L\}$. The view does not contain any terms whose base is a subset of $\{L\}$ so the conditions of Theorem 5 are automatically satisfied. If follows that we can extract the tuples of this term using the predicate $L.lq \neq null$.

We have thus determined that the required tuples can be extracted from the view as follows:

$$\sigma_{otp>50 \wedge ok=lok \wedge lq<100}(O, L) = \sigma_{otp\neq null \wedge lq \neq null} V$$
$$\sigma_{lq<100}(L) = \sigma_{lq \neq null} V$$

6.4 Residual Predicates

Query predicates may be more restrictive than the view predicates. We must eliminate all tuples that do not satisfy the query predicate but the view may not output all the necessary columns. Fortunately, we may not need to apply the complete query predicate; parts of the predicate that are already enforced by the view predicate can be eliminated. In addition, we can exploit equivalences among columns in the view result.

Suppose we have a query term with predicate $P_q = p_1 \wedge p_2 \wedge \ldots \wedge p_n$ (in conjunctive normal form) and a corresponding view term with predicate P_v. A conjunct p_i of the query predicate can be eliminated if $P_v \Rightarrow p_i$, that is, if p_i already holds for all tuples generated by the appropriate term in the view. The implication can be tested using, for example, the subsumption algorithm described in [8].

Applying this to the first term of our example query, we get the following three implications:

$$(otp > 50 \wedge ok = lok \wedge lq < 100) \Rightarrow (otp > 150)$$
$$(otp > 50 \wedge ok = lok \wedge lq < 100) \Rightarrow (ok = lok)$$
$$(otp > 50 \wedge ok = lok \wedge lq < 100) \Rightarrow (lq < 100)$$

It is easy to see that second and third implication hold but the first one does not. Hence, the residual predicate for the first term is $(otp > 150)$.

For the second term we get the implication $(lq < 100) \Rightarrow (lq < 100)$ which trivially holds. Hence, no further predicate needs to be applied for the second term.

6.5 Availability of Output Columns

Before proceeding further we need to discuss how to exploit column equivalences. A column equivalence class is a set of columns that are known to have the same value in all tuples produced by an expression. Equivalence classes are generated by column equality predicates, typically equijoin conditions. A straightforward algorithm for computing equivalence classes is provided in [8].

A SPOJ expression consists of multiple SPJ terms, each one with its own equivalence classes. Once we have recovered the tuples generated by a term of a view, we can safely exploit its equivalence classes in residual predicates applied to that term. Applying the residual predicates may create new column equivalences that should be added to the term's equivalence classes. These updated equivalence classes can then be exploited in output expressions and also when creating grouping columns (covered in Section 8.2).

For our example view, only three terms have non-trivial equivalence classes: $\{\{ck, ock\}, \{ok, lok\}\}$ for the first term, $\{\{ck, ock\}\}$ for the second term, and $\{\{ok, lok\}\}$ for the fourth term. For the query, only the first term has a non-trivial equivalence class, namely, $\{\{ok, lok\}\}$.

We are now ready to check whether all required columns are available. The columns available in the view output are *lok*, *ln*, *lq*, *lp*, *ok*, *od*, *otp*, *ck*, *cn*, and *cnk*. The first query term has one residual predicate: $(otp > 150)$. *otp* is a view output column so the predicate can be applied. The second query term required has no residual predicates. The query output columns are *lok*, *lq*, *lp*, *od*, *otp*, which are all available as view output columns. Hence, all required columns are available.

6.6 Constructing the Substitute Expression

Once we reach this stage, we know that the query can be computed from the view. All that remains is to construct the substitute expression, i.e. an expression that computes the query from the view. This consists of applying the following steps to each SPJ term of the query and combining the resulting expressions with minimum union operators ($\oplus$).

1. Recover the SPJ term from the view using a selection with the appropriate $\sim null$ predicates constructed earlier. Apply duplicate elimination if needed.
2. Restrict the result using a selection with the appropriate residual predicates, if any. Exploit view equivalence classes as needed.
3. Apply projection (without duplicate elimination) to reduce the result to the required output columns. Exploit query equivalence classes as needed. Return null for any output column originating from a table not in the base of the term.

For our example query and view, this process produces the following result.

$$\begin{aligned} Q_1 &= \pi_{lok,lq,lp,od,otp}[\sigma_{otp>150}(\sigma_{otp\neq null \wedge lq \neq null} V_1) \oplus \\ &\qquad \pi^{null}_{lok,ln,lq,lp}(\sigma_{lq \neq null} V_1)] \\ &= \pi_{lok,lq,lp,od,otp}[\sigma_{otp>150 \wedge otp\neq null \wedge lq \neq null} V_1 \oplus \\ &\qquad \pi^{null}_{lok,ln,lq,lp}(\sigma_{lq \neq null} V_1)] \end{aligned}$$

The innermost selection of each term performs the recovery. As determined earlier, no duplicate elimination is required. The selection $\sigma_{otp>150}$ applies the residual predicate needed for the first term. The second term has no residual predicate but needs a null-substituting projection, $\pi^{null}_{lok,ln,lq,lp}$, that retains all available columns of L and substitutes all other columns with null. The final projection reduces the tuples to the desired output columns. The first term can be simplified by combining the two selections.

7 Efficient Substitute Expressions

Unfortunately, the substitute expressions described in the previous section cannot be evaluated directly because no commercial database system supports minimum union. In this section we show how to eliminate minimum unions from substitute expressions and replace them by regular unions. After this conversion, the number of scans of the view can often be reduced by combining terms. In many cases, but not always, only a single scan is necessary.

Example 7. Let's analyze what tuples remain in the result of Q_1 after the minimum union has been applied.

Any tuple t_1 of V_1 that satisfies the predicate of the first SPJ term is retained in the result because this term is not subsumed by any other terms. However, any tuple t_2 that satisfies the predicate of the second SPJ term should be retained only if it does not qualify for the first term.

Suppose t_2 satisfies the predicates of both terms. Then the first term outputs the tuple $s = (t_2.lok, t_2.ln, t_2.lq, t_2.lp, t_2.od, t_2.otp, t_2.ck, t_2.cn, t_2.cnk)$ and the second term outputs $s' = (t_2.lok, t_2.ln, t_2.lq, t_2.lp, null, null, null, null, null)$. The minimum union then eliminates s' because it is subsumed by s.

If t_2 satisfies the predicate of the second term but not the predicate of the first term, the subsuming tuple s is not generated and s' is not eliminated by the minimum union. Furthermore, no other tuple generated by the first term can subsume s'. Columns (lok, ln) is a unique key of V_1 so no other tuple with the key value $(t_2.lok, t_2.ln)$ exists in V_1.

We rewrite the substitute expression as

$$Q_1 = \pi_{lok,lq,lp,od,otp}[\sigma_{otp>150\wedge otp\neq null\wedge lq\neq null}V_1 \cup \pi^{null}_{lok,ln,lq,lp}(\sigma_{lq\neq null\wedge\sim(otp>150\wedge otp\neq null\wedge lq\neq null)}V_1)].$$

Each term of this expression outputs only tuples that will be retained in the result so the minimum union no longer has any effect and has been converted to a regular union.

This substitute expression requires two scans of the view but, in fact, only a single scan is needed. The predicates of the two terms are mutually exclusive so a tuple in the view cannot satisfy both predicates. Hence, a view tuple will contribute at most one tuple to the result and we can select all the required tuples by the expression

$$\sigma_{(otp>150\wedge otp\neq null\wedge lq\neq null)\vee(lq\neq null\wedge(otp\leq 150\vee otp=null))}V_1$$

All tuples of the second term, that is, tuples that satisfy the predicate $(lq \neq null\wedge(otp \leq 150\vee otp = null))$, should output nulls for all columns except lok, ln, lq, and lp. This can be done using the case statement of SQL. It has the following syntax

```
case when P1 then E1
     when P2 then E2
     ...
     when Pn then En
     else En+1
end
```

where $P_1, \cdots, P_n$ are predicates and $E_1, \cdots, E_{n+1}$ are scalar expressions. The predicates are evaluated in the order specified. If P_i is the first predicate to evaluate to true, the result of expression E_i is returned. If none of the predicates evaluates to true, the else branch is executed. To simplify the presentation, we generalize the case statement slightly and allow multiple expressions after "then" and "else".

Using the case statement, we can then write the substitute expression as follows

$$Q_1 = \pi_{lok,lq,lp,cst}\ \sigma_{(otp>150\wedge otp\neq null\wedge lq\neq null)\vee(lq\neq null\wedge(otp\leq 150\vee otp=null))}V_1$$

where

```
cst =
case when lq ≠ null ∧ (otp ≤ 150 ∨ otp = null)
     then null, null
     else od, otp
end
```

After simplification of the complex-looking predicate we obtain

$$Q_1 = \pi_{lok,lq,lp,cst}\ \sigma_{lq\neq null}V_1,$$

which can be evaluated using normal SQL operators and, furthermore, requires only a single scan of the view.

7.1 Minimum Union to Union

This section shows how to replace minimum unions in the substitute expression by regular unions.

7.1.1 Terms without duplicate elimination

In this section, we consider substitute terms without duplicate elimination, that is, terms of the form

$$E^s = \pi_c^{null}\ \sigma_p V$$

where V is the target materialized view, p is the term's selection predicate, and and c its projection list. All columns of V not in c are substituted by nulls. The selection predicate has the form $p = q \wedge \overline{null}(\mathcal{T}_i)$ where $\mathcal{T}_i$ is the term's source table set, $\overline{null}(\mathcal{T}_i)$ the recovery predicate, and q an optional residual predicate. Terms that require duplicate elimination are considered in the next section.

The selection part of E^s, $\sigma_p V$, determines which tuples of the view qualify for this query term. If a tuple t qualifies, its null-extended projection onto c, $\pi_c^{null} t$, will occur in the result of E^s.

We denote by $\hat{E}^s$ the *net contribution* of E^s, that is, the set of tuples in E^s that are not subsumed by any other tuples and hence will occur in the final result of the query (projected onto appropriate columns). The following theorem shows that we can compute $\hat{E}^s$ by selecting from V only those tuples that satisfy p but not the predicates of the parent terms.

Theorem 6. *Let $E_i^s = \pi_{c_i}^{null}\ \sigma_{p_i} V$ be a substitute expression for a term E_i of an SPOJ query E. The net contribution of E_i^s can be computed as*

$$\hat{E}_i^s = \pi_{c_i}^{null}\ \sigma_{p_i \wedge \sim p_{i_1} \wedge \sim p_{i_2} \wedge \cdots \wedge \sim p_{i_k}} V$$

where $p_{i_1}, p_{i_2}, \cdots, p_{i_k}$ are the selection predicates of the parent terms of E_i in the subsumption graph of E, provided that c_i contains a unique key of E_i^s.

Proof. We first prove that no tuples retained by the selection $\sigma_{p_i \wedge \sim p_{i_1} \wedge \sim p_{i_2} \wedge \cdots \wedge \sim p_{i_k}} V$ will be subsumed. Consider a tuple $t \in V$ that t satisfies the selection predicate above. t is output by E_i^s because it satisfies p_i. However, t is not output by any of the parent expressions $E_{i_j}^s$ of E_i^s because t does not satisfy the predicate, p_{i_j}, of any parent. According to Lemma 2, t does not satisfy the predicate of any ancestor of E_i^s either. Hence, the tuple $\pi_{c_i}^{null} t$ is not subsumed by a projection of t generated by some ancestor of E_i^s.

The only other possibility is that tuple $\pi_{c_i}^{null} t$ is subsumed by (the projection of) another tuple $t' \in E_i^s$. However, this is not possible because c_i contains a unique key of E_i^s so no other tuple in E_i^s can match $\pi_{c_i}^{null} t$.

We must also prove that all tuples eliminated by the selection $\sigma_{p_i \wedge \sim p_{i_1} \wedge \sim p_{i_2} \wedge \cdots \wedge \sim p_{i_k}} V$ would have been subsumed. Consider a tuple $t \in V$ that does not satisfy the predicate so it is eliminated from $\hat{E}_i^s$. If t is eliminated because it does not satisfy p_i, t is not output by E_i^s either so clearly it should not be output by $\hat{E}_i^s$. The other possibility is that t violates one of the parent predicates. Without loss of generality, assume that it violates $\sim p_{i_1}$. In other words, t satisfies predicate p_{i_1}, in which case, t is output by $E_{i_1}^s$. $c_i \subseteq c_{i_1}$ so $\pi_{c_{i_1}}^{null} t$ will subsume $\pi_{c_i}^{null} t$. This proves that all tuples that are eliminated would have been subsumed. □

7.1.2 Terms with duplicate elimination

Example 8. Consider the following query

$$\begin{aligned} Q_2 &= \pi_{ok,od,otp,ln,lq}(\sigma_{otp>50}(O) \ltimes_{ok=lok} \sigma_{lq<10}(L)) \\ &= \pi_{ok,od,otp,ln,lq}(\sigma_{otp>50}(O) \oplus \\ &\qquad \sigma_{otp>50 \wedge ok=lok \wedge lq<10}(O, L)) \end{aligned}$$

Following the procedure in previous sections we find that the query can be computed from view V_1. Duplicate elimination is needed to recover the O term but not for the OL term. The substitute expression is

$$\begin{aligned} Q_2 &= \pi_{ok,od,otp,ln,lq}(E_1 \oplus \sigma_{lq<10} E_2) \\ &\quad \textit{where} \\ E_1 &= \delta\ \pi_{ok,od,otp}^{null}(\sigma_{\sim null(O)} V_1) \\ E_2 &= \sigma_{\sim null(L) \wedge \sim null(O)} V_1 \end{aligned}$$

The residual predicates $lq < 10$ can be pushed down and combined with the recovery predicates, which produces the following expression.

$$\begin{aligned} Q_2 &= \pi_{ok,od,otp,ln,lq}(E_1 \oplus E_2') \quad \textit{where} \\ E_2' &= \sigma_{lq<10 \wedge \sim null(L) \wedge \sim null(O)} V_1 \end{aligned}$$

The minimum union is necessary because some tuples of E_1 may be subsumed by tuples produced by its parent term E_2'. The subsumed tuples have been eliminated in E_1' below. After this, the minimum union can be converted to a regular union, producing the following substitute expression.

$$\begin{aligned} Q_2 &= \pi_{ok,od,otp,ln,lq}(E_1' \uplus E_2') \\ E_1' &= \pi_{ok,od,otp}^{null}(\sigma_{cs2=0}\ \gamma_{ok,od,otp}^{cs2=sum(s2)} \\ &\qquad \pi_{ok,od,otp,s2}\ (\sigma_{\sim null(O)} V_1)) \end{aligned}$$

where

$s2 =$ case
when $lq < 10 \wedge \sim null(L) \wedge \sim null(O)$ then 1
else 0
end

The reason why expression E_1' contains no duplicate tuples is most easily understood by analyzing the expression step by step.

- The selection $\sigma_{\sim null(O)}V_1$ extracts from V all tuples that qualify for E'_1.
- The projection $\pi_{ok,od,otp,s2}$ projects them onto the O columns of the view and adds a new column $s2$ that indicates whether or not the tuple also qualifies for the parent term E'_2.
- The aggregation $\gamma^{cs2=sum(s2)}_{ok,od,otp}$ groups the tuples to eliminate duplicates. For each group, we count, in $cs2$, how many of the group's input tuples qualified for E'_2.
- If $cs2 > 0$, the group's output tuple is subsumed by at least one tuple in E'_2 and should be eliminated. This is done by the selection $\sigma_{cs2=0}$.
- The final projection just adds null values corresponding to columns ln and lq.

The following theorem shows how to compute the net contribution of a term for the case when duplicate elimination is required.

Theorem 7. *Let $E^s_i = \delta\ \pi^{null}_{c_i}\ \sigma_{p_i}V$ be a substitute expression for a term E_i of an SPOJ query E where E_i has table source set $\mathcal{T}_i$. The net contribution of E_i can then be computed as*

$$\hat{E}^s_i = \pi^{null}_{c_i}\ \sigma_{csp=0}\ \gamma^{csp=sum(x)}_{c_i}(\sigma_{p_i}V)$$

where x = (case when $p_{i_1} \vee p_{i_2} \vee \cdots \vee p_{i_k}$ then 1 else 0 end) and $p_{i_1}, p_{i_2}, \cdots, p_{i_k}$ are the selection predicates of the parents of E_i in the subsumption graph of E.

Proof. Consider a tuple t in V that satisfies p_i and hence $t' = \pi^{null}_{c_i}t \in E^s$.

Suppose that t' is subsumed by a tuple s' produced by some ancestor term E^s_a of E^s_i. s' is the projection of a tuple $s \in V$, that is, $s' = \pi^{null}_{c_a}s$. Because s' subsumes t', they must agree on columns c_i, that is, $\pi_{c_i}t = \pi_{c_i}s$. We must show that t will not be output by the expression above.

Tuple s satisfies the selection predicate p_a of the term E^s_a and because E^s_a is an ancestor of E^s_i, s must also satisfy predicate p_i (follows from Lemma 2). Because E^s_a is an ancestor of E^s_i, it must be reachable through one of the parent terms of E^s_i. Without loss of generality, assume that it is reachable through the first parent, $E^s_{i_1}$. In that case, tuple s also satisfies predicate p_{i_1} so when evaluated on s, the case statement returns 1. Because $\pi_{c_i}t = \pi_{c_i}s$, tuples s and t are included in the same group g by the group-by operator. Because s is included in group g, the aggregate column csp is non-zero and the result tuple for group g is eliminated by the selection predicate $csp = 0$. Hence, no trace of tuple t is left in $\hat{E}^s_i$.

Now suppose t' is not subsumed by any tuple. In this case, we must show that t' will be contained in $\hat{E}^s_i$.

Because t' is not subsumed, every tuple $s \in V, s \neq t$ that satisfies p_i must either differ from t' when projected onto c_i, that is, $\pi_{c_i}s \neq t'$, or satisfy none of the parent predicates $p_{i_j}, j = 1, \cdots, k$. The tuples with the first property, that is, those that satisfy $\pi_{c_i}s \neq t'$, do not belong to the same group g as tuple t and hence do not affect the sum of group g. For tuples that satisfy none of the parent predicates, the case statement returns zero so they do not increase the sum of group g. We have shown that the sum of group g is zero, which means that it satisfies the selection predicate $csp = 0$. Hence, tuple t' is retained in $\hat{E}^s_i$. □

7.2 Combining Terms to Reduce Scans

The theorems in the previous section show how to convert minimum unions to regular union. If the terms of the union are computed independently, each term requires a separate scan over the view. This is not necessary — terms can be combined in the same scan if their predicates are mutually exclusive.

Theorem 8. *Consider two terms E^s_i and E^s_j, $i \neq j$ of a substitute expression E with $\hat{E}^s_i = \pi^{null}_{c_i}\ \sigma_{q_i}V$ and $\hat{E}^s_j = \pi^{null}_{c_j}\ \sigma_{q_j}V$. If $q_i \wedge q_j = false$ then*

$$\hat{E}^s_i \cup \hat{E}^s_j = \pi^{null}_{c}\ \sigma_{q_i \vee q_j}V$$

where c = (case when q_i then c_i else c_j end).

Proof. If $q_i \wedge q_j = false$, then no tuple in the view can satisfy both predicates at the same time so the two predicates extract non-overlapping subsets from the view. The predicate $q_i \vee q_j$ correctly extracts the combined subsets in a single scan and the case statement outputs the correct set of columns depending on which predicate is satisfied. □

Corollary 1. *If one of the view terms corresponding to E^s_i and E^s_j, $i \neq j$, is an ancestor of the other view term, then $q_i \wedge q_j = false$.*

Proof. Obvious, because the predicate of the descendant substitution term explicitly checks (and rejects) any tuple that also satisfies the predicate of a parent term. □

This corollary gives additional insight into which terms can be combined, namely, any terms that are connected through a sequence of parent-child relationship. Nodes on a common path in the subsumption graph have this property and can thus be combined into the same scan. Any set of non-overlapping paths that together cover all nodes of the graph provides a

valid set of scans. There may be several such sets of paths, which raises the issue of finding the "optimal" set. One could generate several alternative substitute expressions, one for each set of paths, and rely on the query optimizer to select the optimal expression. This may be costly, however, so some form of heuristic solution may be preferred.

In the special case that the subsumption graph contains a single path, the query can be reduced to a single scan of the view.

Terms that require duplicate elimination can be handled by first constructing the substitute expressions for those terms, marking the corresponding nodes from the subsumption graph as already covered, and then covering the remaining nodes as outlined above.

8 Aggregation Views

We now turn to outer-join views with aggregation, that is, views defined by a SPOJ expression and a single group-by operation on top. Aggregation functions are limited to `sum` and `count` to keep the views incrementally maintainable. For aggregation views, we consider substitute expression composed of selection, projection, and aggregation, but disallow minimum union.

Aggregation views require three modifications of the view matching algorithm described in Section 6.

1. In steps one and two, the conversion to normal form and checking of containment is applied to the view and query expressions *without* aggregation (the SPOJ part).

2. Step three, checking recovery, changes significantly as described in details in Section 8.1 below.

3. An new step must also be added (after step four) to check whether further aggregation is needed and possible.

8.1 Tuple Recovery

For aggregation views, the procedure for recovering the required tuples consists of the following steps.

1. Check whether all terms required by the query have the correct duplication factor.
2. Check whether terms *not* required by the query can be eliminated.
3. Construct recovery predicates if required columns are available in the view output.

8.1.1 Correct duplication factor

Step one is necessary to ensure that `sum` and `count` aggregated columns will be correct. A query term may be mapped to a view term that includes additional source tables. These additional joins may change the duplication factor, that is, if the view term is projected onto the same tables as the query term, the result may not contain the correct number of duplicates of each row. If so, a `sum` or a `count` taken from the view would be incorrect. Two terms are guaranteed to produce rows with the same number of duplicates if they have the same hub [8], so this step boils down to computing the hub of each query term and its target view term and verifying that the two are equal. The notion of hubs and how to compute the hub of an SPJ expression are explained in [8].

8.1.2 Recovering required tuples

In a non-aggregation view, we recover the terms one by one but this is not always possible for aggregation views. Rows originating from different terms may belong to the same group. If so, they will be merged into the aggregates of the group's result row. Once the details have been lost through aggregation, it is no longer possible to tease apart the contributions from different terms. The following example illustrates the issue.

Example 9. Consider a view with an aggregate over a left outer join of C and O.

$$\begin{aligned} V_4 &= \gamma_{C.cn}^{count(*),sum(otp)}(C \rightarrowtriangle_p O) \\ &= \gamma_{C.cn}^{count(*),sum(otp)}(C \bowtie O \oplus C) \end{aligned}$$

View V_4 is grouped on $C.cn$. Because $C.cn$ is not a key of C, a row originating from the CO term may well have the same cn value as a row originating from the C term. The sum will then include rows from both terms and the contributions from each term cannot be separated.

$$\begin{aligned} V_5 &= \gamma_{C.cn,O.od}^{count(*),sum(otp)}(C \rightarrowtriangle_p O) \\ &= \gamma_{C.cn,O.od}^{count(*),sum(otp)}(C \bowtie O \oplus C) \end{aligned}$$

However, as soon as the grouping columns include a non-null column of O, as in V_5, the two terms can be separated. All rows originating from the C term are null on $O.od$ while all rows from the first term are non-null on $O.od$. Hence, two rows from different terms are guaranteed to end up in separate groups. Those from the CO term can be extracted by the predicate $O.od \neq null$ and those from the C term by the predicate $O.od = null$.

For this step, we first divide the terms of the view into two sets: terms required by the query and excess terms, that is, terms not required. The goal is to

eliminate the effect of the excess terms from the view result, retaining only the required terms. Suppose we have an aggregation view V that in normalized form equals

$$V = \gamma_C^A(\sigma_{p_1}(\mathcal{T}_1) \oplus \cdots \oplus \sigma_{p_n}(\mathcal{T}_n))$$

where C is a set of grouping columns and A is a sequence of aggregation functions. Suppose the query under consideration requires the first $k, k \leq n$ terms of the view. (We can always reorder terms so this is true.) We rewrite the view as

$$V = \gamma_C^A(V_R^{spj} \oplus V_E^{spj})$$

where $V_R^{spj} = \sigma_{p_1}(\mathcal{T}_1) \oplus \cdots \oplus \sigma_{p_n}(\mathcal{T}_k)$ and $V_E^{spj} = \sigma_{p_1}(\mathcal{T}_{k+1}) \oplus \cdots \oplus \sigma_{p_n}(\mathcal{T}_n)$. We would like to rewrite the expression in the following form and then discard the part of the view that originates from excess terms (the second part of the expression).

$$V = \gamma_C^A(V_R^{spj}) \cup \gamma_C^A(V_E^{spj})$$

Under what circumstances is this a valid rewrite? The following Lemma and Theorem derive sufficient conditions for the rewrite to be applicable.

Note that we have only the grouping columns of the view available to separate between tuples from required terms and tuples from excess terms.

Lemma 5. *Consider an aggregation view V over tables $\mathcal{U}$ and assume that the view outputs a non-null, non-aggregated column for every table in S, $\mathcal{S} \subseteq \mathcal{U}$. Let $\sigma_P(\mathcal{R})$ be an SPJ term of V that is required by a query Q. Then $\sigma_{\overline{null}(\mathcal{S} \cap \mathcal{R})} V \supseteq \sigma_P(\mathcal{R})$ and $\sigma_{\overline{null}(\mathcal{S} \cap \mathcal{R})} V$ contains no tuples from excess terms of V if, for every excess term $\sigma_q(\mathcal{T})$, the set $(\mathcal{U} - \mathcal{T}) \cap \mathcal{S} \cap \mathcal{R}$ is non-empty.*

Proof. We first show that $\sigma_P(\mathcal{R}) \subseteq \sigma_{\overline{null}(\mathcal{S} \cap \mathcal{R})} V$. The tuples in $\sigma_P(\mathcal{R})$ are not null-extended on any table in $\mathcal{R}$ and, hence, every tuple in $\sigma_P(\mathcal{R})$ satisfies the predicate $\overline{null}(\mathcal{R})$. Because $\mathcal{R} \supseteq \mathcal{S} \cap \mathcal{R}$, $\overline{null}(\mathcal{R}) \Rightarrow \overline{null}(\mathcal{S} \cap \mathcal{R})$, that is, every tuple that satisfies the stronger condition $\overline{null}(\mathcal{R})$ must also satisfy $\overline{null}(\mathcal{S} \cap \mathcal{R})$. Consequently, $\sigma_P(\mathcal{R}) \subseteq \sigma_{\overline{null}(\mathcal{S} \cap \mathcal{R})} V$.

To prove the second part of the lemma, consider an excess term $\sigma_q(\mathcal{T})$. The predicate $\overline{null}(\mathcal{S} \cap \mathcal{R})$ will reject all tuples of the term if they are null-extended on at least one table in $\mathcal{S} \cap \mathcal{R}$. All tuples of the term on null-extended on the tables in $(\mathcal{U} - \mathcal{T})$. It follows that they will be eliminated if $(\mathcal{U} - \mathcal{T})$ and $\mathcal{S} \cap \mathcal{R}$ have at least one table in common, which is the condition in the theorem. Hence, if the condition holds for all excess terms, the result of $\sigma_{\overline{null}(\mathcal{S} \cap \mathcal{R})}$ cannot contain tuples from any excess term. □

Theorem 9. *Let $V_R^{spj} = \oplus_{i=1}^n \sigma_{p_i}(\mathcal{R}_i), i = 1, 2, \cdots, n$ be the set of terms of view $V = \gamma_C^A(V^{spj})$ required by a query Q. If every term in V_R^{spj} satisfies the conditions in Lemma 5, then*

$$\gamma_C^A(V_R^{spj})) = \gamma_C^A \sigma_P V$$

where $P = \bigvee_{i=1}^n \overline{null}(\mathcal{R}_i \cap \mathcal{S})$

Each component of predicate P represents the recovery predicate of a required term but restricted to the set of tables that expose at least one non-null column in V.

Proof. Consider a tuple t such that $t \in V_R^{spj}$. Then t originates from one of the required terms, say $\sigma_{p_j}(\mathcal{R}_j)$. t is the projection of some tuple t' onto a subset of columns of tables in $\mathcal{R}_j$. Tuple t' clearly satisfies the recovery predicate $\overline{null}(\mathcal{R}_j)$ of term j. If so, t' must also satisfy the weaker predicate $\overline{null}(\mathcal{R}_j \cap \mathcal{S})$ and so must its projection t because $\overline{null}(\mathcal{R}_j \cap \mathcal{S})$ only references columns available in t. This proves that $t \in \gamma_C^A \sigma_P V$.

Now consider a tuple $t \in V$ but $t \notin V_R^{spj}$. In other words, $t \in V_E^{spj}$ so t originates from one to the excess terms, say, $\sigma_{p_k}(\mathcal{R}_k)$ where $k > n$. It follows that t satisfies the predicate $\overline{null}(\mathcal{R}_k \cap \mathcal{S})$. What we need to prove is that t does not satisfy predicate P, which we do by contradiction. Suppose t satisfies P. Without loss of generality, assume that it satisfies the first component $\overline{null}(\mathcal{R}_1 \cap \mathcal{S})$ of P, which represents the required term $\sigma_{p_1}(\mathcal{R}_1)$. However, this term satisfies the requirements of Lemma 5 which guarantees that the selection $\sigma_{\overline{null}(\mathcal{R}_1 \cap \mathcal{S})} V$ eliminates all tuples from excess terms. Tuple t violates this guarantee and, hence, t cannot satisfy predicate P. This proves that $t \notin \gamma_C^A \sigma_P V$. □

8.1.3 Constructing recovery predicates

For aggregation views, recovery predicates are constructed based on the condition in Theorem 9. Let $\sigma_{p_i}(\mathcal{R}_i), i = 1, 2, \cdots, n$ be the terms of the view required by the query and $\mathcal{S}$ the tables for which the view outputs at least one non-null, non-aggregated column. If all required terms satisfy the condition in Lemma 5, the predicate $\bigvee_{i=1}^n \overline{null}(\mathcal{S} \cap \mathcal{R}_i)$ will recover all tuples of the required terms and the result will not contain tuples from excess terms.

8.2 Further Selection and Aggregation

A substitute expression for an aggregation view may also include further selection and aggregation. For non-aggregated views, each term was recovered separately so different residual selection predicates could be

applied do different terms. In general, this is not possible for aggregation views because individual terms are not recovered separately. However, it is possible if the same residual predicate is to be applied to all required terms. The common residual predicate can be factored out and, provided the necessary columns are available, combined (ANDed) with the recovery predicate. [2]

The groups formed by the query can be computed from the groups of the view if the group-by list of the query is a subset of or equal to the group-by list of the view. That is, if the view is grouped on expressions A, B, C then the query can be grouped on any subset of A, B, C, including the empty set. As shown in [15], this is stricter than absolutely necessary; it is sufficient that the grouping expressions of the view functionally determine the grouping expressions of the query. If the grouping list of the query functionally determines the grouping list of the view and vice-versa, no further aggregation is necessary. If further aggregation is needed, we apply the grouping list of the query.

Example 10. Suppose we have the following outer-join aggregation view.

```
create view revenue_by_custsupp as
select o_custkey, s_suppkey, s_name,
  sum(l_quantity*l_extendedprice) as rev,
  count(l_quantity) as cntq, count(*) as cnt
from supplier full outer join
  (orders left outer join lineitem
    on (l_orderkey=o_orderkey))
  on (s_suppkey=l_suppkey)
group by o_custkey, s_suppkey, s_name
```

The normal form of the SPOJ part of the view contains three terms with source sets $\{S, O, L\}$, $\{O\}$, and $\{S\}$. The hubs of the terms are L, O, and S, respectively, because both orders and supplier are joined to lineitem by extension joins. (Two terms are eliminated from by the full outer join when computing the normal form: the term $\{S, O\}$ by the null-rejecting join predicate and the term $\{O, L\}$ by containment in the term $\{S, O, L\}$.)

Can the following query be computed from the view and, if so, how?

```
select c_nationkey, sum(l_quantity*l_extendedprice)
from (orders left outer join lineitem
  on (o_orderkey = l_orderkey)) q1, customer
where c_custkey = o_custkey
group by c_nationkey
```

[2]This is more restrictive than necessary. If the view outputs enough non-aggregated columns, it is sometimes possible to separate the contributions from individual terms (or groups of terms) and apply residual predicates to individual terms (or groups of terms). We have not determined the exact conditions when this is possible.

Clearly the view cannot match the complete query. However, if we pre-aggregate the result of the left-outer-join expression by customer key, we obtain a matchable subquery.

```
select c_nationkey, sum(sm1)
from (select o_custkey,
        sum(l_quantity*l_extendedprice) sm1
      from orders left outer join lineitem
        on (o_orderkey = l_orderkey)
      group by o_custkey ) as q1, customer
where c_custkey = o_custkey
group by c_nationkey
```

The normal form of the inner subquery contains two terms with source set $\{O, L\}$ and $\{O\}$; they map to the view terms $\{S, O, L\}$ and $\{O\}$, respectively. The $\{O, L\}$ term of the query is contained in the $\{S, O, L\}$ term of the view and with the correct duplication factor (step one) because they have the same hub $\{O\}$.

We then apply the condition in theorem 9 to test whether the excess term $\{S\}$ of the view can be eliminated. For the $\{O, L\}$ term the condition is

$$(\{S, O, L\} - \{S\}) \cap \{S, O\} \cap \{S, O, L\} = \{O\}$$

and for the $\{O\}$ term it is

$$(\{S, O, L\} - \{S\}) \cap \{S, O\} \cap \{O\} = \{O\}.$$

Both set expressions return $\{O\}$ so the excess $\{S\}$ term can be eliminated by the predicate `o_custkey is not null`. This is correct because both required terms have non-null o_custkey values while the excess {S} term is null extended on o_custkey.

The grouping columns of the (inner) query (o_custkey) is a subset of the grouping columns of the view (o_custkey, s_suppkey, s_name) so further aggregation is needed. Combining everything together produces the following rewrite of the query.

```
select c_nationkey, sum(sr)
from (select o_custkey, sum(rev) as sr
    from revenue_by_custsupp
    where o_custkey is not null
    group by o_custkey ) as q1, customer
where o_custkey = c_custkey
group by c_nationkey
```

9 Experimental Results

We ran a series of experiments on Microsoft SQL Server 2005 Beta2 to evaluate the performance benefit of using an outer join view. We followed our algorithms to detect if an outer join view is useable and manually rewrote queries to use the view.

The experiments were performed on a workstation with two 3.20 GHz Xeon processors, 2GB of memory

and three SCSI disks. All queries were against a 1GB version (SF=1) of TPC-H database.

In the first experiment, we created an outer join view of the tables *Customer*, *Orders*, *Lineitem* and ran a set of queries requesting different tuple patterns. We also list abbreviated normal forms, leaving out detailed predicates and output columns.

$V_1 : \pi(\sigma(C) \oplus \sigma(C,O) \oplus \sigma(C,O,L))$

```
create view V1 as
select c_custkey, c_name, c_nationkey, o_orderkey,
  o_custkey, o_orderdate, o_totalprice, l_orderkey,
  l_linenumber, l_partkey, l_quantity, l_extendedprice
from (customer left outer join orders
      on (c_custkey = o_custkey))
  left outer join lineitem on (o_orderkey=l_orderkey
      and l_extendedprice > 50K)
```

$Q_1 : \pi(\sigma(C,O,L))$

```
select V1.*
from customer, orders, lineitem
where c_custkey = o_custkey
and o_orderkey = l_orderkey
and l_extendedprice > 50K
and c_custkey > 100K
```

$Q_2 : \pi(\sigma(C,O) \oplus \sigma(C,O,L))$

```
select V1.*
from (customer join orders on (c_custkey = o_custkey
      and c_custkey > 100K))
  left outer join lineitem on (o_orderkey=l_orderkey
      and l_extendedprice > 50K)
```

$Q_3 : \pi(\sigma(C,O) \oplus \sigma(C,O,L))$

```
select V1.*
from (customer join orders on (c_custkey = o_custkey)
  left outer join lineitem on (o_orderkey=l_orderkey
      and l_extendedprice > 75K)
```

$Q_4 : \pi(\sigma(C,O))$

```
select c_custkey, c_name, c_nationkey, o_orderkey,
  o_custkey, o_orderdate, o_totalprice
from customer, orders
where c_custkey = o_orderkey
```

Term	*COL*	*CO*	*C*
Cardinality	1897761	431165	50004

Table 1: View V_1 Configuration

Table 1 shows cardinalities of different terms in the view V_1. Q_1, Q_2, and Q_4 can be answered by a single scan of V_1 with different predicates. Q_3 requires computing minimum union and Q_4 also requires duplicate elimination. Their rewrites in SQL are shown below.

Q_1':

```
select * from V1
where l_linenumber is not null
and c_custkey > 100K
```

Q_2':

```
select * from V1
where c_custkey > 100K
and o_custkey is not null
```

Q_3':

```
select * from V1
where l_extendedprice > 75K
union all
select c_custkey, c_name, c_nationkey, o_orderkey,
  o_custkey, o_orderdate, o_totalprice
  null, null, null, null, null
from V1
where o_orderkey is not null
group by c_custkey, c_name, c_nationkey,
  o_orderkey, o_custkey, o_orderdate, o_totalprice
having sum(case when l_extendedprice > 75K
  then 1 else 0 end) = 0
```

Q_4':

```
select Q4.* from V1
where o_custkey is not null
group by Q4.*
```

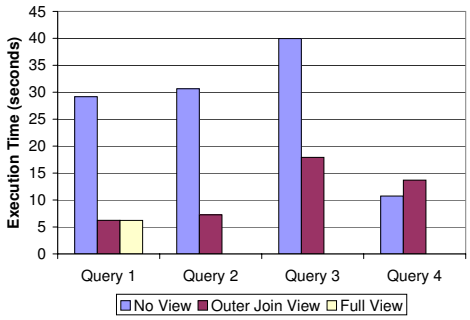

Figure 2: Using Outer Join Views

Figure 2 compares the performance of the original queries and their corresponding rewrites using V_1. Q_1 can also be answered by a normal view over the three tables (with the same join predicates of V_1). Compared with using the normal view, there is little overhead of using the outer join view V_1. Using V_1 improves query performance for the first three queries but Q_4' requires expensive aggregation and the overhead outweighs the benefit. This is an example where an outer join view should not be used even though the query can be computed from the view. As for other types of materialized views, the decision should be made by the optimizer in a cost-based fashion.

In the second experiment, we created an aggregation view to compute total lineitem quantity for every nation, order status and shipment. The aggregation query Q_5 on the next page can be computed from the

view.

```
create view V2 as
select c_nationkey, o_orderstatus, l_shipmode,
  sum(l_quantity) sq, count(*) cn
from (customer left outer join orders
      on (c_custkey = o_custkey))
  left outer join lineitem on (o_orderkey=l_orderkey
      and l_extendedprice > 50K)
group by c_nationkey, o_orderstatus, l_shipmode
```

Q_5:

```
select c_nationkey, o_orderstatus,
  sum(l_quantity), count(*)
from (customer join orders
      on (c_custkey = o_custkey))
  left outer join lineitem on (o_orderkey=l_orderkey
      and l_extendedprice > 50K)
group by c_nationkey, o_orderstatus
```

Q_5':

```
select c_nationkey, o_orderstatus, sum(sq), sum(cn)
from V2
where o_orderstatus is not null
group by c_nationkey, o_orderstatus
```

View V_2 contains 625 rows. Q_5 can be answered from V_2 using a selection and further aggregation, as shown in Q_5'. This reduced the execution time by four orders of magnitude, from 28.3 sec to 0.001 sec.

10 Related Work

To the best of our knowledge, this paper is the first to study view matching for outer join views. We build directly on two earlier papers: Galindo-Legaria's paper on join-disjunctive form for SPOJ expressions [5] and Goldstein and Larson's paper on view matching [8]. Other related work falls into two categories: work on outer joins and work on view matching.

Rewrite rules for outer join expressions are important for query optimization. This is the topic of a series of papers by Galindo-Legaria and Rosenthal culminating in [6], which provides a comprehensive set of simplification and reordering rules for SPOJ expressions. This work was extended by Bhargava, Goel, and Iyer in [2, 7] and by Rao et al in [12, 13].

Larson and Yang [9, 16] were the first to describe a view-matching algorithm for SPJ queries and views. Chaudhuri et al. [4] published the first paper on incorporating the use of materialized views into query optimization, in their case, a System-R style optimizer. Levy, Mendelzon and Sagiv [10] studied the complexity of rewriting SPJ queries using views and proved that many related problems are NP-complete. Srivastava et al. [14] present a view-matching algorithm for aggregation queries and views. Chang and Lee [3] recognized that a view can sometimes be used even if it contains extra tables. Pottinger and Levy [11] considered the view-matching problem for conjunctive SPJ queries and views in the context of data integration where the requirements are somewhat different.

Oracle was the first commercial database system to support materialized views [1]. The query rewrite algorithm is briefly described in the Oracle manuals. Zaharioudakis et al. [17] describe a view-matching algorithm implemented in DB2 UDB. The algorithm performs a bottom-up matching of query graphs but does not require an exact match.

11 Concluding Remarks

This paper provides the first view matching algorithm for views and queries containing outer joins (SPOJG views). By converting expressions into join-disjunctive normal form, the view matching algorithm is able to reason about semantic equivalence and subsumption instead of being based on bottom-up syntactic matching of expressions. The algorithm deals correctly with SQL bag semantics and exploits not-null constraints, uniqueness constraints and foreign key constraints. Experimental results on a few queries show substantial improvements in query performance, especially for aggregation queries.

Efficient incremental update of SPOJG views is an important issue. The join-disjunctive normal form is very helpful here because it makes it possible to detect SPJ terms that are unaffected by an update. Our results on incremental update will be reported in a separate paper.

References

[1] R. G. Bello, K. Dias, A. Downing, J. J. Feenan, Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *Proc. of VLDB Conference*, 1998.

[2] G. Bhargava, P. Goel, and B. R. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *Proc. of SIGMOD Conference*, 1995.

[3] J.-Y. Chang and S.-G. Lee. Query reformulation using materialized views in data warehouse environment. In *Proc. of DOLAP*, 1998.

[4] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. of ICDE Conference*, 1995.

[5] C. Galindo-Legaria. Outerjoins as disjunctions. In *Proc. of SIGMOD Conference*, 1994.

[6] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1), 1997.

[7] P. Goel and B. R. Iyer. Sql query optimization: Reordering for a general class of queries. In *Proc. of SIGMOD Conference*, 1996.

[8] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proc. of SIGMOD Conference*, 2001.

[9] P.-Å. Larson and H. Z. Yang. Computing queries from derived relations. In *Proc. of VLDB Conference*, 1985.

[10] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Rewriting aggregate queries using views. In *Proc. of PODS Conference*, 1995.

[11] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *Proc. of VLDB Conference*, 2000.

[12] J. Rao, B. G. Lindsay, G. M. Lohman, H. Pirahesh, and D. E. Simmen. Using eels, a practical approach to outerjoin and antijoin reordering. In *Proc. of ICDE*, 2001.

[13] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In *Prof. of SIGMOD Conference*, 2004.

[14] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *Proc. of VLDB Conference*, 1996.

[15] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *Proc. of VLDB Conference*, 1995.

[16] H. Z. Yang and P.-Å. Larson. Query transformation for PSJ-queries. In *Proc. of VLDB Conference*, 1987.

[17] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *Proc. of SIGMOD Conference*, 2000.