The VLDB Journal (2007) 16:29–53
DOI 10.1007/s00778-006-0027-9

SPECIAL ISSUE PAPER

View matching for outer-join views

Per-Åke Larson · Jingren Zhou

Received: 30 January 2006 / Accepted: 3 August 2006 / Published online: 13 September 2006

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 a view matching algorithm for views that may also contain outer joins (SPOJG views). The algorithm relies on a normal form for outer-join 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.

Keywords Query processing · Materialized views · View matching · Outer joins · Aggregation

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 a 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 present a general view matching algorithm for views where some of the joins may be outer joins (SPOJG views). An early and incomplete description was provided in a previous paper [10].

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. It is well known that SPJ expressions can be converted to a normal form consisting of the 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 the algorithm in this paper.

Example 1 Suppose we create the following view against tables in the TPC-H 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 $p\_partkey < 100$, 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
```

P.-Å. Larson(✉) · J. Zhou
Microsoft Research, One Microsoft Way,
Redmond, WA 98052, USA
e-mail: palarson@microsoft.com

J. Zhou
e-mail: jrzhou@microsoft.com

A view matching algorithm based on bottom-up syntactic matching would most likely conclude that it cannot because the initial (innermost) joins are over different tables, that is, *Orders* and *Lineitem* in the view versus *Part* and *Lineitem* in the query.

However, the query can in fact be computed from the view. The join between *Orders* and *Lineitem* in the view 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.

In summary, the view will contain one complete tuple for each *Lineitem* tuple and also all non-joining *Part* tuples null-extended on columns from *Lineitem* and *Orders*. Hence, the view contains all required tuples and 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-H 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, 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. Sect. 2 introduces the notation used in the rest of the paper. In Sect. 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 Sect. 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 Sect. 8. Initial experimental results are presented in Sect. 9. Finally, we survey related work in Sect. 10 and conclude in Sect. 11.

2 Definitions and notation

The selection operator is denoted in the normal way by σ_p where p is a predicate. The notation $\sigma_p(\mathcal{T})$ where $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ denotes the expression $\sigma_p(T_1 \times T_2 \times \cdots \times T_n)$, that is, the normal form of a select-inner-join expression.

Projection (without duplicate elimination) is denoted by π_c where c is a list of columns. Borrowing from SQL, we also use the notation $T.*$ where T is a table. $T.*$ denotes all columns of table T that are available in the input to the projection. We slightly extend this notation and write $\mathcal{T}.*$ where $\mathcal{T}$ is a set of tables.

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. Predicates $null(T)$ and $\neg null(T)$ 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 predicate $N(\mathcal{T})$, where $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$, is a shorthand for $null(T_1) \wedge \cdots \wedge null(T_n)$ and $NN(\mathcal{T})$ is a shorthand for $\neg null(T_1) \wedge \cdots \wedge \neg null(T_n)$.

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 sub-*

sumed tuples of T, denoted by $T\downarrow$, returns the tuples of T that are not subsumed by any other tuple in T.

The following new operator is due to Galindo-Legaria [5]. 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 the same precedence as regular union. It can be shown that minimum union is both commutative and associative. In addition, if T_1 does not contain subsumed tuples, then $T_1 \oplus \emptyset = T_1$ and, if $T_2 \subseteq T_1$, then $T_1 \oplus T_2 = T_1$.

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 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 anti (semi) join* is $T_1 \triangleright_p T_2 = \{t_1 | t_1 \in T_1, (\nexists\ t_2 \in T_2 | p\ (t_1, t_2))\}$, that is, a tuple in T_1 qualifies if it does not join with any tuple in T_2. The *left outer join* is $T_1 \bowtie^{lo}_p T_2 = (T_1 \bowtie_p T_2) \uplus (T_1 \triangleright_p T_2)$. The *right outer join* is $T_1 \bowtie^{ro}_p T_2 = T_2 \bowtie^{lo}_p T_1$. The *full outer join* is $T_1 \bowtie^{fo}_p T_2 = (T_1 \bowtie_p T_2) \uplus (T_1 \triangleright_p T_2) \uplus (T_2 \triangleright_p T_1)$

We assume that base tables contain no subsumed tuples. This is usually the case in practice because base tables almost always contain a primary key. We also assume that predicates are null-rejecting on all columns that they reference. For ease of presentation we assume that column names are unique among the tables referenced in an expression.

We make a distinction between unique keys and primary keys. Both key types have the uniqueness property, that is, no two tuples can have the same value of the key columns. A primary key does not allow null values for any of its columns but a unique key does. When comparing two unique keys to determine whether they are equal, nulls are treated as regular values, that is, null is equal to null. In SQL, the nulls are treated in this way in the context of sorting, grouping, and indexing.

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 slightly 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-H 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

$$Q = C \bowtie^{lo}_{ck=ock} \left(O \bowtie^{lo}_{ok=lok} (\sigma_{lp>50K} L) \right).$$

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

$$T_1 \ltimes^{lo}_p T_2 = T_1 \ltimes_p T_2 \oplus T_1; \quad \text{if } T_1 = T_1{\downarrow} \text{ and } T_2 = T_2{\downarrow} \tag{1}$$

$$T_1 \ltimes^{fo}_p T_2 = T_1 \ltimes_p T_2 \oplus T_1 \oplus T_2; \quad \text{if } T_1 = T_1{\downarrow} \text{ and } T_2 = T_2{\downarrow} \tag{2}$$

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

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

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

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 $\sigma_{p(1)}(T_1 \ltimes^{lo}_p T_2) = (\sigma_{p(1)} T_1) \ltimes^{lo}_p T_2$ by expanding the two outer joins.

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

$$\begin{aligned}
&(\sigma_{p(1)}(T_1 \ltimes^{ro}_{p(1,2)} T_2)) \ltimes^{fo}_{p(2,3)} T_3 \\
&= (\sigma_{p(1)}(T_1 \ltimes_{p(1,2)} T_2 \oplus T_2)) \ltimes^{fo}_{p(2,3)} T_3 && \text{by rule (1)} \\
&= ((\sigma_{p(1)} T_1) \ltimes_{p(1,2)} T_2) \ltimes^{fo}_{p(2,3)} T_3 && \text{by rule (4)} \\
&= (T_1 \ltimes_{p(1) \wedge p(1,2)} T_2) \ltimes^{fo}_{p(2,3)} T_3 \\
&= (T_1 \ltimes_{p(1) \wedge p(1,2)} T_2 \ltimes_{p(2,3)} T_3) \oplus \\
&\quad (T_1 \ltimes_{p(1) \wedge p(1,2)} T_2) \oplus T_3 && \text{by rule (2)}
\end{aligned}$$

3.2 Join-disjunctive normal form

In this section, we show that an 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.

Definition 1 *A relational expression of the form*

$$\sigma_{p_1}(\mathcal{T}_1) \oplus \sigma_{p_2}(\mathcal{T}_2) \oplus \cdots \oplus \sigma_{p_n}(\mathcal{T}_n)$$

where $\mathcal{T}_i \neq \mathcal{T}_j$ *for* $1 \leq i,j \leq n, i \neq j$ *is said to be in* join-disjunctive form. *Each* $\sigma_{p_i}(\mathcal{T}_i)$ *is called an* SPJ term *or simply a* term.

The following lemma shows that any SPOJ expression can be converted into join-disjunctive form.

Lemma 1 *Suppose expressions* Q_1 *and* Q_2 *are in join-disjunctive form and* p *a strong predicate. Then the expressions* $\sigma_p(Q_1)$, $Q_1 \ltimes_p Q_2$, $Q_1 \ltimes^{lo}_p Q_2$, *and* $Q_1 \ltimes^{fo}_p Q_2$ *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 $\sigma_{p11}(\mathcal{T}_{11}) \oplus \sigma_{p12}(\mathcal{T}_{12}) \oplus \cdots \oplus \sigma_{p1n}(\mathcal{T}_{1n})$ 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 $\mathcal{T}_{1i}$. 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 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 $Q_1 \ltimes_p Q_2$, repeated application of rule (3) converts the expression into

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

In essence, we are multiplying the two input expressions producing an output expression containing nm terms. This expression can be converted into $\sigma_{p11 \wedge p21 \wedge p}(\mathcal{T}_{11} \cup \mathcal{T}_{21}) \oplus \sigma_{p12 \wedge p21 \wedge p}(\mathcal{T}_{12} \cup \mathcal{T}_{21}) \oplus \cdots \oplus \sigma_{p1n \wedge p2m \wedge p}(\mathcal{T}_{1n} \cup \mathcal{T}_{2m})$, 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, $Q_1 \ltimes^{lo}_p Q_2$ and $Q_1 \ltimes^{fo}_p Q_2$, follow immediately from the join case by first applying rules (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 an SPOJ expression* Q. *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 $p_2 = p_1 \wedge q_1 \wedge \cdots \wedge q_n$. Any tuple that satisfies $p_1 \wedge q_1 \wedge \cdots \wedge q_n$ must also satisfy p_1. □

Theorem 1 (Galindo-Legaria) *The join-disjunctive form of an 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. It follows that $Q_1 \neq Q_2$. □

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 though 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 it. Lemma 1 and its proof provide the basis for an algorithm by showing 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.

Transformation rule (4) discards terms that are eliminated by null-rejecting predicates but additional terms can be eliminated by exploiting foreign keys. A term $\sigma_{p1}(\mathcal{T}_1)$ can be eliminated if there exists a 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 extra 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 earlier. 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 discard subsumed terms when normalizing query expressions but not when normalizing view expressions. Unless otherwise stated, this is the default.

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

$$V_1 = C \ltimes^{lo}_{ock=ck} \left(O \ltimes^{fo}_{ok=lok} (\sigma_{lp<20} L) \right)$$

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;
  /* A select has an input expression IE and a predicate p  */
  case select operator (IE, p):
    TermSet IT = Normalize(IE);
    foreach Term t in IT do
      if p rejects nulls on a table not in t.Tbls then
        IT = IT − {t}; /* apply rule (4)                    */
      else
        t.Pred = t.Pred ∧ p; /* apply rule (5)              */
    end
    return IT;
  /* A join has two input expressions (LE, RE),             */
  /* a join predicate p and a join type                     */
  case join operator (LE, RE, p, 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.Tbls ∪ r.Tbls), l.Pred ∧ r.Pred ∧ p};
        /* Apply rule (4) to eliminate terms               */
        if !(p rejects nulls on a table not in t.Tbls) then
          JT = JT ∪ {t};
          /* Check whether all tuples in input term are
             subsumed                                      */
          /* by the result term by testing containment of SPJ
             */
          /* expressions, see algorithm in [8].           */
          if Flag then
            if σ_{l.Pred}(l.Tbls) ⊆ σ_{t.Pred}(t.Tbls) then
              EL = EL ∪ {l};
            end
            if σ_{r.Pred}(r.Tbls) ⊆ σ_{t.Pred}(t.Tbls) then
              EL = EL ∪ {r};
            end
          end
        end
      end
    end
    /* Add inputs from preserved sides as needed          */
    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 algorithm recursively descends the operator tree. When applied to the innermost join, it produces[1]

$$V_1 = C \ltimes^{lo}_{ock=ck} (\sigma_{lp<20 \wedge ok=lok}(O, L) \oplus \sigma_{lp<20} L \oplus O)$$

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

$$V_1 = \sigma_{lp<20 \wedge ok=lok \wedge ock=ck}(C, O, L) \oplus \sigma_{ck=ock}(C, O) \oplus C$$

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 relationships among terms are not arbitrary as shown in this section.

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

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

Definition 2 *Let* $E = \sigma_{p_1}(\mathcal{S}_1) \oplus \sigma_{p_2}(\mathcal{S}_2) \oplus \cdots \oplus \sigma_{p_n}(\mathcal{S}_n)$ *be the join-disjunctive form of an SPOJ expression. The* subsumption graph *of E contains a node* n_i *for each term* $\sigma_{p_i}(\mathcal{S}_i)$ *in the normal form and the node is labeled with the source table set* $\mathcal{S}_i$*. There is an edge from node* n_i *to node* n_j*, if* $\mathcal{S}_i$ *is a minimal superset of* $\mathcal{S}_j$*.* $\mathcal{S}_i$ *is a minimal superset of* $\mathcal{S}_j$ *if there does not exist a node* n_k *in the graph such that* $\mathcal{S}_j \subset \mathcal{S}_k \subset \mathcal{S}_i$*.*

Figure 1 shows an example of a subsumption graph. The graph for view V_1 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

[1] The term $\sigma_{lp<20} L$ is subsumed by $\sigma_{lp<20 \wedge ok=lok}(O, L)$ because the join is a foreign key join. If *Flag* had been set to *true*, the term $\sigma_{lp<20} L$ would have been eliminated from the result.

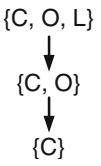

Fig. 1 Subsumption graph for view V_1

liberty of referring to parents, ancestors, and children of terms and of nodes interchangeably.

The following lemma shows that if a tuple of a term is subsumed by any tuple, it is also subsumed by a tuple in one of its parent terms. In other words, to determine whether a tuple is subsumed it is sufficient to check against tuples in the immediate 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, $E_{p_1}, E_{p_2}, \ldots, E_{p_k}$. 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 $q_a \Rightarrow q_{p_1} \Rightarrow q_i$. Tuple 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 all tuples required by the query.

4.1 Subsumption-containment

Definition 3 *Let T_1 and T_2 be tables with the same schema. T_1 is* subsumption-contained *in T_2, denoted by $T_1 \subseteq_s T_2$, 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 $Q_1 \not\subseteq_s Q_2$ 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 $Q_1 \not\subseteq_s Q_2$ 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, respectively. Then $Q_1 \subseteq_s Q_2$ if and only if the following condition holds: for every term $\sigma_{p_1}(\mathcal{S})$ in Q_1', there exists a term $\sigma_{p_2}(\mathcal{S})$ in Q_2' such that $\sigma_{p_1}(\mathcal{S}) \subseteq \sigma_{p_2}(\mathcal{S})$.*

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

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 $\sigma_{p_1}(\mathcal{S}) \not\subseteq \sigma_{p_2}(\mathcal{S})$

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, $Q_1 \not\subseteq_s Q_2$ 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 does not satisfy predicate p_2 so that t is not output by term $\sigma_{p_2}(\mathcal{S})$. Such a tuple must exist because $\sigma_{p_1}(\mathcal{S}) \not\subseteq \sigma_{p_2}(\mathcal{S})$. The rest of the argument is the same as for case (a). □

4.2 Target terms with extra tables

Theorem 2 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 to be contained by a term having additional tables.

Example 5 Consider an SPJ term $E = \sigma_q(R \times S)$. Suppose there is a foreign key constraint from table S to a table T between columns $S.A$ and $T.A$. Now join E and T along the foreign-key constraint, obtaining $E' = \sigma_{q \wedge (S.A=T.A)}(R \times S \times T)$. Then the following equality

$$\sigma_q(R \times S) = \delta\, \pi_{R.*,S.*}\sigma_{q \wedge (S.A=T.A)}(R \times S \times T)$$

holds provided that the result of $\pi_{S.A}\sigma_q(R \times S)$ does not contain nulls. This condition is satisfied if column $S.A$ is part of the primary key of S, $S.A$ is declared with "not null", or q contains a null-rejecting predicate referencing $S.A$.

The important observation is that joining in the additional table T along a foreign key does not eliminate any of the tuples in E because every tuple in E joins with at least one and possibly more tuples in T. Furthermore, if $T.A$ is a unique key of T, then every tuple in E joins with exactly one tuple in T.

Now suppose E is a query term that is mapped to view term $F = \sigma_p(R \times S \times T)$ and we need to determine whether E is contained in F. The test $q \Rightarrow p$ will almost certainly fail because, presumably, p contains a predicate connecting T to the rest of the query. However, we can extend E to E' by logically joining in T along the foreign key constraint and test containment of E' in T instead. That is, instead of testing the implication $q \Rightarrow p$, we test whether $(q \wedge (S.A = T.A)) \Rightarrow p$ holds.

Lemma 5 *Consider two SPJ terms $E_1 = \sigma_{p_1}(\mathcal{R})$ and $E_2 = \sigma_{p_2}(\mathcal{R} \cup \{T\})$. Let $fk(R_i.\mathcal{C}, T.\mathcal{D})$ be a foreign key constraint from columns $\mathcal{C}$ of table $R_i \in \mathcal{R}$ to columns $\mathcal{D}$ of table T. Then $E_1 \subseteq \pi_{\mathcal{R}.*}E_2$ if $(p_1 \wedge (R_i.\mathcal{C} = T.\mathcal{D})) \Rightarrow p_2$ and every tuple in E_1 is non-null on all columns in $R_i.\mathcal{C}$.*

The notation $R_i.\mathcal{C} = T.\mathcal{D}$ is shorthand for an equijoin predicate on the columns of the foreign key condition, that is, $(R_i.C_1 = T.D_1) \wedge \cdots \wedge (R_i.C_m = T.D_m)$.

Proof Consider a tuple r in the result of E_1. We need to show that if the conditions of the lemma are satisfied then there exists a tuple $< r, t >$ in the result of E_2. Because of the foreign key condition and the non-null condition of the lemma, there must exist a tuple t in T that joins with r. Tuple r satisfies the condition p_1 and tuple $< r, t >$ satisfies the join condition $R_i.\mathcal{C} = T.\mathcal{D}$. Consequently, tuple $< r, t >$ satisfies the condition $p_1 \wedge (R_i.\mathcal{C} = T.\mathcal{D})$. Because the implication holds, tuple $< r, t >$ must also satisfy predicate p_2. This being the case tuple $< r, t >$ is present in the result of E_2. This proves that $E_1 \subseteq \pi_{\mathcal{R}.*}E_2$. □

The following theorem shows that the approach of augmentation by foreign-key joins generalizes to multiple tables.

Theorem 3 *Consider two SPJ terms $E_1 = \sigma_{p_1}(\mathcal{R})$ and $E_2 = \sigma_{p_2}(\mathcal{R} \cup \mathcal{S})$, $n = |\mathcal{S}|$, $n \geq 1$. Let $fk_i(T_{i_s}.\mathcal{C}_i, S_i.\mathcal{D}_i)$, $1 \leq i \leq n$ be a sequence of foreign key constraints from columns $\mathcal{C}_i$ of table $T_{i_s} \in \mathcal{R} \cup \mathcal{S}$ to columns $\mathcal{D}_i$ of table $S_i \in \mathcal{S}$. Then $E_1 \subseteq \pi_{\mathcal{R}.*}E_2$ if*

$$\left(p_1 \bigwedge_{i=1}^{n}(T_{i_s}.\mathcal{C}_i = S_i.\mathcal{D}_i)\right) \Rightarrow p_2$$

and, for every $k = 1, 2, \ldots, n$, every tuple in

$$F_k = \sigma_{p_1 \wedge_{i=1}^{k-1}(T_{i_s}.\mathcal{C}_i = S_i.\mathcal{D}_i)}(\mathcal{R} \cup_{i=1}^{k-1} S_i)$$

is non-null on all column in $T_{k_s}.\mathcal{C}_k$.

Proof Follows by, in essence, applying Lemma 5 n times, once for each table in $\mathcal{S}$. Let r be a tuple in E_1. Because of the foreign key condition fk_1 and the non-null condition on columns $T_{1_s}.\mathcal{C}_1$, there exists a tuple $s_1 \in \mathcal{S}_1$ that joins with r, producing the tuple $< r, s_1 >$. Tuple $< r, s_1 >$ satisfies the condition $p_1 \wedge (T_{1_s}.\mathcal{C}_i = S_i.\mathcal{D}_1)$. Applying the same argument to $fk_2, fk_3, \ldots, fk_n$ we find that r must join with tuples $s_1 \in S_1, s_2 \in S_2, \ldots, s_n \in S_n$

and that the tuple $< r, s_1, s_2, \ldots, s_n >$ satisfies the precedent of the implication in the theorem. Because the implication holds, the tuple $< r, s_1, s_2, \ldots, s_n >$ also satisfies p_2 and, consequently, it is present in the result of E_2. This proves that $E_1 \subseteq \pi_{\mathcal{R}.*} E_2$. □

The conditions of the theorem can be slightly relaxed. It is not strictly necessary that a join be on all columns of the foreign key – joining on a subset of the columns is sufficient. This may help in cases when some of the referencing columns (the $\mathcal{C}$ columns) may contain nulls.

The theorem states what condition the sequence of foreign-key joins must satisfy but not how to find a suitable set. Because we need the final predicate to imply predicate p_2, the best place to look for guidance is p_2. An algorithm for finding a sequence of joins satisfying our requirements is described in [8].

It is also easy to see that if E_1 and E_2 have a common unique key, then every tuple t in E_1 occurs exactly once in E_2. In that case, no duplicate elimination is required if E_1 is computed from E_2.

4.3 Mapping query terms to target terms

We must map every term in the query to a target term in the view. Suppose we have a query term $\sigma_q(\mathcal{S})$ and are looking for candidate target term in the view. The normal form of the view may contain several terms such that $\mathcal{S} \subset \mathcal{T}$. It 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. A formal definition follows.

Definition 4 *Let $\sigma_q(\mathcal{S})$ be a term of a query Q. A term $\sigma_{p_i}(\mathcal{T}_i)$ of a view V is a* candidate target term *for $\sigma_q(\mathcal{S})$ if $\mathcal{S} \subseteq \mathcal{T}_i$ and there is no other term $\sigma_{p_j}(\mathcal{T}_j)$ in V such that $\mathcal{S} \subseteq \mathcal{T}_j \subseteq \mathcal{T}_i$.*

A candidate target term with $\mathcal{S} \subset \mathcal{T}_i$ must of course satisfy the conditions stated in Theorem 3 but those conditions will be checked when testing containment.

Normally, a query term has at most one candidate target term in the view but if there are more than one, each candidate target term should be checked for containment.

In a database with constraints it is possible that a query term is not covered by any single one of its candidate target terms but it is covered by the union of two or more of them. However, in this paper we ignore this unlikely scenario and require that each query term be mapped to a single view term. The selected view term is simply called the *target term* of the query term.

5 Recovering all tuples of a term

The tuples of a term in the normal form of a view are contained, implicitly or explicitly, 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 $\sigma_{p2}(\mathcal{S}_2)$, that is, by a term such that $\mathcal{S}_1 \subset \mathcal{S}_2$. 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 tuples of $\sigma_{p1}(\mathcal{S}_1)$ from the view result. The following example illustrates the steps necessary.

Example 6 Consider the following view.

$$\begin{aligned} V_2 &= (\sigma_{cn<5} C) \bowtie^{lo}_{ock=ck} \left(O \bowtie^{lo}_{ok=lok} (\sigma_{lp<20} L) \right) \\ &= \sigma_{cn<5 \wedge lp<20 \wedge ok=lok \wedge ock=ck}(C, O, L) \oplus \\ &\quad \sigma_{cn<5 \wedge ck=ock}(C, O) \oplus \sigma_{cn<5} C \end{aligned}$$

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 $\sigma_{cn<5 \wedge ck=ock}(C, O)$. All the desired tuples are composed of a real C tuple and a real O tuple. In other words, they are not null-extended on C and O and can thus be extracted from the view by the selection $\sigma_{\neg null(C) \wedge \neg null(O)} V_2$. The selection can be simplified to $\sigma_{\neg null(O)} V_2$ because no tuples of V_2 are null-extended on C.

The predicate $\neg null(C)$ 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 multiple L tuples. Hence, if we simply project V_2 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

$$\sigma_{cn<5 \wedge ck=ock}(C, O) = \delta(\pi_{C.*, O.*}(\sigma_{ck \neq null} V_2))$$

The following two theorems show how to recover the tuples of a SPJ term from an SPOJ view. Theorem 4 states under what conditions duplicate elimination is not necessary and Theorem 5 deals with the case when duplicate elimination is needed.

In the following theorem we need to select certain subsets of the terms of a view. Let $\mathcal{R}$ be a subset of the source tables of a view V. Then the *sub-view induced by* $\mathcal{R}$ consists of the minimum union all terms $\sigma_P(\mathcal{T})$ in the normal form of V such that $\mathcal{R} \subseteq \mathcal{T}$. If $\mathcal{R}$ matches the base of a term in V then the sub-view induced by $\mathcal{R}$ consists of the term and all its ancestors in the subsumption graph.

Theorem 4 *Let $\sigma_P(\mathcal{R})$ be an SPJ term in the normal form of a view V. Then $\sigma_P(\mathcal{R}) = \pi_{\mathcal{R}.*}\sigma_{NN(\mathcal{R})}V$ if some set of columns from tables in $\mathcal{R}$ constitute a unique key of of the sub-view induced by $\mathcal{R}$.*

Note that the condition is trivially satisfied for the maximal term of V (the term with the maximal set of tables) because the union of the key columns from every source table is always a key of a view.

Proof Consider a tuple t in $\sigma_P(\mathcal{R})$. To prove the theorem we must show that t is output by $\pi_{\mathcal{R}.*}\sigma_{NN(\mathcal{R})}V$ and without duplicates.

Tuple t may be represented in V either explicitly (null extended on tables not in $\mathcal{R}$) or implicitly, that is, it is subsumed by one or more tuples from ancestor terms. Suppose t occurs explicitly in V. Then t must satisfy the predicate $NN(\mathcal{R})$ because it is not null-extended on any table in $\mathcal{R}$. Therefore, t will be output by $\pi_{\mathcal{R}.*}\sigma_{NN(\mathcal{R})}V$.

Suppose t occurs implicitly in V. Without loss of generality, assume that a single column C from a table in $\mathcal{R}$ is a unique key of the sub-view induced by $\mathcal{R}$ and that tuple t has the key value c_1. Let t_1 be a tuple that subsumes t. By definition, the subsuming tuple t_1 must agree with t on all non-null columns of tables in $\mathcal{R}$. This being the case, t_1 must also satisfy the predicate $NN(\mathcal{R})$ because t does. The t_1 also has the value c_1 for column C. Because C is a key of the sub-view containing all possible source terms of t_1, there are no other tuples with the same value in column C. It follows, that t_1 is the only tuple subsuming t and there are no duplicates. The projection of t_1 onto $\mathcal{R}.*$ produces a tuple equal to t. □

Example 7 Consider the following view.

$$V_3 = (\sigma_{lp<20}O) \bowtie^{lo}_{ock=ck} (\sigma_{cn<5}C)$$
$$= \sigma_{cn<5 \wedge ock=ck \wedge lp<20}(C, O) \oplus \sigma_{lp<20}O$$

The tuples of the O term can be recovered from the view by

$$\sigma_{lp<20}O = \pi_{O.*}\sigma_{\neg null(O)}V_3$$

and no duplicate elimination is needed.

The sub-view induced by O is the complete view. Column *ock* is a key of O. Is it also a key of the view? 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 t_o 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, *ock* is a key of the view.

The example illustrates a case with a single extension join. A join between a table R and S is an extension of R if we can deduce that each tuple in R will join with at most one tuple in S. The join merely extends some R tuples with additional columns. It is easy to see that an equijoin against a unique key of S is an extension join.

Theorem 5 *Let $\sigma_P(\mathcal{R})$ be an SPJ term of a view V. If the view outputs a unique key of $\sigma_P(\mathcal{R})$, then the term can be extracted from the view by $\sigma_P(\mathcal{R}) = \delta(\pi_{\mathcal{R}.*}\sigma_{NN(\mathcal{R})}V)$.*

Proof The expression $\pi_{\mathcal{R}.*}\sigma_{NN(\mathcal{R})}V$ 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 $\sigma_P(\mathcal{R})$. We know that one or more copies of t must exist in $\pi_{\mathcal{R}.*}\sigma_{NN(\mathcal{R})}V$. After duplicate elimination, a single copy of t remains because the view outputs a unique key of $\sigma_P(\mathcal{R})$. It follows that the two expressions are equal. □

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 6 *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 $\sigma_{NN(\mathcal{R})}V = \sigma_{NN(\mathcal{S})}V$ if, for every term $\sigma_q(\mathcal{T})$ in the normal form of V such that $\mathcal{R} \not\subseteq \mathcal{T}$, the set $(\mathcal{R} - \mathcal{T}) \cap \mathcal{S}$ is non-empty.*

Proof The purpose of the predicate $NN(\mathcal{R})$ is to reject all tuples that are null-extended on any table of $\mathcal{R}$, that is, tuples originating from any term that is not an ancestor of $\sigma_P(\mathcal{R})$. A term $\sigma_q(\mathcal{T})$ that is not an ancestor of $\sigma_P(\mathcal{R})$ has the property that $\mathcal{R} \not\subseteq \mathcal{T}$. Tuples originating from a term $\sigma_q(\mathcal{T})$ with $\mathcal{R} \not\subseteq \mathcal{T}$ will be null-extended on tables in $(\mathcal{R} - \mathcal{T})$. If $\mathcal{S}$ overlaps with $(\mathcal{R} - \mathcal{T})$ then

the reduced predicate $NN(\mathcal{S})$ will reject all tuples originating from term $\sigma_q(\mathcal{T})$. Consequently, if the condition holds for every term with $\mathcal{R} \not\subseteq \mathcal{T}$, the reduced predicate $NN(\mathcal{S})$ will reject exactly the same tuples as the original predicate $NN(\mathcal{R})$. □

6 The view matching algorithm

We now have the main tools needed to decide whether an SPOJ query can be computed from a SPOJ view. This section pulls them together into a complete view matching algorithm. The focus of this section is to produce provably correct substitute expressions, not necessarily the most efficient ones. Efficiency is addressed in Sect. 7.

If the query can be computed from the view, the algorithm produces, for each query term $E = \sigma_q(\mathcal{R})$, a substitute expression that computes the term from the view. If the query term is mapped to a view term $\sigma_p(\mathcal{T})$ where $\mathcal{R} \subseteq \mathcal{T}$, the substitute expression E^s has the following general form

$$E^s = \delta\pi_{C_2}\sigma_{P_s}\delta\pi_{C_1}\,\sigma_{NN(\mathcal{T})}V. \tag{6}$$

Every component may not be required, of course, for a given query and view. The expression can be simplified but we leave it in the previous form for ease of presentation. A brief explanation of the purpose of each component follows.

1. $\sigma_{NN(\mathcal{T})}$: this selection recovers the tuples of the target term from the view.
2. $\delta\pi_{C_1}$: the recovered result may contain duplicate tuples. If so, they are eliminated by first projecting the tuples onto all available columns of tables in $\mathcal{T}$ and then applying duplicate elimination.
3. σ_{P_s}: tuples that are not required by the query are discarded by a selection with the appropriate residual predicates.
4. π_{C_2}: the result is projected onto all available columns of tables in $\mathcal{R}$.
5. δ: if the target view term does not have the correct duplication factor, duplicate elimination is applied.

Let $E_1^s, E_2^s, \ldots, E_n^s$ be the substitute expressions for the individual terms of a query Q. The complete substitute expression for the query is then

$$Q = \pi_{C_3}(E_1^s \oplus E_2^s \oplus \cdots \oplus E_n^s) \tag{7}$$

where C_3 is the set of output columns of the query.

Here are the high-level steps of the view matching algorithm; the steps are described in more detail in subsequent sections.

Algorithm SPOJ-view-matching:

1. Convert both the query Q and the view V to join-disjunctive normal form.
2. For each term in Q, perform the following steps
   (a) Locate candidate target terms in V.
   (b) Check containment and select a target term in V.
   (c) Check whether the target term can be recovered from V.
   (d) Determine residual query predicates that must be applied.
   (e) Check whether the target term has the right duplication factor.
   (f) Check whether all columns required by residual predicates and output expressions are available in the view output.
   (g) If the term is not the maximal term of the query, check whether the set of available output columns include a unique key of the term.
3. If the view passes all previous tests, construct the substitute expression.

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

$$V_4 = \pi_{lok,ln,lq,lp,ok,od,otp,ck,cn,cnk}\Big(\sigma_{cnk<10}(C) \ltimes^{ro}_{ock=ck} \Big(\sigma_{otp>50}(O) \ltimes^{fo}_{ok=lok} \sigma_{lq<100}(L)\Big)\Big)$$

$$Q_4 = \pi_{lok,lq,lp,od,otp}\left(\sigma_{otp>150}(O) \ltimes^{ro}_{ok=lok} \sigma_{lq<100}(L)\right)$$

6.1 Column equivalence classes

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 a term. Equivalence classes are generated by column equality predicates, typically equijoin conditions. A straightforward algorithm for computing equivalence classes is provided in [8]. We briefly summarize the algorithm here.

First convert the predicate to conjunctive normal form. This step is optional but omitting normalization may cause some column equivalences to be missed. Begin with each column in its own (trivial) equivalence class. Scan the predicate and for each conjunct of the form $(C_i = C_j)$, find the equivalence class containing C_i and the one containing C_j, and merge the two classes. This algorithm can be generalized to capture equivalences implied by predicates of type $C_i \le c \wedge C_i \ge c$ and type $C_i = c$ where c is a constant.

Each term in an SPOJ expression has 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 subsequent operators, in particular, 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 Sect. 8.6).

6.2 Converting to normal form

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

$$
\begin{aligned}
V_4 = {} & \pi_{lok,ln,lq,lp,ok,od,otp,ck,cn,cnk} \\
& (\sigma_{cnk<10 \wedge ck=ock \wedge otp>50 \wedge ok=lok \wedge lq<100}(C,O,L) \oplus \\
& \sigma_{cnk<10 \wedge ck=ock \wedge otp>50}(C,O) \oplus \sigma_{otp>50}(O) \oplus \\
& \sigma_{otp>50 \wedge ok=lok \wedge lq<100}(O,L) \oplus \sigma_{lq<100}(L)) \\
Q_4 = {} & \pi_{lok,lq,lp,od,otp}(\sigma_{otp>150 \wedge ok=lok \wedge lq<100}(O,L) \oplus \\
& \sigma_{lq<100}(L))
\end{aligned}
$$

Only three terms in view V_4 have non-trivial equivalence classes: $\{\{ck, ock\}, \{ok, lok\}\}$ for the (C, O, L) term, $\{\{ck, ock\}\}$ for the (C, O) term, and $\{\{ok, lok\}\}$ for the (O, L) term. In query Q_4, only the (O, L) term has a non-trivial equivalence class, namely, $\{\{ok, lok\}\}$.

6.3 Locating candidate target terms

We now begin processing each term of the normalized query in turn looking for target terms. Let $\sigma_q(\mathcal{R})$ be a term in the query. In most cases it is trivial to find a target term: simply look for a view term with the same base $\mathcal{R}$.

If a view term with a matching base is not found, we simply loop through every term $\sigma_{p_i}(\mathcal{T}_i)$ in V and check whether the term qualifies as a target term. To qualify, the view term must satisfy the two conditions in the definition: (a) $\mathcal{R} \subseteq \mathcal{T}_i$ and (b) there is no other term $\sigma_{p_j}(\mathcal{T}_j)$ in V such that $\mathcal{R} \subseteq \mathcal{T}_j \subseteq \mathcal{T}_i$.

For our example query we have the following (trivial) mapping of terms: the (O, L) term of Q_4 maps to the (O, L) term of V_4; and the L term of Q_4 maps to the L term of V_4.

6.4 Checking containment

A view contains all tuples required by the query if each query term is contained in one of its candidate target terms (Theorem 2). Suppose $\sigma_q(\mathcal{R})$ is a query term and $\sigma_p(\mathcal{T})$ one of its candidate target terms in the view. How to test two terms for containment is described in Sect. 4.

For the (O, L) term in our example query to be contained in the (O, L) term of the view, the following condition must hold

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

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

The L term of the query is contained in the L term of the view if the condition

$$(lq < 100) \Rightarrow (lq < 100)$$

holds, which it does of course. Both terms of the query pass the containment test, which implies that the view contains all tuples required by the query.

6.5 Checking recovery

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

1. Check whether the view outputs sufficient non-null columns (Theorem 6).
2. Check whether duplicate elimination is required (Theorem 4).
3. If duplicate elimination is required, find a unique key of the term (Theorem 5) and check whether the view outputs the required columns.

Our example view V_4 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. Columns $O.otp$ and $L.lq$ are referenced by null-rejecting predicates, which guarantees they will be non-null in the view.

Our example query requires recovery of two terms from the view; we consider each term in turn.

$\{O, L\}$ term, step 1: non-null columns. $O.otp$ is non-null in the view because of the predicate $otp > 50$ and $L.lq$ is non-null because of the predicate $lq < 100$. As the view outputs a non-null column for both source tables (O and L), the conditions of Theorem 6 are automati-

cally satisfied. Hence, the tuples of (O, L) term can be recovered using the predicate $O.otp \neq null$ and $L.lq \neq null$.

$\{O, L\}$ *term, step 2: duplicate elimination.* The sub-view induced by $\{O, L\}$ contains two terms: (O, L) and (C, O, L). We claim that columns $\{ok, ln\}$ constitute a unique key of this sub-view. Clearly, (O, L) tuples are unique on $\{ok, ln\}$. An (O, L) tuple can join with at most one C tuple because the join is an equijoin against ck which is the primary key of table C. Any such joined tuple subsumes its participating (O, L) tuple which will thus be eliminated from the view. Therefore, no two tuples in the sub-view can have the same value on $\{ok, ln\}$. The conditions of Theorem 4 are thus satisfied and no duplicate elimination is needed.

$\{O, L\}$ *term, step 3: unique key.* This step is not needed.

$\{L\}$ *term, step 1: non-null columns.* $L.lq$ is non-null output column of the view so the conditions of Theorem 6 are automatically satisfied. If follows that we can extract the tuples of the L term using the predicate $L.lq \neq null$.

$\{L\}$ *term, step 2: duplicate elimination.* The sub-view induced by $\{L\}$ contains three terms: L, (O, L), and (C, O, L). Columns $\{lok, ln\}$ are a unique key of this sub-view because $\{lok, ln\}$ is a key of the L term and both tables O and C are joined in by means of an equijoin against a unique key (ok and ck, respectively). In other words, an L tuple joins with at most one O tuple and at most one C tuple. Consequently, no duplicate elimination is required for this term either.

$\{L\}$ *term, step 3: unique key.* This step is not needed.

We have thus determined that the tuples of the two target terms can be recovered from the view as follows:

$$\sigma_{otp>50 \wedge ok=lok \wedge lq<100}(O, L) = \sigma_{otp\neq null \wedge lq \neq null} V_4$$
$$\sigma_{lq<100}(L) = \sigma_{lq \neq null} V_4$$

6.6 Residual predicates

The query predicates may be more restrictive than the view predicates. We must eliminate all tuples that do not satisfy the query predicate but to do so we may not need to apply the complete query predicate; the parts of the query 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 $P_q = p_1 \wedge p_2 \wedge \cdots \wedge p_n$ (in conjunctive normal form) and a target 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 algorithm described in [8].

Applying this to the (O, L) 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 the first implication does not hold but the second and third implications do. Hence, the residual predicate for the (O, L) term is $(otp > 150)$.

For the L term we get the implication

$$(lq < 100) \Rightarrow (lq < 100)$$

which trivially holds. Hence, no residual predicate needs to be applied for this term.

6.7 Checking duplication factor

After we have applied any residual predicates to the (recovered) tuples of the target term, the result contains all tuples of the query term but some tuples could be duplicated. This can happen only when the target term references more tables than the query term.

Let $\sigma_q(\mathcal{R})$ be a query term and $\sigma_p(\mathcal{T})$, $\mathcal{R} \subseteq \mathcal{T}$ its target term in the view. At this point we know that all tuples of the target term can be recovered from the view, we have proven containment, and deduced what residual predicate, if any, to apply. Duplicate elimination is not required when $\mathcal{R} = \mathcal{T}$. Otherwise, there are two cases to consider.

Case 1: $\mathcal{R} \subset \mathcal{T}$ and some set of columns from tables in $\mathcal{R}$ constitute a unique key of both terms. If so, each query tuple will occur exactly once and no further action is needed. Note that it is not necessary that the view outputs the key columns – the fact that a common unique key exists is sufficient.

Case 2: $\mathcal{R} \subset \mathcal{T}$ but the two terms do not have a common unique key. If so, some query tuples may be duplicated in the result and we have to perform duplicate elimination. Suppose the view outputs columns $\mathcal{C}$ from tables in $\mathcal{R}$ and possibly some additional columns from the extra tables $\mathcal{T} - \mathcal{R}$. We first project the result onto $\mathcal{C}$ and then apply duplicate elimination. This will produce the correct result only if $\mathcal{C}$ contains a unique key of the query term $\sigma_q(\mathcal{R})$.

In our example query and view, both query terms are mapped to target terms with the same base as the query terms so duplicate elimination is not required.

6.8 Column availability

The columns available in the view output are *lok*, *ln*, *lq*, *lp*, *ok*, *od*, *otp*, *ck*, *cn*, and *cnk*.

The (O, L) term requires a residual predicate: ($otp >$ 150). The *otp* is a view output column of the view so the predicate can be applied. The L term requires no residual predicate.

The query output columns are *lok*, *lq*, *lp*, *od*, *otp*, which are all available as view output columns. Hence, all required columns are available.

In our example, the view outputs all columns referenced by residual predicates and by the output expressions of the query so there is no need to exploit column equivalence classes. However, this is not always the case.

6.9 Unique key availability

Our example query contains two terms, (O, L) and L. The substitute expressions for the two terms have the form

$$E^s_{OL} = \pi_{lok,ln,lq,lp,ok,od,otp}\ \sigma_{p_{OL}} V_4$$
$$E^s_L = \pi_{lok,ln,lq,lp}\ \sigma_{p_L} V_4$$

The exact form of the predicates p_{OL} and p_L is unimportant for the moment. The final substitute expression will be of the form

$$Q = \pi_{lok,lq,lp,od,otp}(E^s_{OL} \oplus E^s_L)$$

The purpose of the minimum union is to eliminate tuples in the E^s_L term that are subsumed by tuples in the E^s_{OL} term. Note that the final projection does not include column *ln*.

To guarantee that the minimum union produces a correct result, the output columns of E^s_L must include a unique key of the query's L term, that is, columns *lok* and *ln*. To see why, consider two tuples $t_a = (a_1, a_2, a_3, a_4)$ and $t_b = (b_1, b_2, b_3, b_4)$ output by E^s_L. Suppose table O contains a tuple that joins with t_a but none that joins with t_b. In other words, t_a is subsumed but t_b is not. Then E^s_{OL} contains a tuple $s_a = (a_1, a_2, a_3, a_4, c_1, c_2, c_3)$ but no tuple $s_b = (b_1, b_2, b_3, b_4, \ldots,)$. The minimum union will then eliminate t_a but not t_b. The reason is that t_b has the unique key value b_1, b_2 and, because t_b did not join with any O tuple, no tuple in E^s_{OL} has the value b_1, b_2 in columns *lok* and *ln*.

But now suppose the view does not output column *ln*, that is, it outputs only *lok*, *lq*, *lp*, *ok*, *od*, *otp*, *ck*, *cn*, and *cnk*. The E^s_L term then contains the tuples $t'_a = (a_1, a_3, a_4)$ and $t'_b = (b_1, b_3, b_4)$ and the E^s_{OL} term contains the tuple $s'_a = (a_1, a_3, a_4, c_1, c_2, c_3)$. It may happen that $t'_a = t'_b$; this is allowed because the tuples do not include a unique key. If so, the minimum union will eliminate not only t'_a but also t'_b producing an incorrect result. The problem is caused by the fact that the output of the view does not include a unique key of the O term.

Here is how we can ensure that the minimum union eliminates only subsumed tuples and no more. Let E_i be a query term and K_i a set of columns forming a unique key of E_i. If E_i is not the maximal term of the query, E^s_i may contain subsumed tuples, which need to be eliminated. The minimum union will produce the correct result if the output columns of the substitute expressions for E_i and all its parent terms include K_i. If so, a tuple t in E^s_i can only match with a tuple s in a parent term if s is the join of t with some other tuples.

In our example query, the *lok* and *ln* form a unique key of the L term. To ensure a correct result the output columns of substitute expressions E^s_L and E^s_{OL} are required to include *lok* and *ln*.

Table 1 Components of term substitute expressions for Q_4.

Component	(O, L) term	L term
$\sigma_{NN(\mathcal{T})}$	$\sigma_{otp\neq null \wedge lq\neq null} V_4$	$\sigma_{lq\neq null} V_4$
$\delta\pi_{C_1}$		
σ_{P_s}	$\sigma_{otp>150}$	
π_{C_2}	$\pi_{lok,ln,lq,lp,od,otp}$	$\pi_{lok,ln,lq,lp}$
δ		

6.10 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 final substitute expression, that is, an expression that computes the query from the view.

Expressions 6 and 7 in the beginning of this section showed the components and overall structure of the substitute expression. Table 1 lists the actual components for the two terms in our example query. Components that are not needed are shown as blank.

Combining the previous components, we obtain the following substitute expression for query Q_4.

$$\begin{aligned} Q_4 &= \pi_{lok,lq,lp,od,otp} \\ &\quad (\pi_{lok,ln,lq,lp,ok,od,otp}\ \sigma_{otp>150}\sigma_{otp\neq null\wedge lq\neq null} V_4\ \oplus \\ &\quad\ \pi_{lok,ln,lq,lp}\sigma_{lq\neq null} V_4) \\ &= \pi_{lok,lq,lp,od,otp} \\ &\quad (\pi_{lok,ln,lq,lp,ok,od,otp}\ \sigma_{otp>150\wedge otp\neq null\wedge lq\neq null} V_4\ \oplus \\ &\quad\ \pi_{lok,ln,lq,lp}\sigma_{lq\neq null} V_4) \end{aligned}$$

7 Efficient substitute expressions

While correct, 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 outer unions. After this conversion, the number of scans of the view can often be reduced by combining terms. In many cases, but not always, a single scan of the view is sufficient.

Example 8 Let us analyze what tuples remain in the result of Q_4 after the minimum union has been applied.

Any tuple t_1 of V_4 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.

To see why, suppose t_2 satisfies the predicates of both terms. Then the first term outputs the tuple

$$s = \pi_{lok,ln,lq,lp,od,otp} t_2$$

and the second term outputs

$$s' = \pi_{lok,ln,lq,lp,null,null} t_2.$$

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_4 so no other tuple with the key value $(t_2.lok, t_2.ln)$ exists in V_4.

It follows that we can rewrite the substitute expression as

$$\begin{aligned} Q_4 = {} & \pi_{lok,lq,lp,od,otp} \\ & (\pi_{lok,ln,lq,lp,od,otp}\, \sigma_{otp>150 \wedge otp\neq null \wedge lq\neq null} V_4 \ \uplus \\ & \pi_{lok,ln,lq,lp}\, \sigma_{lq\neq null \wedge \neg(otp>150 \wedge otp\neq null \wedge lq\neq null)} V_4) \\ = {} & \pi_{lok,lq,lp,od,otp}\, \sigma_{otp>150 \wedge otp\neq null \wedge lq\neq null} V_4 \ \uplus \\ & \pi_{lok,lq,lp}\, \sigma_{lq\neq null \wedge \neg(otp>150 \wedge otp\neq null \wedge lq\neq null)} V_4 \end{aligned}$$

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 an outer union.

This substitute expression still 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 of them. Hence, a view tuple will contribute at most one tuple to the result and we can select all the required tuples by the expression

$$\sigma_{(otp>150 \wedge otp\neq null \wedge lq\neq null) \vee (lq\neq null \wedge (otp\leq 150 \vee otp=null))} V_4.$$

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*, *lq*, and *lp*. This can be done using the case statement of SQL, which has the following syntax

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

where $P_1, \ldots, P_n$ are predicates and $E_1, \ldots, E_{n+1}$ 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

$$\begin{aligned} Q_4 = {} & \pi_{lok,lq,lp,cst} \\ & \sigma_{(otp>150 \wedge otp\neq null \wedge lq\neq null) \vee (lq\neq null \wedge (otp\leq 150 \vee otp=null))} V_4 \end{aligned}$$

where

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

After simplification of the complex-looking selection predicate we obtain

$$Q_4 = \pi_{lok,ln,lq,lp,cst}\, \sigma_{lq\neq null} V_4,$$

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

7.1 Minimum union to outer union

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

While the substitute expression shown in Eq. (6) is correct, it is unnecessarily complex for actual computation. We simplify it by first pushing the selection σ_{P_s} past the duplicate elimination and projection.

$$E^s = \delta\pi_{C_2}\sigma_{P_s}\delta\pi_{C_1}\ \sigma_{NN(\mathcal{T})}V$$
$$= \delta\pi_{C_2}\delta\pi_{C_1}\ \sigma_{P_s\wedge NN(\mathcal{T})}V$$

The operators $\delta\pi_{C_1}$ are now redundant and can be eliminated, which gives us the following simplified form the substitute expression for a term

$$E^s = \delta\pi_{C_2}\sigma_{P_s\wedge NN(\mathcal{T})}V. \quad (8)$$

7.1.1 Terms without duplicate elimination

We first consider substitute expressions without duplicate elimination, that is, expressions of the form

$$E^s = \pi_{C_2}\ \sigma_{P_s\wedge NN(\mathcal{T})}V.$$

The selection predicate $P_s \wedge NN(\mathcal{T})$ determines which tuples of the view qualify for this query term. If a tuple t qualifies, its projection onto C_2 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 the columns of the query). The following theorem shows that we can compute $\hat{E}^s$ by selecting from V only those tuples that satisfy $P_s \wedge NN(\mathcal{T})$ but not the predicates of the parent terms (in the subsumption graph).

Theorem 7 *Let $E_i^s = \pi_{C_{i,2}}\ \sigma_{p_i}V$ be a substitute expression for a term E_i of an SPOJ query. Provided that $C_{i,2}$ contains a unique key of E_i^s, the net contribution of E_i^s can be computed as*

$$\hat{E}_i^s = \pi_{C_{i,2}}\ \sigma_{p_i\wedge\neg p_{i_1}\wedge\neg p_{i_2}\wedge\cdots\wedge\neg p_{i_k}}V$$

where $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ are the selection predicates of the substitute expressions of the parent terms of E_i.

Proof We first prove that no tuples retained by the selection $\sigma_{p_i\wedge\neg p_{i_1}\wedge\neg p_{i_2}\wedge\cdots\wedge\neg p_{i_k}}V$ will be subsumed. Consider a tuple $t \in V$ that satisfies the selection predicate. Tuple t is contained in E_i^s because it satisfies p_i. However, t is not contained in 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 $\pi_{C_{i,2}}t$ is not subsumed by a tuple generated by an ancestor of E_i^s.

The only other possibility is that tuple $\pi_{C_{i,2}}t$ is subsumed by (the projection of) another tuple $t' \in E_i^s$. However, this is not possible because $C_{i,2}$ contains a unique key of E_i^s so no other tuple in E_i^s can match $\pi_{C_{i,2}}t$.

We must also prove that all tuples eliminated by the selection $\sigma_{p_i\wedge\neg p_{i_1}\wedge\neg p_{i_2}\wedge\cdots\wedge\neg p_{i_k}}V$ 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 it clearly should not be output by $\hat{E}_i^s$. The other possibility is that t satisfies one of the parent predicates p_{i_k}. Without loss of generality, assume that it satisfies p_{i_1}. This being the case, t is output by $E_{i_1}^s$. But $C_{i,2} \subseteq C_{i_1,2}$ so $\pi_{C_{i_1,2}}t$ will subsume $\pi_{C_{i,2}}t$. This proves that all tuples that are eliminated would have been subsumed. □

7.1.2 Terms with duplicate elimination

Example 9 Consider the following query

$$Q_5 = \pi_{ok,od,otp,ln,lq}(\sigma_{otp>50}(O) \bowtie^{lo}_{ok=lok}\ \sigma_{lq<10}(L))$$
$$= \pi_{ok,od,otp,ln,lq}(\sigma_{otp>50}(O) \oplus$$
$$\sigma_{otp>50\wedge ok=lok\wedge lq<10}(O,L))$$

Following the procedure in previous sections we find that the query can be computed from view V_4. Duplicate elimination is needed to recover the O term but not for the OL term. The substitute expression is

$$Q_5 = \pi_{ok,od,otp,ln,lq}(E_O^s \oplus E_{OL}^s)$$
where
$$E_O^s = \delta\ \pi_{ok,od,otp}\ (\sigma_{\neg null(O)}V_4)$$
$$E_{OL}^s = \pi_{ok,od,otp,lok,ln,lq,lp}\ \sigma_{lq<10\wedge\neg null(L)\wedge\neg null(O)}V_4$$

The minimum union is necessary because some tuples of E_O^s may be subsumed by tuples produced by its parent term E_{OL}^s. The subsumed tuples have been eliminated in the following $\hat{E}_O^s$. $\hat{E}_{OL}^s = E_{OL}^s$ because E_{OL}^s is the maximal term of the query and contains no subsumed tuples. After this, the minimum union can be converted to an outer union, producing the following substitute expression.

$$Q_5 = \pi_{ok,od,otp,ln,lq}(\hat{E}_O^s \uplus \hat{E}_{OL}^s)$$
$$\hat{E}_{OL}^s = E_{OL}^s$$
$$\hat{E}_O^s = \pi_{ok,od,otp}\Big(\sigma_{cs2=0}\ \gamma^{cs2=sum(s2)}_{ok,od,otp}$$
$$\pi_{ok,od,otp,s2}\ (\sigma_{\neg null(O)}V_4)\Big)$$

where

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

The reason why expression $\hat{E}_O^s$ contains no duplicate tuples is most easily understood by analyzing the expression step by step.

1. The selection $\sigma_{\neg null(O)} V_4$ extracts from V_4 all tuples that qualify for E^s_O.
2. 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^s_{OL}.
3. 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^s_{OL}.
4. If $cs2 > 0$, the group's output tuple is subsumed by at least one tuple in E^s_{OL} and should be eliminated. This is done by the selection $\sigma_{cs2=0}$.
5. The final projection just eliminates the added column $cs2$.

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

Theorem 8 *Let $E^s_i = \delta\, \pi_{C_{i,2}}\, \sigma_{p_i} V$ be the substitute expression for a term E_i. The net contribution of E_i to the query result can then be computed as*

$$\hat{E}^s_i = \pi_{C_{i,2}}\, \sigma_{csp=0}\, \gamma^{csp=sum(x)}_{C_{i,2}}(\sigma_{p_i} V)$$

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

Proof Consider a tuple t in V that satisfies p_i and hence $t' = \pi_{C_{i,2}} t \in E^s_i$.

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, $s' = \pi_{C_{a,2}} s$. Because s' subsumes t', they must agree on columns $C_{i,2}$, that is, $\pi_{C_{i,2}} t = \pi_{C_{i,2}} s$. We must show that t will not be output by the expression in the theorem.

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_{i_1}. Its substitute expression $E^s_{i_1}$ contains a selection with predicate p_{i_1}. In that case, tuple s also satisfies predicate p_{i_1} so when evaluated on s, the case statement returns 1. Because $\pi_{C_{i,2}} t = \pi_{C_{i,2}} s$, 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 nonzero 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 $C_{i,2}$, that is, $\pi_{C_{i,2}} s \neq t'$, or satisfy none of the parent predicates $p_{i_j}, j = 1, \ldots, k$. The tuples with the first property, that is, those that satisfy $\pi_{C_{i,2}} s \neq t'$, 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 outer unions. 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 9 *Consider substitute expressions $\hat{E}^s_i = \pi_{C_{i,2}}\, \sigma_{p_i} V$ and $\hat{E}^s_j = \pi_{C_{j,2}}\, \sigma_{p_j} V$ that compute the net contribution of term E_i and E_j. If $p_i \wedge p_j = false$ then*

$$\hat{E}^s_i \uplus \hat{E}^s_j = \pi_c\, \sigma_{p_i \vee p_j} V$$

where $c =$ (case when p_i then $C_{i,2}$ else $C_{j,2}$ end).

Proof If $p_i \wedge p_j = false$, 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 $p_i \vee p_j$ 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 terms corresponding to E^s_i and E^s_j, $i \neq j$, is an ancestor of the other term, then $p_i \wedge p_j = false$.*

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 further insight into which terms can be combined, namely, any terms that are connected through a sequence of parent–child relationship, provided that their substitute expressions do not require duplicate elimination. Nodes on a common path in the subsumption graph have this property and can thus be combined into the same scan.

More generally, any set of non-overlapping paths that together cover all nodes of the graph provides a valid set

of scans. There may be several valid 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 more practical.

In the special case that the subsumption graph contains a single path and no substitute expression requires duplicate elimination, 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 of the subsumption graph as already covered, and then covering the remaining nodes as outlined earlier.

8 Aggregation views

We now turn to outer-join views with aggregation, that is, views defined by an SPOJ expression and a single group-by operation on top. For non-aggregated views we recovered target terms one by one. For aggregation views this is normally not feasible because tuples from different terms may have been merged together into the same group. If so, contributions from different terms can no longer be separated.

8.1 Class of substitute expressions

Consider an aggregation query and view defined as follows

$$Q = \pi_{C_q} \gamma_{G_q}^{A_q} Q^{spj}$$
$$V = \pi_{C_v} \gamma_{G_v}^{A_v} V^{spj}$$

where Q^{spj} and V^{spj} denote the non-aggregated parts of the expressions. We denote the source tables of Q by $\mathcal{R}$ and those of V by $\mathcal{T}$, $\mathcal{R} \subseteq \mathcal{T}$.

In this paper we restrict substitute expressions for aggregation queries and views to the following form

$$Q' = \pi_{C_q}\ \gamma_{G_q}^{A'_q} \sigma_{P_s} \sigma_{P_r} V.$$

The operator σ_{P_r} recovers all required terms (as a group), σ_{P_s} applies residual predicates, if any, $\gamma_{G_q}^{A'_q}$ performs further aggregation, if needed, and π_{C_q} projects the result onto the same columns as the query. A particular query and view may not require every operator, of course.

To guarantee a correct result we must construct the substitute expression so that $Q = Q'$ for all valid database instances. Clearly, the query, the view, and the operators in the substitute expression must satisfy several conditions for this to hold. Determining a sufficient set of such conditions is the goal of the rest of this section. The conditions we impose will be sufficient but we do not claim that they are necessary.

The *first condition* we impose is containment of the non-aggregated query in the non-aggregated view, that is, $Q^{spj} \subseteq_s V^{spj}$. Given the class of substitute expression we consider, this condition is necessary if there are no constraints on the database.

We now expand the expression for Q' by plugging in the expression for V, which produces

$$Q' = \pi_{C_q} \gamma_{G_q}^{A'_q} \sigma_{P_s} \sigma_{P_r} (\pi_{C_v} \gamma_{G_v}^{A_v} V^{spj})$$

The *second condition* is that predicates P_r and P_s, and the set G_q of grouping columns can only reference non-aggregated output columns of the view, that is, columns in the set $C_v \cap G_v$. Similarly, A'_q can only reference columns in C_v.

For non-aggregated views we recovered target terms from the query one by one. In general, this is not possible for aggregated views because tuples from different terms may have been combined into a single output tuple for a group. Once tuples have been merged in this way, the contributions from different terms can no longer be separated. A different approach is needed.

For recovery purposes, we divide the terms of the view into two sets: terms required by the query and excess terms, that is, terms not required. A term is required if it may contain tuples that contribute to the query result.

Definition 5 *A term $\sigma_p(\mathcal{T})$ of a view V is* required by a query Q *if the query contains a term $\sigma_q(\mathcal{R})$ such that $\mathcal{R} \subseteq \mathcal{T}$. A term of V that is not required by the query is called an* excess term.

It is easy to see that the required terms include all target terms of the query.

Without loss of generality, suppose the query requires the first $k, k \leq n$ terms of the view. (We can always reorder terms so this is true.) We rewrite the view expression as

$$V = \pi_{C_v}\ \gamma_{G_v}^{A_v} (V_R^{spj} \oplus V_E^{spj})$$

where

$$V_R^{spj} = \sigma_{p_1}(\mathcal{T}_1) \oplus \cdots \oplus \sigma_{p_k}(\mathcal{T}_k)$$
$$V_E^{spj} = \sigma_{p_{k+1}}(\mathcal{T}_{k+1}) \oplus \cdots \oplus \sigma_{p_n}(\mathcal{T}_n).$$

Now assume that we can rewrite the view expression in the following form

$$V = \pi_{C_v}\ \gamma_{G_v}^{A_v} (V_R^{spj}) \cup \pi_{C_v}\ \gamma_{G_v}^{A_v} (V_E^{spj}). \quad (9)$$

Furthermore, assume that we can construct a predicate P_r such that

$$\pi_{C_v}\ \gamma_{G_v}^{A_v}(V_R^{spj}) = \sigma_{P_r}V. \tag{10}$$

where P_r references only columns in $C_v \cap G_v$. Then the selection $\sigma_{P_r}V$ recovers the contribution of all required terms to the view result and completely eliminates the contributions of excess terms.

Our *third condition* is that the query and view are such that Eqs. (9) and (10) are valid. In Sect. 8.3 we show how to determine whether the query and view satisfy the necessary requirements and how to construct predicate P_r.

We now plug in the expression for V on the right-hand side of Eq. (10), which yields

$$\pi_{C_v}\ \gamma_{G_v}^{A_v}V_R^{spj} = \sigma_{P_r}(\pi_{C_v}\ \gamma_{G_v}^{A_v}V^{spj}).$$

Because P_r reference only columns in $C_v \cap G_v$, we can push the selection σ_{P_r} past the project and the aggregation. This produces the equality

$$\pi_{C_v}\ \gamma_{G_v}^{A_v}V_R^{spj} = \pi_{C_v}\ \gamma_{G_v}^{A_v}\sigma_{P_r}V^{spj}.$$

Both expressions now have exactly the same projection and aggregation so they can be omitted. This yields the equality

$$V_R^{spj} = \sigma_{P_r}V^{spj}. \tag{11}$$

We now return to the expression for Q'. Because P_r and P_s reference only non-aggregated output columns of the view, the selects can be pushed past the projection π_{C_v} and aggregation $\gamma_{G_v}^{A_v}$. The projection π_{C_v} has no effect on the final result and can be discarded. After applying these transformations we have

$$Q' = \pi_{C_q}\gamma_{G_q}^{A'_q}\gamma_{G_v}^{A_v}\sigma_{P_s}\sigma_{P_r}V^{spj}$$

The *fourth condition* is that the compensating aggregation $\gamma_{G_q}^{A'_q}$ is constructed in such a way that $\gamma_{G_q}^{A'_q}\gamma_{G_v}^{A_v} = \gamma_{G_q}^{A_q}$. Applying this equality we can simplify the expression to

$$Q' = \pi_{C_q}\gamma_{G_q}^{A_q}\sigma_{P_s}\sigma_{P_r}V^{spj}$$

We now apply Eq. (11) to eliminate the selection σ_{P_r}, which produces our final expression for Q'

$$Q' = \pi_{C_q}\gamma_{G_q}^{A_q}\sigma_{P_s}V_R^{spj}$$

Recall that the goal is to guarantee that $Q = Q'$, that is,

$$\pi_{C_q}\gamma_{G_q}^{A_q}Q^{spj} = \pi_{C_q}\gamma_{G_q}^{A_q}\sigma_{P_s}V_R^{spj}$$

Both expressions apply the same aggregation and projection, namely, $\pi_{C_q}\gamma_{G_q}^{A_q}$. If the inputs to the aggregation are the same for all database instances, they obviously produce the same result. However, recall that the query expression Q^{spj} is over tables in $\mathcal{R}$, and the view expression V_R^{spj} is over table in $\mathcal{T}$ where $\mathcal{R} \subseteq \mathcal{T}$. That is, the view may include tables that are not referenced by the query.

Because of the assumption that every table has a unique key, Q^{spj} cannot contain duplicate tuples nor can $\sigma_{P_s}V_R^{spj}$. However, suppose we project $\sigma_{P_s}V_R^{spj}$ onto $\mathcal{R}.*$. This will eliminate the unnecessary columns but the result may contain duplicates. This can only happen if the view contains extra tables, that is, if $\mathcal{T} - \mathcal{R} \neq \emptyset$. If so, we cannot guarantee a correct result if the query contains duplicate-sensitive aggregation functions.

So our *fifth and final condition* is that the equality

$$Q^{spj} = \pi_{\mathcal{R}.*}\sigma_{P_s}V_R^{spj}$$

holds.

8.2 Algorithm

The algorithm below lists the high-level steps in our algorithm for matching aggregated outer-join views.

Algorithm Aggregation-view-matching:

1. Convert the non-aggregated part of the query Q and the view V to join-disjunctive normal form.
2. For each term in Q, perform the following steps
   (a) Locate candidate target terms in V.
   (b) Check containment and select a target term in V.
   (c) Determine residual query predicates to be applied.
3. Check whether all required terms can be recovered from V.
4. Check whether the required terms have the right duplication factor.
5. Check whether the residual predicates are the same for all target terms.
6. Check whether further aggregation is needed.
7. Check whether all columns required by residual predicates, grouping columns, and output expressions are available in the view output.
8. If the view passes all previous tests, construct the substitute expression.

Most of the steps in the algorithm are either straightforward or similar to steps in the algorithm for non-aggregated view but a few comments are in order. Steps

1 and 2 are applied to the non-aggregated part of the query. Step 3 is necessary to verify that we can apply the same residual predicate P_s to all tuples recovered from the view. Steps 3 to 6 will be described in more detail in subsequent sections. Steps 7 and 8 are the same as in Algorithm 6 with minor modifications.

8.3 Tuple recovery

As mentioned earlier, tuples originating from different terms may end up in the same group. If so, they will be combined into the group's single result tuple. Once the details have been lost through aggregation, it is no longer possible to reconstitute the contributions from different terms. The following example illustrates the issue.

Example 10 Consider a view with an aggregate over a left outer join of C and O.

$$V_6 = \gamma_{C.cn}^{count(*),sum(otp)}(C \bowtie_p^{lo} O)$$
$$= \gamma_{C.cn}^{count(*),sum(otp)}(C \bowtie_p O \oplus C)$$

View V_6 is grouped on $C.cn$. Because $C.cn$ is not a key of C, a tuple originating from the (C, O) term may well have the same cn value as a tuple originating from the C term. The sum will then include tuples from both terms and the contributions from each term cannot be separated.

$$V_7 = \gamma_{C.cn,O.od}^{count(*),sum(otp)}(C \bowtie_p^{lo} O)$$
$$= \gamma_{C.cn,O.od}^{count(*),sum(otp)}(C \bowtie_p O \oplus C)$$

However, as soon as the grouping columns include a non-null column of O, as in V_7, the two terms can be separated. All tuples originating from the C term are null on $O.od$ while all tuples from the (C, O) term are non-null on $O.od$. Hence, two tuples from different terms are guaranteed to end up in separate groups. Those from the (C, O) term can be extracted by the predicate $O.od \neq null$ and those from the C term by the predicate $O.od = null$.

As described in Sect. 8.1, we divide the terms of the view into two sets: terms required by the query and excess terms, that is, terms not required. A term is required if it may contain tuples that contribute to the query result. The objective is to show that tuples from required terms and tuples from excess terms end up in separate groups.

Suppose the query requires the first $k, k \leq n$ terms of the view. We rewrite the view as

$$V = \pi_{C_v}\, \gamma_{G_v}^{A_v}(V_R^{spj} \oplus V_E^{spj})$$

where

$$V_R^{spj} = \sigma_{p_1}(\mathcal{T}_1) \oplus \cdots \oplus \sigma_{p_k}(\mathcal{T}_k)$$
$$V_E^{spj} = \sigma_{p_{k+1}}(\mathcal{T}_{k+1}) \oplus \cdots \oplus \sigma_{p_n}(\mathcal{T}_n).$$

If the view expression can be converted into the following form, it may be possible to recover only the part of the view that originates from required terms.

$$V = \pi_{C_v}\, \gamma_{G_v}^{A_v}(V_R^{spj}) \cup \pi_{C_v}\, \gamma_{G_v}^{A_v}(V_E^{spj})$$

Even if the view can be rewritten in this way, we may still not be able to separate tuples from the two parts using only the non-aggregated output columns of the view. The following Lemma and Theorem derive sufficient conditions for the rewrite to be valid and for deciding whether the two parts are separable. These are the conditions we use to determine whether the contributions of the required terms can be recovered from the view.

Lemma 6 *Consider an aggregation view V over tables $\mathcal{U}$ that outputs a non-null, non-aggregated column for every table in $\mathcal{S}$, $\mathcal{S} \subseteq \mathcal{U}$. Let $\sigma_P(\mathcal{T})$ be an SPJ term of V that is required by a query Q. Then $\sigma_P(\mathcal{T}) \subseteq \sigma_{NN(\mathcal{S}\cap\mathcal{T})}V^{spj}$ and $\sigma_{NN(\mathcal{S}\cap\mathcal{T})}V$ does not include contributions from excess terms of V if, for every excess term $\sigma_q(\mathcal{R})$, the set $(\mathcal{U} - \mathcal{R}) \cap \mathcal{S} \cap \mathcal{T}$ is non-empty.*

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

We must also prove that $\sigma_{NN(\mathcal{S}\cap\mathcal{T})}V^{spj}$ contains no tuples from excess terms. Consider an excess term $\sigma_q(\mathcal{R})$. The predicate $NN(\mathcal{S} \cap \mathcal{T})$ will reject all tuples of the term if they are null-extended on at least one table in $\mathcal{S} \cap \mathcal{T}$. All tuples of the term on null-extended on the tables in $(\mathcal{U} - \mathcal{R})$. It follows that they will be eliminated if $(\mathcal{U} - \mathcal{R})$ and $\mathcal{S} \cap \mathcal{T}$ have at least one table in common, which is the condition in the theorem. As the condition holds for all excess terms, the result of $\sigma_{NN(\mathcal{S}\cap\mathcal{T})}V^{spj}$ cannot contain tuples from any excess term. □

Theorem 10 *Let $\sigma_{p_i}(\mathcal{T}_i), i = 1, 2, \cdots, k$ be the terms of a view V that are required by a query Q. If every required term satisfies the conditions in Lemma 6, then*

$$\pi_{C_v}\, \gamma_{G_v}^{A_v}(\sigma_{p_1}(\mathcal{T}_1) \oplus \cdots \oplus \sigma_{p_k}(\mathcal{T}_k)) = \sigma_{P_r}V$$

where $P_r = \bigvee_{i=1}^{k} NN(\mathcal{T}_i \cap \mathcal{S})$

Each component of predicate P_r 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. Typically, P_r can be significantly simplified by repeatedly applying the rule $NN(\mathcal{T}_i) \vee NN(\mathcal{T}_j) = NN(\mathcal{T}_i)$ if $\mathcal{T}_i \subseteq \mathcal{T}_j$.

Proof The view outputs at least one non-aggregated, non-null column for every table in $\mathcal{S}$. Select one such column from each table in $\mathcal{S}$ and denote the resulting set of columns by C_{NN}. Clearly, $C_{NN} \subseteq G_v$.

Consider a tuple t that originates from one of the required terms. Without loss of generality, assume that $t \in \sigma_{p_1}(\mathcal{T}_1)$. Then tuple t satisfies the predicate $NN(\mathcal{T}_1)$. Clearly, t must also satisfy the weaker predicate $NN(\mathcal{T}_j \cap \mathcal{S})$. After grouping, t will be included in the group $\pi_{G_v}t$. But $\pi_{G_v}t$ must also satisfy the predicate $NN(\mathcal{T}_j \cap \mathcal{S})$ because all columns referenced by the predicate are available in G_v and in C_v. In summary, the contribution of tuple t will be included in the group $\pi_{G_v}t$ and the tuple for this group satisfies P_r. This proves that every tuple from the required terms will be included in some group satisfying P_r.

Now consider a tuple t that originates from an excess term. We prove that t does not satisfy predicate P_r by contradiction. Suppose t satisfies P_r. First, assume that it satisfies the first component $NN(\mathcal{T}_1 \cap \mathcal{S})$ of P, which corresponds to the required term $\sigma_{p_1}(\mathcal{T}_1)$. However, this term satisfies the requirements of Lemma 6 which guarantees that the selection $\sigma_{NN(\mathcal{T}_1 \cap \mathcal{S})}V$ does not contain contributions from excess terms. Hence, t cannot satisfy predicate $NN(\mathcal{T}_1 \cap \mathcal{S})$. The same argument can be applied to show that t cannot satisfy any predicate $NN(\mathcal{T}_i \cap \mathcal{S})$ in P_r. It follows that t cannot satisfy predicate P_r. Consequently, t does not contribute to any of the groups selected by P_r. □

8.4 Correct duplication factor

Theorem 10 shows how to recover the contribution of required terms from the view. However, the view may include tables that are not referenced in the query. These extra tables may change the duplication factor, that is, the terms recovered from the view may not contain the same number of duplicates of each row as the corresponding view terms. If so, any duplicate-sensitive aggregate function, for example, `sum` or `count`, taken from the view would be incorrect.

As detailed in Sect. 8.1, this can be handled by ensuring that the following equality holds

$$Q^{spj} = \pi_{\mathcal{R}.*}\, \sigma_{P_s} V_R^{spj}.$$

Recall that we assume that every table has a unique key. Let $\kappa(T_i)$ denote a unique key of table T_i. We extend this notation to sets of tables and define $\kappa(\mathcal{T}) = \cup_{t \in \mathcal{T}} \kappa(t)$, that is, the union of the unique keys of all tables in the set. For a non-aggregated SPOJ expression E over tables $\mathcal{T}$, $\kappa(\mathcal{T})$ is always a unique key of E. (A subset of $\kappa(\mathcal{T})$ may also suffice but that is not important here.) This can easily be proven by deriving $\kappa()$ bottom-up through the operators that may occur in an SPOJ expression.

Theorem 11 *Consider a query Q defined over tables in $\mathcal{R}$ and view V that satisfy the first four conditions in Sect. 8.1. Then the equality $Q^{spj} = \pi_{\mathcal{R}.*}\sigma_{P_s}V_R^{spj}$ holds if $\kappa(\mathcal{R})$ is a unique key of V_R^{spj}.*

Proof Q^{spj} is defined over $\mathcal{R}$, so $\kappa(\mathcal{R})$ is a unique key of Q^{spj}. Because the query and view satisfy the first four conditions in Sect. 8.1, we know that every tuple in Q^{spj} also occurs in $\pi_{\mathcal{R}.*}\sigma_{P_s}V_R^{spj}$ and vice versa. $\kappa(\mathcal{R})$ is a unique key of V_R^{spj} so no two tuples in V_R^{spj} can have the same value key value. Because $\kappa(\mathcal{R})$ only involves column in $\mathcal{R}$, $\kappa(\mathcal{R})$ must also be a unique key of $\pi_{\mathcal{R}.*}\sigma_{P_s}V_R^{spj}$ if it is a unique key of V_R^{spj}. It follows that every tuple in Q^{spj} occurs exactly once in $\pi_{\mathcal{R}.*}\sigma_{P_s}V_R^{spj}$ and the equality holds. □

8.5 Residual predicates

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

Residual predicates are determined in Step 2(c) of our view matching algorithm. If we find that the same residual predicate is to be applied to all target terms and the necessary columns are output by the view, this common residual predicate can be combined with the recovery predicate.[2]

8.6 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

[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.

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.

To perform further aggregation it must be possible to rewrite the query's aggregation functions in terms of the output provided by the server. This is a separate issue and because it is unaffected by the type of joins allowed in views and queries we will not discuss it further here.

8.7 Example

Example 11 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, respectively. (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).) lok, ln is a unique key of table L and also of the SPOJ part of the view because both S and O are joined to L through foreign-key joins.

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 source sets (O, L) and O. Columns lok, ln is a unique key of the non-aggregated part of the query because O is joined with L through a foreign-key join.

The (O, L) term of the query is contained in the (S, O, L) term of the view. The additional join with S is a pure extension join, that is, it does not add or delete rows but simply extends each existing row with values from exactly one S tuple. No residual predicate is needed.

The O term of the query is contained in the O term of the view. No residual predicate is needed.

We then apply the condition in Theorem 10 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 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. We also need to verify that the duplication factor of the view is the same as for the query. For the query we have $\kappa(O, L) = \{ok, lok, ln\}$. Recall that any superset of a unique key is also a unique key. The query has a unique key $\{lok, ln\}$ so clearly $\{ok, lok, ln\}$ is also sufficient. Hence, the query and the view have the same duplication factor.

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 construct substitute expression and manually rewrote queries.

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 listed 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
```

The view consists of three terms (C,O,L), (C,O), and C containing 1,897,761 rows, 431,165 rows, and 50,004 rows, respectively. Q_1, Q_2, and Q_4 can be answered by a single scan of V_1 with different predicates. Q_3 requires explicit elimination of subsumed tuples and Q_4 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 c_custkey, c_name, c_nationkey, o_orderkey,
  o_custkey, o_orderdate, o_totalprice
from V1
where o_custkey is not null
group by c_custkey, c_name, c_nationkey, o_orderkey,
  o_custkey, o_orderdate, o_totalprice
```

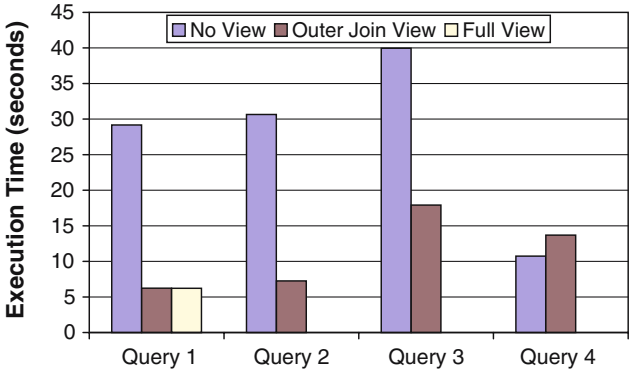

Fig. 2 Execution times for test queries Q_1 to Q_4

Figure 2 compares the performance of the original queries and their corresponding rewrites using V_1. Q_1 can also be answered by an inner-join view over the three tables (with the same join predicates of V_1). Compared with using the inner-join 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 a 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 V_2 to compute total lineitem quantity for every nation, order status, and shipment. The aggregation query Q_5 below 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

Q5:
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

Q5':
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 describe a general view matching algorithm 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 et al. 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 et al. [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.

11 Concluding remarks

This paper provides the first general 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.

In Sect. 7.2 we described when the substitute expressions for two terms can be combined into the same scan. There may be multiple ways to combine term substitutes into scans, which raises the issue of finding the optimal set of scans. This is currently an open issue. Whether or not terms that require duplicate elimination can be combined into the same scan is also an open issue.

For aggregation views we only considered recovering all required terms as a single group. This has the drawback that we can only handle queries where the same residual predicate is to be applied to all required terms. If different residual predicates are to be applied to different terms, we need to recover smaller groups of terms. However, how to determine the smallest recoverable groups of terms is not known at this time.

References

1. Bello, R.G., Dias, K., Downing, A., Feenan Jr., J.J., Finnerty, J.L., Norcott, W.D., Sun, H., Witkowski, A., Ziauddin, M.: Materialized views in Oracle. In: Proceedings of VLDB Conference, pp. 659–664 (1998)

2. Bhargava, G., Goel, P., Iyer, B.R.: Hypergraph based reorderings of outer join queries with complex predicates. In: Proceedings of SIGMOD Conference, pp. 304–315 (1995)
3. Chang, J.-Y., Lee, S.-G.: Query reformulation using materialized views in data warehouse environment. In: Proceedings of DOLAP, pp. 54–59 (1998)
4. Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim, K.: Optimizing queries with materialized views. In: Proceedings of ICDE Conference, pp. 190–200 (1995)
5. Galindo-Legaria,C.: Outerjoins as disjunctions.In: Proceedings of SIGMOD Conference, pp. 348–358 (1994)
6. Galindo-Legaria, C., Rosenthal, A.: Outerjoin simplification and reordering for query optimization. ACM Tran. Database Syst. **22**(1) (1997)
7. Goel, P., Iyer, B.R.: Sql query optimization: reordering for a general class of queries. In: Proceedings of SIGMOD Conference, pp. 47–56 (1996)
8. Goldstein, J., Larson, P.-Å.: Optimizing queries using materialized views: a practical, scalable solution. In: Proceedings of SIGMOD Conference, pp. 331–342 (2001)
9. Larson, P.-Å., Yang, H.Z.: Computing queries from derived relations. In: Proceedings of VLDB Conference, pp. 259–269 (1985)
10. Larson, P.-Å., Zhou, J.: View matching for outer-join views. In: Proceedings of VLDB Conference, pp. 445–456 (2005)
11. Levy, A.Y., Mendelzon, A.O., Sagiv, Y., Srivastava, D.: Answering queries using views. In: Proceedings of PODS Conference, pp. 95–104 (1995)
12. Pottinger, R., Levy, A.Y.: A scalable algorithm for answering queries using views. In: Proceedings of VLDB Conference, pp. 484–495 (2000)
13. Rao, J., Lindsay, B.G., Lohman, G.M., Pirahesh, H., Simmen, D.E.: Using eels, a practical approach to outerjoin and antijoin reordering. In: Proceedings of ICDE, pp. 585–594 (2001)
14. Rao, J., Pirahesh, H., Zuzarte, C.: Canonical abstraction for outerjoin optimization. In: Proceedings of SIGMOD Conference, pp. 671–682 (2004)
15. Srivastava, D., Dar, S., Jagadish, H.V., Levy, A.Y.: Answering queries with aggregation using views. In: Proceedings of VLDB Conference, pp. 318–329 (1996)
16. Yang, H.Z., Larson,P.-Å.: Eager aggregation and lazy aggregation. In: Proceedings of VLDB Conference, pp. 345–357 (1995)
17. Yan, W.P., Larson,P.-Å.: Query transformation for PSJ-queries. In: Proceedings of VLDB Conference, pp. 245–254 (1987)
18. Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., Urata, M.: Answering complex sql queries using automatic summary tables. In: Proceedings of SIGMOD Conference, pp. 105–116 (2000)