The VLDB Journal (2009) 18:599–610
DOI 10.1007/s00778-008-0110-5

REGULAR PAPER

Reducing outer joins

Gerhard Hill · Andrew Ross

Received: 2 August 2006 / Revised: 10 June 2008 / Accepted: 19 June 2008 / Published online: 7 August 2008

Abstract We present a method for transforming some outer joins to inner joins and describe a generalized semijoin reduction technique. The first part of the paper shows how to transform a given outer join query whose join graph is a tree to an equivalent inner join query. The method uses derived relations and join predicates. Derived relations contain columns corresponding to join conditions and may have virtual row identifiers, rows and attribute values. The constructed inner join query, after elimination of virtual row identifiers, has the same join tuples as the outer join query. Both the theoretical maximum number of virtual rows and the average number in practice are shown to be low. The method confines consideration of the non-associativity of outer joins to a single step. The second part of the paper generalizes to outer joins the well known technique of semijoin reduction of inner joins. It does so by defining the notions of influencing and needing, and using them to define full reduction and reduction plans. The technique is applied here to perform one step of the method presented in the first part. Semijoin reduction is useful in practice for executing join queries in distributed databases.

Keywords Outer join evaluation · Virtual row method · Join transformation · Semijoin reduction · Efficient join evaluation

G. Hill · A. Ross (✉)
SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
e-mail: a.ross@sap.com

G. Hill
e-mail: gerhard.hill@sap.com

1 Introduction

In a database management system, evaluating a join query is generally an expensive operation, in terms both of its computational time and of the memory and other resources consumed. For this reason, much research has been done to develop fast and efficient methods to evaluate them. Outer join queries tend to be even more laborious to process than inner join queries, yet fewer algorithms are available for them. Given this fact, we hope that the database community will welcome our new technique for processing outer join queries by transforming them to equivalent inner join queries, which can then be evaluated in any standard way. We developed this method for use by a join engine in the SAP NetWeaver technology platform, where the method has made an essential contribution to the performance boost delivered by the engine. Against this background, we believe the method to be of wider interest.

In Sect. 2, we formulate the join transformation problem and describe our solution, the *virtual row method*. The convention that row identifiers and attribute values are positive integers enables us to use negative integers as *virtual* values in a mechanism for preserving appropriate rows in a join condition. The method consists of seven steps:

1. Choose *preserve numbers*. These are negative integers, pairwise distinct, to be used in derived relations.
2. Define *derived relations*, with columns corresponding to certain join conditions.
3. On the derived relations, define *derived join predicates* with extended domains including virtual rows.
4. Reduce the derived relations fully. It is inessential which means is chosen to do this. One way is described in Sect. 3.
5. Generate *virtual rows*. This step gives the method its name. Whereas a real row might have some virtual attribute

values, a virtual row consists entirely of them. Their function is to control the join tuple assembly of the inner join query so as to produce the same join tuples as in the original query. This is the only step where join execution order (non-associativity) needs to be considered.
6. Assemble the modified data into a modified operator tree that defines an inner join.
7. Eliminate the virtual row identifiers.

We illustrate the method with two examples, both based on the same relations and predicates.

Next, we discuss general properties of the virtual row method. As the theoretical worst case, the exact maximum of the number of virtual rows generated in all relations is determined to be $\frac{1}{2} * (n-1) * (n+2)$ for a join on $n \in \mathbf{N}^+$ relations. This agrees well with the practical average case. Our samples of at least 10,000 random join queries of size $n \leq 100$ showed that maximum and average numbers of virtual rows per relation also grow linearly with n.

In Sect. 3, we show how the well known technique of semijoin reduction of inner joins can be extended to apply to outer joins as well.

To do so, we generalize semijoin reduction steps to outer joins, distinguishing *reducing* from *preserving* moves. The cumulative effect of several such moves is captured in the *influencing* relation. We use a connected subset of the incidences and its underlying subtree of vertexes of the join graph to define the *needed* relation. Thus we define full reduction for any vertex in the join graph, which gives us the notion of a reduction plan. And there we stop, without discussing how to optimize outer join reduction plans.

Semijoin reduction of outer joins provides a way to perform the full reduction of the derived relations mentioned above. Previously, semijoin reduction has been seen primarily as a useful preparatory step for processing inner join queries. However, here we present semijoin reduction as a sufficient tool for evaluation of inner or outer joins. Moreover, it is one that can readily be implemented in a distributed database so as to minimize interhost communication.

In this section, too, we illustrate the new technique with a running example.

A different way to handle the non-associativity of outer joins by introducing generalized outer joins was presented in [5]. We consider our method to be simpler and more efficient.

2 Reducing outer joins to inner joins

In this section, we describe a method for transforming a given outer join to an equivalent inner join. When one compares how easy it is to handle inner joins with how hard it is to manipulate outer joins, as a consequence of their awkward algebraic properties, one may find the method attractive as a theoretical as well as a practical tool.

2.1 Preliminaries

We shall use the following standard sets of integers:

$$\mathbf{N} = \{0, 1, 2, \ldots\}, \mathbf{N}^+ = \{1, 2, 3, \ldots\}$$
$$\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

We write a *null* value of an attribute as zero ($0 \in \mathbf{Z}$).

An *incidence* (V, e) in a graph $\boldsymbol{G}$ is a pair consisting of a vertex $V \in \boldsymbol{G}$ and an edge $e \in \boldsymbol{G}$ such that V is incident with e, that is, V is one of the two vertexes joined by edge e. Two incidences $(V, e), (W, f) \in \boldsymbol{G}$ are *neighbors* iff $V = W$ or $e = f$. Thus, *paths* of incidences and *connectedness* of sets of incidences are also defined. For visualization, an incidence can be depicted as a half-edge.

A join condition j is given by a left relation R_l, a right relation R_r, a join type and a join predicate p. We write j as follows:

$j : R_l \overset{p}{\text{———}} R_r$ or $R_l \overset{p}{\bowtie} R_r$ denotes an inner join,
$j : R_l \xrightarrow{p} R_r$ denotes a left outer join,
$j : R_l \xleftarrow{p} R_r$ denotes a right outer join,
$j : R_l \overset{p}{\longleftrightarrow} R_r$ denotes a full outer join.

The *preserved* relations depend on the join type as follows:

– an inner join preserves no relation,
– a left outer join preserves R_l,
– a right outer join preserves R_r,
– a full outer join preserves R_l and R_r.

We shall use the following three representations of a join:

– as a join expression,
– as a join graph,
– as an operator tree.

A *join expression* is the algebraic representation of a join. It is an expression composed of join conditions. Execution order is indicated by setting parentheses, obeying the usual rules of well formed formulas.

A *join graph* (also called a *query graph*) [6] is obtained by drawing a node for each relation in the join and an edge joining two nodes for each join condition. Nodes are labeled by their relations, and edges display their join type and join predicate as above. The execution order of the join conditions is indicated in a separate sequence; for an inner join this sequence is redundant and may be omitted.

An *operator tree* is obtained by drawing a node for each relation in the join and also a node for each join condition, and labeling the nodes accordingly [1,5]. The relation nodes are the leaf nodes of the operator tree and the join condition nodes are its inner nodes. From each join condition an edge is drawn to each of its operands, which may be relations or further join conditions.

Example Consider four relations Q, R, S, T and three predicates on them:

p_2 on $R \times Q$,
p_3 on $R \times S$,
p_4 on $R \times T$.

Then the query represented by the join expression

$$((Q \xleftarrow{p_2} R) \overset{p_3}{\text{——}} S) \xleftrightarrow{p_4} T \qquad \text{(JF)}$$

may also be represented by the join graph shown in Fig. 1. The join order is given by the sequence j_2, j_3, j_4, where the join conditions are:

$j_2 : Q \xleftarrow{p_2} R$
$j_3 : R \overset{p_3}{\text{——}} S$
$j_4 : R \xleftrightarrow{p_4} T.$

The sequence of join conditions already conveys all the information in the join graph, so if the sequence is given, the graph can be regarded as merely an aid to visualization.

The query can also be represented as an operator tree, in either of the ways shown in Fig. 2.

$$\begin{tikzcd}
 & Q & \\
S \arrow[r, dash, "p_3"] & R \arrow[u, "p_2"'] \arrow[r, leftrightarrow, "p_4"] & T
\end{tikzcd}$$

Fig. 1 Join graph for JE

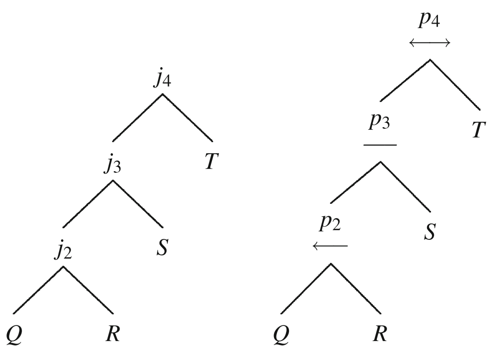


Fig. 2 Two alternative operator trees for JE

2.2 The problem

We are given $n \in \mathbf{N}^+$ relations

$R_1, R_2, \ldots, R_n$

such that every attribute domain is $\mathbf{N}$ and every domain of row identifiers is $\mathbf{N}^+$.

On the relations $R_1, R_2, \ldots, R_n$, an outer join J is given by its operator tree $\boldsymbol{T}$, where the inner nodes of $\boldsymbol{T}$ are $n - 1$ join conditions

$j_2, j_3, \ldots, j_n.$

The order presented is a possible execution order. Each join condition j_k references a predicate p_k, which is *binary* and *null-rejecting* [5].

Both the join graph $\boldsymbol{JG}$ and the operator tree $\boldsymbol{T}$ of $\boldsymbol{J}$ are binary trees, and join $\boldsymbol{J}$ is a tree query [3]. We know that every inner join is equivalent to a tree query [6].

The problem is to transform $\boldsymbol{J}$ into an equivalent inner join.

2.3 The virtual row method

We solve the problem by the *virtual row method* in seven steps.

2.3.1 Choose preserve numbers

In step 1, to every incidence $(R, j) \in \boldsymbol{JG}$ consisting of a relation $R \in \{R_1, R_2, \ldots, R_n\}$ and a join condition j referencing and preserving R, we choose a negative integer

$\pi(R, j) < 0.$

We call $\pi(R, j)$ its preserve number. All preserve numbers chosen must be different. Recalling the join condition sequence $j_2, j_3, \ldots, j_n$, we see that we are in effect just numbering its arrowheads. There can be at most $2n - 2$ preserve numbers.

2.3.2 Create derived relations

In step 2, for every relation $R \in \{R_1, R_2, \ldots, R_n\}$ we create a derived relation R'.

The domain of row identifiers in R' is set as $\mathbf{Z} - \{0\}$. A positive row identifier and its row are called *real*. A negative row identifier and its row are called *virtual*. All rows appearing before step 5 will be real.

The columns of R' correspond to the join conditions j referencing and preserving R. For each such join condition j, the derived relation R' has an attribute $R'.j$. If there are no such join conditions, a row of R' consists of its row identifier alone. The domain of the attributes $R'.j$ is $\mathbf{Z}$. We call positive or null values of $R'.j$ real and negative values virtual.

We initialize R' by inserting into R' for every row in R one row with the same row identifier and all attribute values set to 1. In all the steps, R' will retain the following properties. For every real row $r' \in R'$, there is exactly one row $r \in R$ having the same row identifier. And for every virtual row $r' \in R'$, there is no row $r \in R$ having the same row identifier. These properties could also have served as a definition of real and virtual rows.

To summarize, we have created and initialized n new relations $R'_1, R'_2, \ldots, R'_n$.

2.3.3 Create derived predicates

In step 3, we create derived predicates on the new relations $R'_1, R'_2, \ldots, R'_n$.

Let j be any of the given join conditions $j_2, j_3, \ldots, j_n$, let R, S be the two relations referenced by j, and let

$$p : R \times S \rightarrow \{\text{False, Unknown, True}\}$$

be the predicate of j. We shall use p to define a predicate

$$p' : R' \times S' \rightarrow \{\text{False, Unknown, True}\}$$

on the derived relations R', S'. To this end, let $r' \in R'$, $s' \in S'$ have row identifiers $\rho, \sigma \in \mathbf{Z}$. If $\rho > 0$, let $r \in R$ be the unique row with row identifier ρ, likewise for $\sigma > 0$ and $s \in S$. We now set

$$\begin{aligned} p'(r', s') = {} & (\rho > 0 \wedge \sigma > 0 \wedge p(r, s)) \\ & \vee (j \text{ preserves } R \wedge r'.j = \pi(R, j) = \sigma) \\ & \vee (j \text{ preserves } S \wedge s'.j = \pi(S, j) = \rho) \\ & \vee (\rho < 0 \wedge \sigma < 0 \wedge \rho = \sigma). \end{aligned}$$

Since p was null-rejecting, p' is too. For two real rows

$$p'(r', s') = p(r, s).$$

To summarize, we have created $n - 1$ derived predicates

$$p'_2, p'_3, \ldots, p'_n$$

on the new relations from step 2.

2.3.4 Reduce derived relations

Step 4 consists of two phases. The first phase decreases relation sizes:

```
While there exist
− a join condition $j$, referencing $R$, $S$, not preserving $R$,
− a row $r' \in R'$ such that under the derived predicate $p'$
  on $R' \times S'$, $r'$ matches no row $s' \in S'$
delete $r'$ from $R'$.
```

The second phase sets the right content:

```
For every attribute $R'.j$ of every derived relation $R'$:
  let $j$ reference $R$ and $S$ and $p'$ be the derived predicate
  on $R' \times S'$;
  for every row $r' \in R'$
    if $r'$, under $p'$, matches no row in $S'$
      update $r'.j = \pi(R, j)$
```

We call the first phase full reduction. We need it to ensure that in the second phase the attributes are updated to the correct values. We need these in turn to ensure that the inner join query gives the correct result.

In principle, the virtual row method is independent of the technique used to achieve full reduction and to set the content. The technique we use is described in Sect. 3. Our technique executes the two phases jointly. Whichever technique is used, our formulation shows that reducing the derived relations is independent of the execution order of the join conditions within the given join $\boldsymbol{J}$.

After this step, there will in general be real rows in R' that have virtual attribute values.

2.3.5 Generate virtual rows

Step 5 is the heart of the method. Virtual rows will consist exclusively of virtual attribute values. Their purpose is to serve as artificial join partners for rows to preserve. For any column $R'.j$, the rows to preserve are the rows $r' \in R'$ having $r'.j = \pi(R, j)$.

This is the only step in the entire method for transforming the outer join to an equivalent inner join where the execution order $j_2, j_3, \ldots, j_n$ is critical.

The procedure is:

```
 For $k = 2$ to $n$
 {
   let $j = j_k$ and $R$, $S$ be the two relations $j$ references;
   if $j$ preserves $R$
     $generateVirtualRows(R, j)$;
   if $j$ preserves $S$
     $generateVirtualRows(S, j)$;
 }
using
 $generateVirtualRows(R, j)$
 {
   let $R$, $S$ be the two relations $j$ references;
   let $\pi = \pi(R, j) < 0$;
   let $\pi' = \pi(S, j) < 0$, if $j$ preserves $S$, else let $\pi' = 0$;
   // for any relation $T$ let $T'$ be its derived relation
   1: for all virtual rows $r' \in R'$ with $r'.j \neq \pi'$
     update $r'.j = \pi$;
   2: if there are any rows $r' \in R'$ with $r'.j = \pi$
     for all $T$ in the subtree of $j$ in $\boldsymbol{T}$ containing $S$
       insert into $T'$ the virtual row with row ID and
       all attribute values equal to $\pi$;
 }
```

The task of statement 1 in the procedure is to determine the rows that need action in order to be preserved. For real rows $r' \in R'$, there is nothing to do. Their attribute $r'.j$ has already been marked as 1 or π and they will be preserved. In general, virtual rows must be preserved too. This is achieved by the updates $r'.j = \pi$. Virtual rows inserted later will not be preserved.

However, one possible row must not be preserved. If $\pi' < 0$ and the virtual row Π' with all entries equal to π' is present in R', we know that Π' was generated by the immediately preceding "twin" call *generateVirtualRows*(S, j). It must not be preserved. The purpose of Π' is to ensure that j preserves S for the rows $s' \in S'$ with $s'.j = \pi'$. If we also updated $\Pi'.j = \pi$, then its purpose would be thwarted. Moreover, if we did so, the twin calls *generateVirtualRows*(R, j) and *generateVirtualRows*(S, j) would no longer commute, which they obviously must, and the inner join to be defined in step 6 would contain all null join tuples, which is plainly wrong.

The task of statement 2 is to perform the required action on the rows that need it. The loop over the relations $T \in \boldsymbol{T}$ does this by padding them with null join partners to form join tuples of the join corresponding to the subtree of j containing S.

2.3.6 Define inner join

In step 6, with the virtual rows present in the derived tables, we define the desired inner join. For every given join j_k in J with left relation R_l, right relation R_r, and predicate p_k, we use the derived relations and join predicates to define a corresponding inner join

$$j'_k : R'_l \xrightarrow{p'_k} R'_r.$$

Now, replacing in $\boldsymbol{T}$ every leaf node label R_l by R'_l and every inner node label j_k by j'_k leads to an operator tree $\boldsymbol{T}'$. Clearly $\boldsymbol{T}'$ defines an inner join $\boldsymbol{J}'$. We can evaluate $\boldsymbol{J}'$ by any inner join technique we like.

2.3.7 Eliminate virtual row identifiers

In step 7, we replace all the virtual row identifiers uniformly by nulls. This replacement transforms each join tuple of $\boldsymbol{J}'$ in $R'_1 \times R'_2 \times \cdots \times R'_n$ into a join tuple of $\boldsymbol{J}$ in $R_1 \times R_2 \times \cdots \times R_n$. Conversely, every join tuple of $\boldsymbol{J}$ arises from exactly one join tuple of $\boldsymbol{J}'$. This is what we mean by declaring $\boldsymbol{J}'$ and $\boldsymbol{J}$ to be equivalent.

2.4 Two examples

We illustrate the virtual row method with two examples. Both examples use the four relations Q, R, S, T specified in Table 1. The examples also use the following three predicates a, b, c specified for all $q \in Q, r \in R, s \in S, t \in T$:

Table 1 Examples: four relations

$\boldsymbol{Q}$	A
1	16
2	17
3	18
4	19
5	20

$\boldsymbol{R}$	A	B
1	11	15
2	12	20
3	0	20
4	0	21
5	14	22
6	15	24
7	16	25
8	17	27
9	18	28
10	19	30

$\boldsymbol{S}$	B
1	21
2	23
3	25
4	30
5	0

$\boldsymbol{T}$	C
1	0
2	10
3	15
4	20
5	25

a: $R \times Q \to \{\text{False, Unknown, True}\}$
$a(r, q) = (r.A^2 + q.A^2 \leq 555)$

b: $R \times S \to \{\text{False, Unknown, True}\}$
$b(r, s) = (|r.B - s.B| \leq 1)$

c: $R \times T \to \{\text{False, Unknown, True}\}$
$c(r, t) = (\max(r.A, r.B) = t.C)$.

2.4.1 Example 1

To give the method some hard work to do, we first apply it to a set of full outer joins. We set $p_2 = a$, $p_3 = b$, $p_4 = c$, and define the join conditions

$$j_2 : Q \xleftrightarrow{p_2} R$$
$$j_3 : S \xleftrightarrow{p_3} R$$
$$j_4 : T \xleftrightarrow{p_4} R.$$

We use these join conditions to define the outer join $\boldsymbol{J}_1$ by the operator tree $\boldsymbol{T}_1$ shown in Fig. 3.

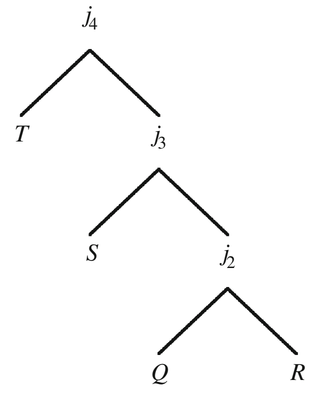


Fig. 3 Operator tree for Example 1

As an algebraic join expression, $\boldsymbol{J}_1$ may be written

$$T \overset{p_4}{\longleftrightarrow} (S \overset{p_3}{\longleftrightarrow} (Q \overset{p_2}{\longleftrightarrow} R)).$$

In this case, the join conditions are associative and brackets are superfluous. However, in general they are not, so we leave the expression as given and process the join conditions in the order indicated.

We now follow the seven steps of the virtual row method. Choosing the preserve numbers arbitrarily, we set

$$\pi(Q, j_2) = -1$$
$$\pi(R, j_2) = -2$$
$$\pi(S, j_3) = -3$$
$$\pi(R, j_3) = -4$$
$$\pi(T, j_4) = -5$$
$$\pi(R, j_4) = -6.$$

The derived relations have the following schemas:

Q'	j_2		
R'	j_2	j_3	j_4
S'	j_3		
T'	j_4.		

Initially all the attribute values are 1.

Next, we define the derived predicates. As an example, take p_2' on $R' \times Q'$. For any rows $r' \in R'$, $q' \in Q'$ with row identifiers $\rho, \kappa \in \mathbf{Z}$ and corresponding rows $r \in R$, $q \in Q$, if these exist, we have

$$\begin{aligned} p_2'(r', q') = & (\rho > 0 \wedge \kappa > 0 \wedge a(r, q)) \vee \\ & (r'.j_2 = -2 = \kappa) \vee \\ & (q'.j_2 = -1 = \rho) \vee \\ & (\rho < 0 \wedge \kappa < 0 \wedge \rho = \kappa). \end{aligned}$$

For the case of two real rows, this is just

$$p_2'(r', q') = a(r, q).$$

Table 2 Derived relations for Example 1

Q'	j_2
1	1
2	1
3	1
4	1
5	1

R'	j_2	j_3	j_4
1	1	−4	1
2	1	1	1
3	−2	1	−6
4	−2	1	−6
5	1	1	−6
6	1	1	−6
7	1	1	1
8	1	−4	−6
9	−2	−4	−6
10	−2	1	−6

S'	j_3
1	1
2	1
3	1
4	1
5	−3

T'	j_4
1	−5
2	−5
3	1
4	1
5	1

Clearly, when we reduce the derived relations, the size of a full outer join is not reduced. The updated relations are shown in Table 2.

Now we generate virtual rows. Since the column $Q'.j_2$ does not contain its preser ve number $\pi(Q, j_2) = -1$, to process j_2 we simply insert the virtual row corresponding to $\pi(R, j_2) = -2$ into Q'.

Notice that under the predicate p_2' the virtual row with row identifier -2 in Q' matches exactly the rows of R' with identifiers 3, 4, 9, 10. Therefore, the inner join

$$j_2' : Q' \overset{p_2'}{\text{——}} R'$$

is equivalent to the full outer join

$$j_2 : Q \overset{p_2}{\longleftrightarrow} R.$$

Next, we process j_3. We insert virtual rows corresponding to $\pi(S, j_3) = -3$ into Q' and R' and the virtual row corresponding to $\pi(R, j_3) = -4$ into S'.

For the same reasons as before, using the join condition

$$j_3' : S' \overset{p_3'}{\text{——}} Q'$$

the inner join

$$S' \overset{p_3'}{\text{——}} Q' \overset{p_2'}{\text{——}} R'$$

is equivalent to the full outer join

$$S \overset{p_3}{\longleftrightarrow} (Q \overset{p_2}{\longleftrightarrow} R).$$

Table 3 Final contents of derived relations for Example 1

Q'	j_2
1	1
2	1
3	1
4	1
5	1
−2	−2
−3	−3
−5	−5

R'	j_2	j_3	j_4
1	1	−4	1
2	1	1	1
3	−2	1	−6
4	−2	1	−6
5	1	1	−6
6	1	1	−6
7	1	1	1
8	1	−4	−6
9	−2	−4	−6
10	−2	1	−6
−3	−3	−3	−6
−5	−5	−5	−5

S'	j_3
1	1
2	1
3	1
4	1
5	−3
−4	−4
−5	−5

T'	j_4
1	−5
2	−5
3	1
4	1
5	1
−6	−6

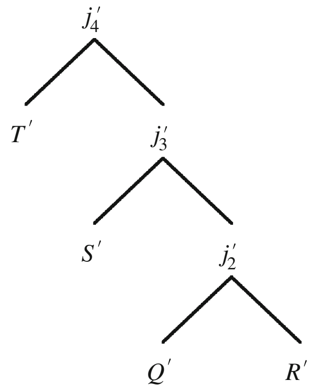


Fig. 4 Equivalent inner join for Example 1

Finally, we process j_4. We insert virtual rows corresponding to $\pi(T, j_4) = -5$ into Q', R', S'. In the j_4 column of R', we update the row with identifier -3 to $\pi(R, j_4) = -6$ and insert a virtual row with identifier -5 into T' to give the final contents of the derived relations as shown in Table 3. We see that under predicate p_4', the row $(-3, -3, -3, -6) \in R'$ matches the row $(-6, -6) \in T'$, whereas under p_3' it continues to match the row $(5, -3) \in S'$ and under p_2' the row $(-3, -3) \in Q'$. This shows that the inner join $\boldsymbol{J}_1'$ shown in Fig. 4 contains the join tuple $(-3, -3, 5, -6) \in Q' \times R' \times S' \times T'$, which upon virtual row identifier elimination corresponds to the join tuple $(0, 0, 5, 0) \in Q \times R \times S \times T$ of $\boldsymbol{J}_1$.

As above, we define the join condition

$$j_4' : T' \overset{p_4'}{\text{---}} R'.$$

We have shown that the inner join $\boldsymbol{J}_1'$ represented by the operator tree $\boldsymbol{T}_1'$ of Fig. 4 is equivalent to the full outer join $\boldsymbol{J}_1$ given by the operator tree $\boldsymbol{T}_1$ of Fig. 3. This the main result of the method. The inner join $\boldsymbol{J}_1'$ has the following 32 join tuples in $Q' \times R' \times S' \times T'$:

$(1, 1, -4, 3), (1, 2, 1, 4), (1, 5, 1, -6), (1, 5, 2, -6),$
$(1, 6, 2, -6), (1, 6, 3, -6), (1, 7, 3, 5), (1, 8, -4, -6),$
$(2, 1, -4, 3), (2, 2, 1, 4), (2, 5, 1, -6), (2, 5, 2, -6),$
$(2, 6, 2, -6), (2, 6, 3, -6), (2, 7, 3, 5), (3, 1, -4, 3),$
$(3, 2, 1, 4), (3, 5, 1, -6), (3, 5, 2, -6), (3, 6, 2, -6),$
$(3, 6, 3, -6), (4, 1, -4, 3), (4, 2, 1, 4), (5, 1, -4, 3),$
$(5, 2, 1, 4), (-2, 3, 1, -6), (-2, 4, 1, -6),$
$(-2, 9, -4, -6), (-2, 10, 4, -6), (-3, -3, 5, -6),$
$(-5, -5, -5, 1), (-5, -5, -5, 2).$

Virtual row identifier elimination leads to the 32 join tuples of $\boldsymbol{J}_1$ in $Q \times R \times S \times T$:

$(1, 1, 0, 3), (1, 2, 1, 4), \ldots, (0, 0, 0, 2).$

Consider for comparison a variant of this example with a different execution order of the same full outer join conditions, such as the join $\boldsymbol{J}_2$ defined by the operator tree $\boldsymbol{T}_2$ of Fig. 5.

Naturally, this leads to the same final result. However, the result is reached in a different way. The inner join $\boldsymbol{J}_2'$ equivalent to $\boldsymbol{J}_2'$ is based on the relations in Table 4.

We see that the number of virtual rows has changed, both for each relation and in total. Also, the three join tuples

$(0, 0, 5, 0), (0, 0, 0, 1), (0, 0, 0, 2)$

of the outer join $\boldsymbol{J}_1 = \boldsymbol{J}_2$ are produced by the inner joins $\boldsymbol{J}_1'$ and $\boldsymbol{J}_2'$ in different ways. Via $\boldsymbol{J}_1'$, they stem from:

$(-3, -3, 5, -6), (-5, -5, -5, 1), (-5, -5, -5, 2).$

Via $\boldsymbol{J}_2'$, they stem from:

$(-2, -3, 5, -3), (-2, -5, -4, 1), (-2, -5, -4, 2).$

2.4.2 Example 2

The second example shows that the virtual row method can also be employed for join operator simplification. Again, we use the relations Q, R, S, T and the predicates $p_2 = a$,

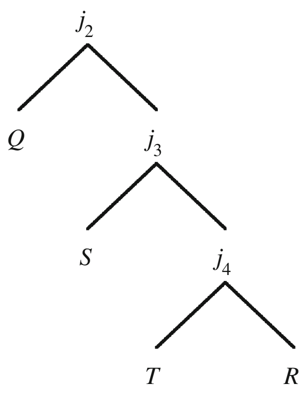


Fig. 5 A variant of Example 1

Table 4 Derived relations for a variant of Example 1

Q'	j_2
1	1
2	1
3	1
4	1
5	1
−2	−2

R'	j_2	j_3	j_4
1	1	−4	1
2	1	1	1
3	−2	1	−6
4	−2	1	−6
5	1	1	−6
6	1	1	−6
7	1	1	1
8	1	−4	−6
9	−2	−4	−6
10	−2	1	−6
−5	−2	−4	−5
−3	−2	−3	−3

S'	j_3
1	1
2	1
3	1
4	1
5	−3
−4	−4

T'	j_4
1	−5
2	−5
3	1
4	1
5	1
−6	−6
−3	−3

$p_3 = b$, $p_4 = c$. We now change the join types and define the join conditions:

$$j_2 : Q \xleftarrow{p_2} R$$
$$j_3 : S \xleftrightarrow{p_3} R$$
$$j_4 : T \xrightarrow{p_4} R.$$

Consider the join $\boldsymbol{J}_3$ given by the operator tree $\boldsymbol{T}_3$ shown in Fig. 6.

The tree $\boldsymbol{T}_3$ can be simplified by an elegant and efficient procedure based on *null rejection* [1,5]. However, we first leave $\boldsymbol{T}_3$, hence $\boldsymbol{J}_3$, unchanged. We also use the same preserve numbers:

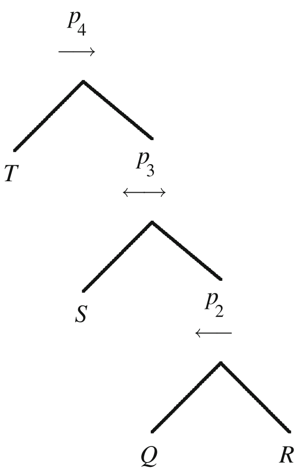


Fig. 6 Operator tree for Example 2

$$\pi(R, j_2) = -2$$
$$\pi(S, j_3) = -3$$
$$\pi(R, j_3) = -4$$
$$\pi(T, j_4) = -5.$$

At this point, we depart from the method as described above. We ignore all original relational content and create the derived relations. We represent the original content symbolically by one artificial row with identifier 1 and attribute values set to their preserve numbers. Next, we proceed immediately to virtual row generation as shown in Table 5.

Using the relations in Table 5 we can now define the derived predicate p'_4. For rows $r' \in R'$, $t' \in T'$ with row identifiers ρ and τ, and any rows $r \in R$, $t \in T$ corresponding to real r' and t', predicate p'_4 is given by

$$p'_4(r', t') = (\rho > 0 \wedge \tau > 0 \wedge c(r, t)) \vee$$
$$(t'.j_4 = -5 = \rho) \vee$$
$$(\rho < 0 \wedge \tau < 0 \wedge \rho = \tau).$$

Table 5 Derived relations for Example 2

Q'
1
−2
−3
−5

R'	j_2	j_3
1	−2	−4
−3	−3	−3
−5	−5	−5

S'	j_3
1	−3
−4	−4
−5	−5

T'	j_4
1	−5

The row $r' \in R'$ with $\rho = -3$, under p'_4, does not match the only row of T', so r' can be deleted from R', as can the row with $\kappa = -3$ from Q'. Now the symbolic row $s' \in S'$ with $\sigma = 1$ matches no row in R': We can update $s'.j_3$ to 1. Now the preserve number $\pi(S, j_3) = -3$ is no longer used, and j_3 can be simplified to a right outer join. Hence, $\boldsymbol{J}_3$ is equivalent to the join $\boldsymbol{J}'_3$ shown in Fig. 7.

This example shows that the virtual row method permits us to make the same simplification as by using null rejection [5] simply by checking matches and deleting dangling rows.

2.5 Properties

We now return to the virtual row method in general and review some of its properties. One remarkable fact is that the number of virtual rows is always quite low. For the worst case we derive an upper bound that is independent of the number of real rows.

Property. The maximum number of virtual rows generated for an outer join query on $n \in \mathbf{N}^+$ relations is exactly

$$\frac{1}{2}(n-1)(n+2).$$

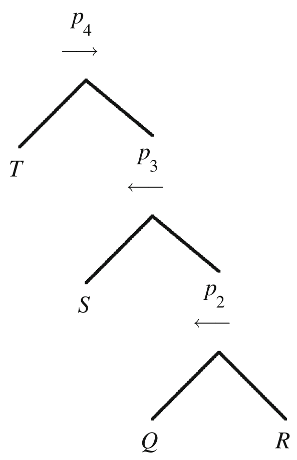
Proof Let a join query on $n \in \mathbf{N}^+$ relations be given by its operator tree $\boldsymbol{T}$ and let $j_2, j_3, \ldots, j_n$ be the ordered sequence of its join conditions. For $2 \le k \le n$, let r_k be the number of relations in the subtree of $\boldsymbol{T}$ rooted at j_k; we clearly have $r_k \le k$. For a fixed k, let R and S be the two relations referenced by j_k, and let $r_{k,R} \in \mathbf{N}^+$ be the number of relations in the subtree of $\boldsymbol{T}$ rooted at j_k containing R, similarly for $r_{k,S}$. We have $r_{k,R} + r_{k,S} = r_k$. Processing j_k requires at most the two calls *generateVirtualRows*(R, j_k) and *generateVirtualRows*(S, j_k). Since the former call generates at most $r_{k,S}$ virtual rows and the latter at most $r_{k,R}$, for j_k at most $r_k \le k$ new virtual rows are generated. Summation gives the upper bound

$$\sum_{k=2}^{n} k = \frac{1}{2}(n-1)(n+2).$$

This upper bound is reached by the full outer join $R \leftrightarrow S_i$, for $i = 1, \ldots, n-1$. □

The *tuple form* of a join tuple consists exclusively of 0 and 1 values, and is obtained from the join tuple by replacing each row identifier > 1 by 1 and each row identifier < 0 by 0. All join tuples of an inner join have the same tuple form $(1, 1, \ldots, 1)$. Unlike the number of virtual rows, the number of tuple forms of an outer join on n relations may be exponential in n, both in the theoretical worst and in the practical average cases. In the worst case, the full outer join

$$R \leftrightarrow S_i, i = 1, \ldots, n$$



$\overset{p_4}{\longrightarrow}$
T
$\overset{p_3}{\longleftarrow}$
S
$\overset{p_2}{\longleftarrow}$
Q
R

Fig. 7 Simplified operator tree for Example 2

on $n+1$ relations has exactly $2^n + n$ tuple forms, consisting of 2^n tuple forms with all combinations of 0 and 1 in which $r = 1$, together with the n tuple forms

$$(r = 0, 0, \ldots, 0, s_i = 1, 0, \ldots, 0).$$

So, in any case a small number of virtual rows suffices to control the assembly of a possibly large number of tuple forms.

For the practical average case, consider Tables 6 and 7, which contain the results of examining a large number of randomly chosen join queries whose graphs are trees. The random choices were made as follows. Start with a tree of one node. Connect each new node to an existing node, chosen

Table 6 Average and maximum numbers of tuple forms

Number of relations	Average number of tuple forms	Maximum number of tuple forms
5	5.32	20
10	17.3	223
15	57.7	1 860
20	208	10 100
25	766	45 900
30	2 820	121 000

Table 7 Average and maximum numbers of virtual rows per relation

Number of relations	Average number of virtual rows per relation	Maximum number of virtual rows per relation
5	1.32	4
10	2.18	9
15	2.85	12
20	3.41	15
25	3.94	17
30	4.44	20
40	5.32	23
50	6.16	25
60	6.91	27
70	7.69	33
80	8.37	33
90	9.02	36
100	9.71	37

with equal probability, and assign the edge a join type. When the graph is complete, assign the edges an order of execution. Finally, simplify the join operators [1,5].

Each row of Tables 6 and 7 represents at least 10,000 cases of the indicated size. Here we assume that all columns $R'.j$ contain their preserve numbers $\pi(R, j)$ in a real row.

3 Semijoin reduction of outer joins

In this section, we extend the well known technique of semijoin reduction of inner joins [2] to apply also to outer joins. This is one way to perform step 4 in the virtual row method (Sect. 2.3.4). We chose it for implementation in our join engine because it is particularly suited to process joins in a distributed landscape.

As described in Sect. 2, the relations to be fully reduced are the derived relations $R'_1, R'_2, \ldots, R'_n$. However, to reduce the clutter of primes, we describe semijoin reduction of outer joins as applied directly to the original relations $R_1, R_2, \ldots, R_n$. We can do this if we simply append the new columns $R.j$ to R instead of collecting them into a derived relation R'. Since step 4 is performed when all relations still contain only real rows, the original and derived predicates agree.

For semijoin reduction of outer joins, the execution order of join conditions is irrelevant, so we simplify our treatment by representing any joins by their join graphs alone.

We begin by generalizing semijoin reduction steps. We introduce *preserving* moves and define the *influencing* relation between a vertex and an incidence of the join graph. This captures the effect of iterating reduction steps. We introduce the *materialization set* and the *materialization tree*; the latter is the subtree of $\boldsymbol{JG}$ that needs to be fully reduced. We then define the *needed* relation for any vertex $R \in \boldsymbol{JG}$, where *needed*(R) is the set of vertexes needed to reduce R. Next, we define *full reduction* of vertexes. We define *reduction plans* as sequences of semijoin reduction steps fully reducing the materialization tree. The extension of semijoin reduction to outer joins is simply to execute a reduction plan.

As the running example for this section, we use a join whose join graph is shown in Fig. 8.

3.1 Semijoin reduction steps

In semijoin reduction of inner joins, a semijoin reduction step consists in replacing some relation R by one of its possible semijoins $R \ltimes S$, where R, S are neighbored in the join graph $\boldsymbol{JG}$. This is called the *reduction* of R by S or a *reduction move* from S to R.

For the extension, let j be a join condition referencing the original relations R, S. If j does not preserve R, the semijoin reduction step from S to R is called *reducing* and has the same effect as above: It deletes from R any row that matches no row in S. If j preserves R, the reduction step is called *preserving*. It updates any row $r \in R$ that matches no row in S by setting $r.j$ to $\pi(R, j)$.

In the running example, looking at join condition i, a reduction move from R to S is reducing, a move from S to R is preserving.

$$\begin{tikzcd} & & S & & \\ P \arrow[r, "g"] & Q & R \arrow[l, "h"'] \arrow[u, "i"'] & T \arrow[l, "j"'] \arrow[r, leftrightarrow, "k"] & U \end{tikzcd}$$

Fig. 8 Example of a join for outer join reduction

3.2 Influencing

Next, we define the *influencing* relation on $\boldsymbol{JG}$ by specifying whether a vertex S influences an incidence (R, j). Roughly, this is the case when the path from S to (R, j) does not run against an arrowhead.

Influencing captures the effect of iterated semijoin reduction. Let $R, S \in \boldsymbol{JG}$ be any two vertexes. Informally, S influences an incidence (R, j) iff reduction of R by S via intermediate reduction steps can change the column $R.j$, by updates or deletes.

Formally, let $e \in \boldsymbol{JG}$ be an edge joining the two vertexes U, T. Let $\boldsymbol{I}(U, e)$ be the set of all incidences on the U side of e and $\boldsymbol{V}(T, e)$ the set of all vertexes on the T side of e:

$$\boldsymbol{I}(U,e) = \{(R,j) \in \boldsymbol{JG} : \text{ the incidence path from } U \text{to } (R,j) \text{ does not include } (U,e)\}$$
$$\boldsymbol{V}(T,e) = \{S \in \boldsymbol{JG} : \text{ the node path from } T \text{to } S \text{ does not include } U\}.$$

In the general case we have these properties:

$$(U,e), (T,e) \notin \boldsymbol{I}(U,e)$$
$$\boldsymbol{I}(U,e) = \emptyset \text{ iff } U \text{ is a leaf in } \boldsymbol{JG}$$
$$T \in \boldsymbol{V}(T,e)$$
$$U \notin \boldsymbol{V}(T,e).$$

We now say that $S \in \boldsymbol{JG}$ influences $(R, j) \in \boldsymbol{JG}$ iff there is no edge $e \in \boldsymbol{JG}$, referencing relations $U, T \in \boldsymbol{JG}$ and preserving U, such that $S \in \boldsymbol{V}(T, e)$ and $(R, j) \in \boldsymbol{I}(U, e)$.

For a given incidence (R, j), we denote the set of all vertexes influencing (R, j) by *infl*(R, j):

$$\mathit{infl}(R,j) = \{S \in \boldsymbol{JG} : S \textit{ influences } (R,j)\}.$$

If j joins the two relations R, S, then at least

$$R, S \in \mathit{infl}(R,j).$$

Table 8 Influencing relation for the example in Fig. 8

Influences	P	Q	R	S	T	U
(P, g)	+	+	+	i	+	k
(Q, g)	+	+	+	i	+	k
(Q, h)	+	+	+	i	+	k
(R, h)	+	+	+	i	+	k
(R, i)	h	h	+	+	+	k
(R, j)	h	h	+	i	+	k
(S, i)	h	h	+	+	+	k
(T, j)	h	h	+	i	+	k
(T, k)	j, h	j, h	j	i, j	+	+
(U, k)	j, h	j, h	j	i, j	+	+

For an inner join the influencing relation is always trivial:

$$\mathit{infl}(R, j) = \boldsymbol{JG}.$$

For our running example, the influencing relation is given by the matrix in Table 8. A plus sign in a cell indicates that the relation of the column influences the incidence of the row. Instead of entering a minus sign in the remaining cells, we list all edges (that is, join conditions) preventing this influence.

One can read the *infl* sets immediately from the rows:

$$infl\ (P, g) = \{P, Q, R, T\}$$
$$\dots$$
$$infl(U, k) = \{T, U\}.$$

3.3 Materialization tree

We define the materialization set $\boldsymbol{M} \subseteq \boldsymbol{JG}$ as a certain connected set of incidences. Its *materialization tree* $\boldsymbol{F}$ is the set of all vertexes underlying $\boldsymbol{M}$ and is always a subtree of $\boldsymbol{JG}$. It is the set of all vertexes that have to be fully reduced.

If we had considered the transformation of an outer join query to an equivalent inner join query as a standalone problem where every tuple of the join must be computed, we could only make the trivial choices

$$\boldsymbol{M} = \boldsymbol{JG} = \boldsymbol{F}.$$

But if we assume that the query result is required to be free of duplicates, we can do much better. We set

$$\boldsymbol{M} = \langle \boldsymbol{A} \rangle \subseteq \boldsymbol{JG}$$

where $\boldsymbol{A} \subseteq \boldsymbol{JG}$ is some set of incidences always comprising the set $\boldsymbol{A}_0$, to be defined below, with $\boldsymbol{A} = \boldsymbol{A}_0$ being the simplest and also the most common case, and $\langle \boldsymbol{A} \rangle$ is the connected set of incidences spanned by $\boldsymbol{A}$.

To construct $\boldsymbol{A}_0$, for every attribute $R.A$ referenced in the SELECT clause of the query, include in $\boldsymbol{A}_0$ all incidences (R, j) such that either j references $R.A$ or the reduction move to R via j reduces R. Note that in general even $\boldsymbol{A} = \emptyset = \boldsymbol{M}$ is possible. To continue our running example, we choose

$$\boldsymbol{A} = \{(R, h), (T, k)\}$$
$$\boldsymbol{M} = \langle \boldsymbol{A} \rangle = \{(R, h), (R, j), (T, j), (T, k)\}$$
$$\boldsymbol{F} = \{R, T\}.$$

3.4 Needing

To fully reduce a given vertex, we need to reduce it by certain other vertexes. Let us call this set of vertexes the *needed* set. We initially define it for $R \in \boldsymbol{F}$:

$$\mathit{needed}(R) = \bigcup \{infl(R, j) : (R, j) \in \boldsymbol{M}\}.$$

In the inner join case, this relation is trivial:

$$\mathit{needed}(R) = \boldsymbol{JG} \text{ if } R \in \boldsymbol{F}.$$

We now extend the definition to all of $\boldsymbol{JG}$: For $R \in \boldsymbol{JG}$ let $N \in \boldsymbol{F}$ be the unique vertex nearest to R among the vertexes in $\boldsymbol{F}$; this is well defined since $\boldsymbol{F}$ is a tree. Then

$$\mathit{needed}(R) = \{T \in \mathit{needed}(N) : \text{ the path from } T \text{ to } N \text{ passes through } R\} \cup \{R\}.$$

In the case that $R \in \boldsymbol{F}$ we have $N = R$ and the two definitions coincide. In our example, on $\boldsymbol{F}$ we initially have

$$\mathit{needed}(R) = \mathit{infl}(R, h) \cup \mathit{infl}(R, j) = \{P, Q, R, T\}$$
$$\mathit{needed}(T) = \mathit{infl}(T, j) \cup \mathit{infl}(T, k) = \{R, T, U\}.$$

Note that $U \notin \mathit{needed}(R)$ because U does not influence (R, j). And S influences (R, i) but $S \notin \mathit{needed}(R)$ because $(R, i) \notin \boldsymbol{M}$. When we extend this to all of $\boldsymbol{JG}$ we get

$$\mathit{needed}(P) = \{P\}$$
$$\mathit{needed}(Q) = \{P, Q\}$$
$$\mathit{needed}(S) = \{S\}$$
$$\mathit{needed}(U) = \{U\}.$$

3.5 Reduction plans

Having found the *needed* sets, we can proceed to define reduction plans. Executing a sequence of semijoin reduction steps, we associate to each node $R \in \boldsymbol{JG}$ its *reducer set*

$$\mathit{red}(R).$$

This is the set of relations by which R has been reduced so far. Initially,

$$red(R) = \{R\}.$$

Every reduction move from some node $S \in \boldsymbol{JG}$ to a neighbor $R \in \boldsymbol{JG}$ via edge j changes exactly one reducer set, namely $red(R)$, by updating

$$red(R) = red(R) \cup (red(S) \cap infl(R, j)).$$

Call a vertex $R \in \boldsymbol{JG}$ *fully reduced* iff

$$red(R) \supseteq needed(R).$$

An arbitrary set of vertexes is fully reduced iff every element is. A sequence of semijoin reduction steps is a *reduction plan* iff it fully reduces the materialization tree $\boldsymbol{F}$.

Let us see this in our running example. If $X \Rightarrow Y$ stands for using X to reduce Y, we claim that the move sequence

$$P \Rightarrow Q, Q \Rightarrow R, R \Rightarrow T, U \Rightarrow T, T \Rightarrow R$$

is a reduction plan. Initially,

$$red(P) = \{P\}, red(Q) = \{Q\}, red(R) = \{R\},$$
$$red(S) = \{S\}, red(T) = \{T\}, red(U) = \{U\}$$

and $P, S, U \in \boldsymbol{JG}$ are fully reduced right from the start. Next, move 1, $P \Rightarrow Q$, updates

$$red(Q) = \{P, Q\}.$$

This move fully reduces Q. Move 2, $Q \Rightarrow R$, updates

$$red(R) = \{P, Q, R\}.$$

Move 3, $R \Rightarrow T$, updates

$$red(T) = \{R, T\}.$$

Move 4, $U \Rightarrow T$, again updates

$$red(T) = \{R, T, U\}.$$

Now T is fully reduced. Finally, move 5, $T \Rightarrow R$, again updates

$$red(R) = \{P, Q, R, T\}.$$

Now R is fully reduced, and so is $\boldsymbol{F}$, establishing our claim.

4 Conclusion

In our SAP NetWeaver engine implementation, we next had to implement an optimizer finding reasonably good outer join reduction plans. However, since this is a complicated topic and only distracts from the generality and simplicity of the basic method, we do not describe this optimizer here. Regarding inner join optimization, see [2,4,6,7].

In short, we have developed an efficient method for reducing outer joins to inner joins. Because we have implemented the method successfully in a commercial engine, we trust that the method is not only of theoretical but also of practical interest.

Finally, we thank the anonymous reviewers for their comments on an earlier draft of this article. The paper has been materially improved as a result.

References

1. Bhargava, G., Goel, P., Iyer, B.: Simplification of outer joins. In: Proceedings Conference of the Centre for Advanced Studies on Collaborative Research, Toronto (1995)
2. Bernstein, P.A., Goodman, N., Wong, E., Reeve, C.L., Rothnie, J.B. Jr.: Query processing in a system for distributed databases (SDD-1). ACM Trans. Database Syst. **6**(4), 602–625 (1981)
3. Chiu, D.M., Ho, Y.C.: A methodology for interpreting tree queries into optimal semi-join expression. In: Proceedings ACM-SIGMOD International Conference on Management of Data, Santa Monica, CA, pp. 169–178. ACM, New York (1980)
4. Galindo-Legaria, C.A., Rosenthal, A.: How to extend a conventional optimizer to handle one- and two-sided outerjoin. In: Proceedings 8th International Conference on Data Engineering, Tempe, AZ, pp. 402–409 (1992)
5. Galindo-Legaria, C.A., Rosenthal, A.: Outerjoin simplification and reordering for query optimization. ACM Trans. Database Syst. **22**(1), 43–74 (1997)
6. Kambayashi, Y., Yoshikawa, M., Yajima, S.: Query processing for distributed databases using generalized semi-joins. In: Proceedings ACM-SIGMOD International Conference on Management of Data, Orlando, FL, pp. 151–160. ACM, New York (1982)
7. Stocker, K., Braumandl, R., Kemper, A., Kossmann, D.: Integrating semi-join-reducers into state-of-the-art query processors. In: Proceedings 17th International Conference on Data Engineering (ICDE'01), Heidelberg, pp. 575–584 (2001)