View Matching for Outer-Join Views

Per-Åke Larson Jingren Zhou

Microsoft Research
{palarson, jrzhou}@microsoft.com

Abstract

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 does not use 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.

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 projection, selection, 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)
```


Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

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 7. Initial experimental results are presented in Section 8. Finally, we survey related work in Section 9 and conclude in Section 10. Due to space limitations, we omit most proofs; all proofs can be found in [10].

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. Borrowing from SQL, we use the shorthand $T.*$ where T is a single table or a set of tables. $T.*$ denotes all columns of table(s) T. 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. The notation $null(\mathcal{S})$, where $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, is a shorthand for $null(S_1) \wedge \ldots \wedge null(S_n)$ and $\sim null(\mathcal{S})$ is a shorthand for $\sim null(S_1) \wedge \ldots \wedge \sim null(S_n)$. $null(T)$ 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.

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* 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 $T_1 \rtimes_p T_2 = (T_1 \bowtie_p T_2) \oplus T_1$. The *right outer join* is $T_1 \ltimes_p T_2 = T_2 \rtimes_p T_1$. 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 $C \ltimes_{ck=ock} O \ltimes_{ok=lok} (\sigma_{lp>50K} L)$, all tuples of $C \ltimes_{ck=ock} O$, 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 $C \ltimes_{ck=ock} O$.

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.

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

3.1 Transformation Rules

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

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

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

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

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

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

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 $(T_1 \ltimes_p T_2 \oplus T_2)$ 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 $\sigma_{p(1)}(T_1 \rtimes_p T_2) = (\sigma_{p(1)} T_1) \rtimes_p T_2$ by expanding the two outer joins.

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]. We retain the proof of Lemma 1 because it shows how to compute the normal form but omit the other proofs for lack of space.

Lemma 1. *Let Q_1 and Q_2 be SPOJ expressions in join-disjunctive form. Then the expressions $\sigma_p(Q_1)$, $Q_1 \ltimes_p Q_2$, $Q_1 \rtimes_p Q_2$, 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 $\mathcal{T}_{11}$, $\mathcal{T}_{12}$, …, $\mathcal{T}_{1n}$ 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.

The case $\sigma_p(Q_1)$ is straightforward and omitted. For the case $Q_1 \ltimes_p Q_2$, repeated application of rule (3) converts the expression into

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

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, $Q_1 \rtimes_p Q_2$ 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 $p_2 \Rightarrow p_1$.*

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

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. The proof does not consider any constraints on the database. 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 computing the normal form. 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. The algorithm, shown in 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 ($Pred$).

Example 3. We compute the normal form of the view

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

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

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

The term $\sigma_{lp<20} L$ is subsumed by the term $\sigma_{lp<20 \wedge ok=lok}(O, L)$ because the join is a foreign key join and therefore eliminated from the result. Next, the algorithm is applied to the left outer join and produces the normal form.

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

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

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 typically 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 [8].

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, i.e. 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.*

Algorithm 1: Normalize(E)

```
Input: Expression E
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 ∧ 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 = ∅;  /*terms after join                      */
  TermSet EL = ∅;  /*terms eliminated by subsumption       */
  /* Multiply the two input sets (rule (3))                 */
  foreach Term l ∈ LT do
   foreach Term r ∈ RT do
    Term t = {(l.Tables ∪ r.Tables), l.Pred ∧ r.Pred ∧ JP};
    /* Apply rule (3) to eliminate terms                    */
    if !(JP rejects nulls on a table not in t.Tables) then
     JT = JT ∪ {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 σ_{l.Pred}(l.Tables) ⊆ σ_{t.Pred}(t.Tables) then
      EL = EL ∪ {l};
     end
     if σ_{r.Pred}(r.Tables) ⊆ σ_{t.Pred}(t.Tables) then
      EL = EL ∪ {r};
     end
    end
   end
  end
  /* Add inputs from preserved side(s) (rules (1) and (2))  */
  switch JoinType do
   case full outer:
    JT = JT ∪ LT ∪ RT; break;
   case left outer:
    JT = JT ∪ LT; break;
   case right outer:
    JT = JT ∪ RT; break;
  end
  /* Discard terms eliminated by subsumption               */
  JT = JT − EL;
  return JT
end
```

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 and Q_1' and Q_2' their join-disjunctive forms.*

Then $Q_1 \subset_s Q_2$ if and only if the following condition holds: for every term $\sigma_{p1}(\mathcal{S})$ in Q_1', there exists a term $\sigma_{p2}(\mathcal{T})$ in Q_2' such that $\mathcal{S} \subseteq \mathcal{T}$ and $\sigma_{p1}(\mathcal{S}) \subseteq \pi_{\mathcal{S}.*}\sigma_{p2}(\mathcal{T})$.

Corollary 1. *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.*

This important corollary follows immediately from the theorem because we require that each result tuple can be constructed from a single view tuple.

5 Recovering All Tuples of a Term

The result tuples of a term in the normal form of a SPOJ view are implicitly contained in the result the view. A tuple t in the result 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 SPJ query $\sigma_{p3}(\mathcal{S}_1)$ and we have shown that all tuples needed by the query are contained in the term $\sigma_{p1}(\mathcal{S}_1)$ of the view. 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 4. 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

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

The following theorem shows how to recover the tuples of a SPJ term from a SPOJ view when a unique key is available.

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

Now we consider how to recover the tuples of a SPJ term when no key is available.

Example 5. 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.

Theorem 4. *Let $\sigma_P(\mathcal{R})$ be a SPJ term of a view V. Then $\sigma_P(\mathcal{R}) = \pi_{\mathcal{R}.*}\sigma_{\sim null(\mathcal{R})}V$ 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.

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 $\sigma_{\sim null(\mathcal{R})}V = \sigma_{\sim null(\mathcal{S})}V$ 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 $\sim null(\mathcal{R})$ 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 $\sim null(\mathcal{S})$ 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 is the high-level steps of the view matching algorithm; the steps are described in more detail in separate sections.

Algorithm SPOJ-View-Matching:

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.

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

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 and query produces the following expressions.

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

6.2 Checking Containment

To check that the view contains all tuples required by the query we must check containment of each term of the query (Theorem 2). That is, for every term $\sigma_{p1}(\mathcal{S})$ in the query, we must 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 7.2).

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, 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 $null$ and $\sim null$ predicates constructed earlier. Apply duplicate elimination if needed.
2. Restrict the result using a selection with the appropriate residual predicates, if any.
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.

$$\begin{aligned} Q_1 &= \pi_{lok,lq,lp,od,otp}(\sigma_{otp>150}(\sigma_{otp\neq null \wedge lq \neq null} V_1)) \\ &\quad \oplus \pi_{lok,lq,lp,null,null}(\sigma_{lq\neq null} V_1) \\ &= \pi_{lok,lq,lp,od,otp}(\sigma_{otp>150 \wedge lq\neq null} V_1) \\ &\quad \oplus \pi_{lok,lq,lp,null,null}(\sigma_{lq\neq null} V_1) \end{aligned}$$

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 projections reduce the tuples to the desired output columns. Note that the columns *od* and *otp* have been replaced by nulls in the second term. The first part of the expression can be simplified by combining the two selections and the predicate $otp \neq null$ can be discarded because the predicate $(otp > 150)$ is null-rejecting on *otp*.

So far so good but, unfortunately, the resulting expression cannot be evaluated because no commercial database systems supports minimum union directly. However, minimum union can be expressed in SQL, which is the topic of the next subsection.

6.7 Computing Minimum Union Using SQL

Let's analyze what the minimum union in the substitute expression actually does. All tuples generated by the first SPJ term will remain in the result. A tuple t_2 generated by the second SPJ term will be eliminated from the results if it is subsumed by a tuple t_1 generated by the first term. Tuple t_2 must be null on all columns from table O, so tuple t_1 subsumes t_2 if it agrees with t_2 on all columns from table L. We can express this in SQL using an exists subquery.

```
// first term
select lok, lq, lp, od, otp
from V1
where otp > 150 and lq is not null
union all
// second term
select lok, lq, lp, null, null
from V1 vo
where lq is not null
and not exists (select * from V1 vi
     where vi.otp > 150 and vi.lq is not null
     and vi.lok = vo.lok and vi.ln = vi.ln
     and vi.lq = vo.lq and vi.lp = vo.lp)
```

The *where* clause in the subquery can be simplified by observing that $\{lok, ln\}$ is a unique key of the second SPJ term of the query. If two tuples agree on the key, they must agree on all other columns as well. Even so, this expression may be rather inefficient because of the join of the view with itself. We can eliminate the join and compute the second term using a group-by with having as follows.

```
// first term
select lok, lq, lp, od, otp
from V1
where otp > 150 and lq is not null
union all
// second term
select lok, lq, lp, null, null
from V1
where lq is not null
group by lok, ln, lq, lp
having sum(case when otp > 150
     and lq is not null then 1 else 0 end) = 0
```

The predicate "*lq* is not null" recovers all tuples of the second SPJ term. The group-by brings together all tuples that agree on all columns originating from L. Note that the list of grouping columns includes the unique key (lok, ln) of the term. The aggregation function $sum()$ counts the number of tuples in each group that belong to the first SPJ term. If there are one or more such tuples, the output tuple from the group is rejected because it is subsumed by a tuple included in the first SPJ term. Note that the initial SQL expression is always a valid way of computing the minimum union, while the second expression is more efficient but requires a unique key. [1]

This example illustrates the general procedure for constructing a substitute expression using only SQL constructs. Suppose we need to construct the SQL statement for a SPJ term with base $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. Denote the predicate required for tuple recovery by P_{v1} and the residual predicate by P_{r1}. Suppose the query outputs columns $C_1, C_2, \ldots, C_k$ from tables $\mathcal{R}$ (possibly mapped to equivalent columns). The view may output additional columns, which we denote by $C_{k+1}, C_{k+2}, \ldots, C_m$. Some of the tuples generated by $\sigma_p(\mathcal{R})$ may be subsumed by tuples generated by other SPJ terms. We need only consider one such term which we denote by $\sigma_q(\mathcal{R}^+)$ and call the maximally subsuming term. $\sigma_q(\mathcal{R}^+)$ is the term in the query whose base, $\mathcal{R}^+$, is the smallest superset of R. (That is, no other term has a base that is a superset of $\mathcal{R}$ and a subset of $\mathcal{R}^+$.) Suppose $\mathcal{R}^+ = \{R_1, R_2, \ldots, R_n, S_1, \ldots, S_t\}$. We denote the tuple recovery predicate and residual predicates of $\sigma_q(\mathcal{R}^+)$ by P_{v2} and P_{r2}, respectively. Note that $\sigma_p(\mathcal{R})$ does not have a maximally subsuming term if $\mathcal{R}$ includes all source tables of the view.

We distinguish two cases depending on whether columns forming a unique key of term $\sigma_p(\mathcal{R})$ are available in the view output or not. (Query equivalence classes can be applied when looking for key columns.)

If a key of term $\sigma_p(\mathcal{R})$ is *not* available, duplicate elimination cannot be applied but in that case, it should not be necessary either (verified in earlier steps). In this case, we use the following SQL expression. If $\mathcal{R}$ includes all source tables of the query, the last part of the expression (the part enclosed in square brackets) is omitted.

```
select C1, C2, ..., Ck, null, ..., null
from view v1
where Pv1 and Pr1
[and not exists (select * from view v2
```

[1] The columns (lok, ln) are a key not only of the term but also of the complete view V. This fact can be exploited to remove the group by and convert the having clause to a select.

```
where Pv2 and Pr2
and v1.C1=v2.C1 and ... and v1.Cm=v2.Cm)]
```

If the key is available, we use the following SQL expression. Note that the grouping is on all columns from $\mathcal{R}$ that are output by the view so all key columns are automatically included. The *having* clause is omitted if $\mathcal{R}$ includes all source tables of the query.

```
select C1, C2, ..., Ck, null, ..., null
from view
where Pv1 and Pr1
group by C1, C2, ..., Cm
[having sum(case when Pv2 and Pr2 then 1 else 0) = 0]
```

After the expressions for all SPJ terms of the query have been constructed, we simply tie them together using `union all`.

6.8 Avoiding Minimum Union and Duplicate Elimination

Minimum union and duplicate elimination are not always necessary to answer a query from a SPOJ view. For a certain type of queries, a single scan of the view result, with possible selection and projection, is sufficient. Consider the following query

$$\begin{aligned} Q_2 &= \pi_{lok,lq,lp,od,otp}(\sigma_{lq<100}(L) \rtimes_{ok=lok} \sigma_{otp>50}(O)) \\ &\quad \rtimes_{ck=ock} \sigma_{cnk<10}(C) \\ &= \pi_{lok,lq,lp,od,otp}(\sigma_{lq<100}(L) \oplus \\ &\quad \sigma_{otp>50 \wedge ok=lok \wedge lq<100}(O,L) \oplus \\ &\quad \sigma_{cnk<10 \wedge ck=ock \wedge otp>50 \wedge ok=lok \wedge lq<100}(C,O,L)) \end{aligned}$$

By checking containment and recovery, we find that query Q_2 can be computed from view V_1. The query has the same predicates as the view so no residual predicates are required. For each term t of the query, every term in the view whose base is a superset of the base of t is also included in the query result. It follows that, for each tuple in the query, every tuple in the view that can potentially subsume it appears in the query result too. In other words, the subsumption relationships of the view still hold for the query. We can rewrite the query Q_2 as a single scan of the view V_1 combining the recovery predicates for each query term.

$$\begin{aligned} Q_2 = \pi_{lok,lq,lp,od,otp}(&\sigma_{(\sim null(C) \wedge \sim null(O) \wedge \sim null(L)) \vee} \\ &(null(C) \wedge \sim null(O) \wedge \sim null(L)) \vee \\ &(null(C) \wedge null(O) \wedge \sim null(L))} V_1 \end{aligned}$$

The following theorem shows under what circumstances a query can be answered by a single scan of the view result, applying residual predicates to individual terms respectively.

Theorem 6. *Let Q be a query that can be computed from a view V. The query can be computed from the view using a single scan if, for each term $\sigma_P(\mathcal{S})$ in Q, the following conditions are satisfied:*

1. *Every term $\sigma_P(\mathcal{T})$ in V such that $\mathcal{S} \subset \mathcal{T}$ is also included in Q.*
2. *Residual predicates, if any, reference only tables in $\mathcal{S}$.*

The intuition behind this theorem is that as long as the subsumption relationships of the view hold also in the query, the query can be answered by a single scan.

Consider the following query Q_3. The only difference between Q_2 and Q_3 is that Q_3 has a more restrictive predicate on table L. In this case, L is the only table involved in a residual predicate.

$$\begin{aligned} Q_3 = \pi_{lok,lq,lp,od,otp}(&\sigma_{\mathbf{lq<50}}(L) \rtimes_{ok=lok} \sigma_{otp>50}(O)) \\ &\rtimes_{ck=ock} \sigma_{cnk<10}(C) \end{aligned}$$

By Theorem 6, Q_3 can be rewritten as

$$\begin{aligned} Q_3 = \pi_{lok,lq,lp,od,otp}(&\sigma_{(\sim null(C) \wedge \sim null(O) \wedge \sim null(L) \wedge lq<50) \vee} \\ &(null(C) \wedge \sim null(O) \wedge \sim null(L) \wedge lq<50) \vee \\ &(null(C) \wedge null(O) \wedge \sim null(L) \wedge lq<50)} V_1) \end{aligned}$$

The selection predicate can be simplified to $(lq < 50) \wedge ((\sim null(C) \wedge \sim null(O)) \vee (null(C) \wedge \sim null(O)) \vee (null(C) \wedge null(O)))$. Note that if Q_3 had a residual predicate on either C or O, the query could not be computed by simple selections.

7 Aggregation Views

In this section, we 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. 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). Step three, checking recovery, changes significantly as described in details in Section 7.1 below. An new step must also be added (after step four) to check whether further aggregation is needed and possible.

Aggregation views normally contain a `count(*)` column because it is needed for incremental maintenance. To assist tuple recovery, we assume that the view also contains a not-null count for each table that may be null-extended but does not output a non-nullable and non-aggregated column. The not-null count of T is denoted by $nn\_count(T)$. In SQL, a not-null count for a table T is simply $count(T.c)$ where c is any column of T that is not nullable.

7.1 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.

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 hubs and verifying that they are equal. The notion of hubs and how to compute the hub of an expression are explained in [8].

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.

For steps two and three above, we first divide the terms of the view into two sets: terms required by the query and excess terms, that is, terms not required. Suppose the view contains an excess term $\sigma_p(\mathcal{S})$. We need to eliminate all tuples of $\sigma_p(\mathcal{S})$ but this is not possible if the aggregation may combine tuples from $\sigma_p(\mathcal{S})$ into the same group as tuples from a required term. The following theorem states a sufficient condition, which is the test we apply in step two. The test is stricter than absolutely necessary but it is simple and covers most of the cases occurring in practice. A more general theorem is covered in reference [10].

Theorem 7. *Let $\sigma_p(\mathcal{S})$ be an excess SPJ term of an aggregation view. The tuples of $\sigma_p(\mathcal{S})$ are guaranteed to remain in separate groups if the view's grouping columns functionally determine a unique key of $\sigma_p(\mathcal{S})$.*

In step three, recovery predicates are constructed. Section 6.3 describes how to construct selection predicates that recover all tuples of a term. The predicates can use non-aggregated output columns that are not nullable and also the not-null count columns mentioned above. Theorem 5 states that a (required) term should be checked against every other term $\sigma_q(\mathcal{T})$ in the normal form. For aggregation view, it is sufficient to check it against excess terms only because we do not extract each required term separately .

7.2 Further Aggregation

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 [16], 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 6. Suppose we have the following outer-join aggregation view. The normal form contains three terms with patterns SOL, S, and O.

```
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
```

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
```

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 patterns OL and O. The OL term of the query is contained in the SOL term of the view and with the correct duplication factor (step one) because *Supplier* is joined in through an extension join. The excess S term can be eliminated (step two), because the required terms both have not-null `o_custkey` values while the S term is null extended on O and L. Hence, we can recover the required terms using the predicate $o\_custkey \neq null$. Further grouping on `o_custkey` is also needed. Combining everything together produces the following rewrite of the query.

```
select c_nationkey, sum(sr)
from (select o_custkey, sum(rev) 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
```

8 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
      and c_custkey > 100K))
  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. By Theorem 6, Q_1, and Q_2 can be answered by a single scan of V_1 with different predicates. Q_3 requires computing minimum union and Q_4 requires duplicate elimination. Their rewrites in SQL are shown below.

Q_3':

```
select * from V1
where l_extendedprice > 75000
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 > 75000
  then 1 else 0 end) = 0
```

Q_4':

```
select Q4.* from V1
where o_custkey is not null
and l_linenumber is null
union all
select Q4.* from V1
where l_linenumber is not null
group by Q4.*
```

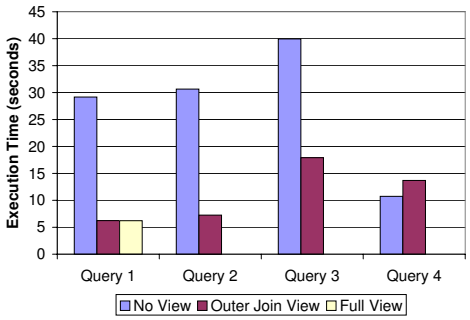

Figure 1: Using Outer Join Views

Figure 1 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.

9 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 [13, 14].

Larson and Yang [9, 17] 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 [11] studied the complexity of rewriting SPJ queries using views and proved that many related problems are NP-complete. Srivastava et al. [15] 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 [12] 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. [18] 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.

10 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. *VLDB*, 1998.

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

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

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

[5] C. Galindo-Legaria. Outerjoins as disjunctions. *SIGMOD*, 1994.

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

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

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

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

[10] P.-Å. Larson and J. Zhou. View matching for outer-join views. Technical report, Microsoft Resarch, MSR-TR-2005-78, 2005.

[11] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Rewriting aggregate queries using views. *PODS*, 1995.

[12] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. *VLDB*, 2000.

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

[14] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. *SIGMOD*, 2004.

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

[16] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. *VLDB*, 1995.

[17] H. Z. Yang and P.-Å. Larson. Query transformation for PSJ-queries. *VLDB*, 1987.

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