Outerjoins as disjunctions

C.A. Galindo-Legaria

Computer Science/Department of Algorithmics and Architecture

CS-R9404 1994

Outerjoins as Disjunctions

César A. Galindo-Legaria
CWI
P. O. Box 4079, 1009 AB Amsterdam, The Netherlands
cesar@cwi.nl

Abstract

The outerjoin operator is currently available in the query language of several major DBMSs, and it is included in the proposed SQL2 standard draft. However, "associativity problems" of the operator have been pointed out since its introduction. In this paper we propose a shift in the intuition behind outerjoin: Instead of computing the join while also *preserving* its arguments, outerjoin delivers tuples that come *either* from the join *or* from the arguments. Queries with joins and outerjoins deliver tuples that come from one out of several joins, where a single relation is a trivial join. An advantage of this view is that, in contrast to preservation, *disjunction* is commutative and associative, which is a significant property for intuition, formalisms, and generation of execution plans.

Based on a disjunctive normal form, we show that some data merging queries cannot be evaluated by means of binary outerjoins, and give alternative procedures to evaluate those queries. We also explore several evaluation strategies for outerjoin queries, including the use of semijoin programs to reduce base relations.

CR Subject Classification (1991): [H.2.4] Database Systems, Query Processing; [H.3.3] Information Search and Retrieval, Query Formulation; [G.2.2] Graph Theory; [F.2.2] Nonnumerical algorithms and problems.
Keywords and Phrases: Outerjoin, Query Graph, Query Formulation and Processing.
Note: The author had an ERCIM postdoctoral fellowship while conducting this work.

1 Introduction

1.1 Motivation

Relational query languages and optimizers are designed to exploit the properties of Select/Project/Join (S/P/J) queries. In particular, they depend on the fact that join queries can be unambiguously specified using a *query graph*. The graph does not impose a partial ordering on the operations, but instead shows relations as nodes, and join predicates as edges. Since execution order is inessential to the query's semantics, languages and optimization of S/P/J queries are relatively simple: Users need only provide enough information to produce the query graph, e. g., the SQL Select-From-Where clauses. And every operator tree that "implements" the query graph is known to evaluate to the same result.

But there are applications of matching elements of two relations that are not expressed properly by means of join. A problem arises if we want to preserve all elements of one (or both) of the relations in the result, even if there is no matching element in the other relation. As an example, in a listing of customers and their purchase orders, we often want to see *all* customers, even those without purchase orders. To obtain this rather natural result in standard SQL, we have to compute the union of two SQL query blocks, one of which has a nested block.

The *outerjoin* operator is a modification of join that adds to the result the non-matching tuples from one or both of the relations being combined. *One-sided* or *left* outerjoins preserve only one of its arguments. They are used basically to flatten conceptual hierarchies, where some of the "parents" have no "children" [Dav91], as in the above example of customers and purchase orders. *Full* outerjoins preserve both of its arguments. They are used basically to merge data from different sources [DH84, SL90], e. g., a list of information about clients in database1 and database2, but a single row with all information for each client appearing in both databases.

An outerjoin construct is included in the SQL2 standard draft [ANS92], and proprietary or SQL2-compatible extensions for outerjoins are currently available in Sybase, NonStop SQL of Tandem, SSQL of ShareBase, and ORACLE/SQL [Dav91, HM92]. The operator is also used to represent algebraically some loop constructs of Daplex [Shi81], and to flatten nested SQL queries with Count aggregates [Day87, Mur89].

Unfortunately, it is known that outerjoins are neither commutative nor associative, in general. For this reason, a query graph in which edges indicate an operator (join or outerjoin), in addition to match predicates, is not an unambiguous representation of a Outerjoin/Join (O/J) query. Sometimes the same query graph can be derived from two queries that compute different results.

For join, the intuition of pairs of matching tuples is easily extended from a single operator to queries composed of several joins. And then, from the query graph formalism, it is easy to generate execution plans for the query. We claim that these "query-wide" intuition and formalisms provide an extremely important insight, both in the formulation and evaluation of queries, that goes beyond what can be grasped from a collection of algebraic identities. For example, it would be difficult to study semijoin reduction programs based only on algebraic identities for binary join and semijoin.

For O/J queries, what is the intuition? What is the appropriate formalism? What are the feasible execution plans?

In this paper we propose a shift in the intuition behind outerjoin: Instead of computing the join while also *preserving* its arguments, outerjoin delivers tuples that come *either* from the join *or* from the arguments. O/J queries deliver tuples that come from one out of several joins (a single relation is a trivial join). An advantage of this view is that, in contrast to preservation, *disjunction* is commutative and associative, which is a significant property for intuition, formalisms, and generation of execution plans.

1.2 Basic Definitions

We adhere to the usual concepts and notation of relational algebra and relational calculus.

A *scheme* is a finite set of attribute names. A *tuple t on scheme S* is an assignment of values to attribute names in S. The scheme of tuple t is denoted sch(t). For $X \subseteq$ sch(t), we say t is a tuple *over* X. A *null tuple* on scheme S has a null value assigned for all attributes, and is denoted null_S.

Tuples t_1, t_2 on schemes S_1, S_2, respectively, can be *concatenated* if the schemes are disjoint or the assignments coincides for all attributes in $S_1 \cap S_2$ in both tuples. The concatenation is a tuple on $S_1 \cup S_2$, whose assignment coincides with that of t_1 for attributes in S_1, and with that of t_2 for attributes in S_2. Concatenation of tuples t_1, t_2 is denoted (t_1, t_2). If t is a tuple on scheme S, we may obtain a tuple t' on scheme $S' \supseteq S$ by *padding*, i. e. concatenating t with $\text{null}_{S'-S}$.

A *relation* on scheme S is a finite set of tuples on scheme S. The scheme of relation R is denoted $\text{sch}(R)$. A *database* is a set of relations whose schemes are mutually disjoint; they will be called *base relations.*

A *tuple predicate* p is defined on some set of attributes S; it maps tuples over S to $\{True, False\}$. Such S is called the *scheme* of the predicate, and denoted $\text{sch}(p)$. For any tuple t over $S = \text{sch}(p)$, $p(t)$ is defined and depends only on the values of attributes S in t. Given a database, a predicate p is said to be n-ary if $\text{sch}(p)$ intersects the scheme of n base relations of the database. Note that our definition of predicate covers arbitrary, user-defined predicates.

We call a predicate p *strong* with respect to a set S of attributes if, whenever a tuple t has a null value for all attributes in S, $p(t)$=False [RGL90]. Comparisons (and boolean combinations of comparisons) in the where clause of SQL queries actually behave as strong predicates, on each of the attributes they reference. If an attribute being compared turns out to be null for a given tuple, the value of the predicate is "unknown," and the tuple is not selected [NPS91]. In this paper we assume that predicates are strong on each attribute they reference.

Let R_1, R_2 be relations with schemes S_1, S_2, respectively. The *outerunion*, denoted $R_1 \uplus R_2$, first pads the tuples of each relation to scheme $(S_1 \cup S_2)$, and then computes the union of the resulting sets [Cod79]. Outerunion has lower precedence than join —i. e. in an expression without parenthesis, joins should be evaluated first, and then outerunions.

Let R_1, R_2 be relations with disjoint schemes S_1, S_2, respectively, and predicate p, such that $\text{sch}(p) \subseteq (\text{sch}(R_1) \cup \text{sch}(R_2))$. The *join* is $R_1 \overset{p}{\bowtie} R_2 := \{(t_1, t_2) \mid t_1 \in R_1, t_2 \in R_2, p(t_1, t_2)\}$. The *antijoin* is $R_1 \overset{p}{\triangleright} R_2 := \{t_1 \mid t_1 \in R_1, (\neg\exists t_2 \in R_2)(p(t_1, t_2))\}$. The *left outerjoin* is $R_1 \overset{p}{\rightarrow} R_2 := R_1 \overset{p}{\bowtie} R_2 \uplus R_1 \overset{p}{\triangleright} R_2$. The *right outerjoin* is $R_1 \overset{p}{\leftarrow} R_2 := R_2 \overset{p}{\rightarrow} R_1$. The *full outerjoin* is $R_1 \overset{p}{\leftrightarrow} R_2 := R_1 \overset{p}{\bowtie} R_2 \uplus R_1 \overset{p}{\triangleright} R_2 \uplus R_2 \overset{p}{\triangleright} R_1$.

Since left and right outerjoin are symmetric cases, we use the term "left outerjoin" in a generic sense, as an outerjoin that preserves all tuples from one of its arguments only. Also, we use the general term "outerjoin" to refer both to left or full outerjoin. The symbol $\odot$ is used to denote join or outerjoin in expressions of the form $R_1 \overset{p}{\odot} R_2$.

A *query* is an expression in relational algebra. We usually view it as an operator tree, i. e. a tree whose leaves correspond to relation variables, and whose internal nodes contain joins, outerjoins, and other algebraic operators. The result of a query Q is denoted *eval*(Q), and is defined by the usual bottom-up evaluation of expressions. If $op_1, op_2, \ldots$ denote generic operators (e.g., selection, outerjoin), then an $op_1/op_2/\ldots$ query is a an expression in which only the mentioned operators may appear (e.g., Outerjoin / Join query).

In this paper we assume a *conjunctive normal form* for predicates of join and outerjoin, where each conjunct is a tuple predicate in the sense defined above. We also make a *connectivity assumption*, namely, for any subtree $(Q_1 \overset{p_1 \wedge \cdots \wedge p_n}{\odot} Q_2)$ of a join/outerjoin query, each p_i references relations both in Q_1 and in Q_2.

Given a join query, its *query graph* has a node for each relation mentioned in the query, and an edge for each conjunct of the join predicates. An edge labeled p exists between the nodes of two relations, say R_i, R_j, if p references attributes of R_i, R_j; more exactly, $\mathrm{sch}(p) \not\subseteq \mathrm{sch}(R_i)$, $\mathrm{sch}(p) \not\subseteq \mathrm{sch}(R_2)$, but $\mathrm{sch}(p) \subseteq (\mathrm{sch}(R_i) \cup \mathrm{sch}(R_j))$. If query conjuncts refer to more than two relations, then the resulting query is a hyper-graph [Ull89]. We do not consider hyper-graphs explicitly, but, in principle, our results extend to them as well.

For an outerjoin/join query, we construct the graph using the same procedure, ignoring whether conjuncts appear in join or outerjoin predicates. Obviously, the resulting query graph does not provide enough information to compute the query. The graph corresponding to query Q is denoted *graph*(Q).

2 A normal form for outerjoins

2.1 Join-disjunctive queries

We introduce the idea of join-disjunctive queries with an example.

Example 1 Suppose we have a database with relations CUSTOMERS, ORDERS, and ITEMS, with the obvious meaning, and predicates p_1 to match customers to orders, and p_2 to match orders to items. Assuming ITEMS′ has our luxury items, we want a query to list all our customers, with their orders, and their luxury items. Tuples in the result must have the following format:

CUSTOMERS	ORDERS	ITEMS′
c	o	i
c	o	$-$
c	$-$	$-$

The first row corresponds to customers and orders for luxury items; the second row corresponds to customers and orders for non-luxury items; and the third row corresponds to clients who have not ordered anything. The result contains all tuples in the join (CUSTOMERS $\overset{p_1}{\bowtie}$ ORDERS $\overset{p_2}{\bowtie}$ ITEMS′), all tuples in the join (CUSTOMERS $\overset{p_1}{\bowtie}$ ORDERS), and also all tuples in CUSTOMERS.

Each of the three "sub-results" can be retrieved by projecting on the appropriate attributes. These "sub-results" are included in the result in a "minimal" way. For example, if $(c, -, -)$ appears in the result, it means that $c \in$ CUSTOMER, but there is no $o \in$ ORDERS such that $(c, o) \in$ (CUSTOMERS $\overset{p_1}{\bowtie}$ ORDERS). ■

The query in the previous example is called *join-disjunctive*, because, intuitively, it consists of the *minimum union* of several join queries on a given query graph. A formal definition of minimum union is given next[1].

[1]This in not the only approach to formalize the operator we need. However, it is simpler than an alternative approach to minimum union given in [GL92], which is based on projection and set difference.

Definition. We say that a tuple t_1 *subsumes* t_2 if they are defined on the same attributes, t_2 has more null values than t_1, and t_1 coincides with t_2 in all non-null attributes of t_2. The *removal of subsumed tuples* of R, denoted $R\downarrow$, returns the tuples of R that are not subsumed by any other tuple in R [Ull89].

Definition. The *minimum union* of relations R_1, R_2 is $R_1 \oplus R_2 := (R_1 \uplus R_2)\downarrow$. Minimum union has lower precedence than join.

Observation 1. Minimum union is commutative and associative. In addition, if R_1 does not contain subsumed tuples, then $(R_1 \oplus \emptyset) = R_1$; and if $R_2 \subseteq R_1$ then $(R_1 \oplus R_2) = R_1$.

Now, based on minimum union, we define *join-disjunctive queries.*

Definition. Let $G = (V, E)$ be a query graph. The *join evaluation* of a query graph is $\bowtie(G) = \sigma_{p_1 \wedge \cdots \wedge p_n}(R_1 \times \cdots \times R_m)$, where $\{p_1, \ldots, p_n\}$ are the labels of edges E and $\{R_1, \ldots, R_n\}$ are the labels of nodes V.

Definition. Let $G = (V, E)$ be a graph and $V' \subseteq V$. The *induced subgraph*, denoted $G|_{V'}$, is (V', E'), where $E' = \{(u, v) \mid (u, v) \in E, u \in V', v \in V'\}$. When the query graph G is understood, we write the join evaluation of a subgraph $\bowtie(G|_{V'})$ simply as $\bowtie(V')$.

Definition. Let $G = (V, E)$ be a query graph. Let $V_1, \ldots, V_n \subseteq V$ be sets of nodes such that each induced subgraph $G|_{V_1}, \ldots, G|_{V_n}$ is connected. The query $\bowtie(V_1) \oplus \cdots \oplus \bowtie(V_n)$ is called a *join-disjunctive query*, or a *join-disjunction*, on G. Each $\bowtie(V_i)$ is called a *term* of the query.

Each term of a join-disjunctive query corresponds to a join, and the query result contains the results of these various joins. Although not required, join-disjunctions on G often contain a term for the join evaluation of the whole graph $\bowtie(G)$.

Since both minimum union and joins are commutative and associative, join-disjunctive queries can be specified in a *set notation.* Given a query graph G, we write query $\bowtie(V_1) \oplus \cdots \oplus \bowtie(V_n)$ as $\{V_1, \ldots, V_n\}$. A term $\bowtie(V_i)$, where $V_i = \{R_{i_1}, \ldots R_{i_n}\}$ is written simply as V_i or, using juxtaposition of its elements, as $R_{i_1} \cdots R_{i_n}$.

Observation 2. The set notation of a join-disjunctive query on graph G denotes a unique result.

Observation 3. If normal disjunctive queries Q_1, Q_2 have different set notations, then there are database instances for which eval$(Q_1) \neq$ eval(Q_2).

Example 2 For the query of example 1, the query graph has three relations, and two edges, corresponding to predicates p_1, p_2. The set notation for the join-disjunctive query is {CUSTOMERS ORDERS ITEMS$'$, CUSTOMERS ORDER, CUSTOMERS}. ■

2.2 Rewriting outerjoins as disjunctions

The following identities are used to rewrite join/outerjoin queries:

$$R_1 \xrightarrow{p} R_2 = R_1 \overset{p}{\bowtie} R_2 \oplus R_1; \text{ if } R_1 = R_1\downarrow, R_2 = R_2\downarrow . \quad (1)$$

$$R_1 \overset{p}{\leftrightarrow} R_2 = R_1 \overset{p}{\bowtie} R_2 \oplus R_1 \oplus R_2; \text{ if } R_1 = R_1\downarrow, R_2 = R_2\downarrow . \quad (2)$$

$$(R_1 \oplus R_2) \overset{p}{\bowtie} R_3 = R_1 \overset{p}{\bowtie} R_3 \oplus R_2 \overset{p}{\bowtie} R_3; \text{ if } R_3 = R_3\downarrow . \quad (3)$$

Identities 1 and 2 above are straightforward, but distributivity of join over minimal union, identity 3, requires closer analysis. It is based on the distributivity of cartesian product and selection over minimal union, but there is a problem with selection, because it does not always commute with removal of subsumed tuples. For instance, assume $t_1 \in R\downarrow$, $t_2 \in R$, t_1 is the only tuple that subsumes t_2, p is false in t_1, but p is true in t_2. Then $t_2 \notin \sigma_p(R\downarrow)$, but $t_2 \in (\sigma_p R)\downarrow$. However, this problem does not appear under our stated assumption that predicates are strong on each attribute they reference.

We shall assume that base relations do not contain subsumed tuples —which is usually the case in practice, given that tuples have some user- or system-assigned unique identifier. Then, by identities 1–3, intermediate results in join/outerjoin queries do not contain subsumed tuples. The following example illustrates how to rewrite a query as a join-disjunction, using identities 1–3.

Example 3 The query $(R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3$ is rewritten as a join-disjunction as follows:

$$\begin{aligned}
&(R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 \\
&= (R_1 \overset{p^{12}}{\bowtie} R_2 \oplus R_1 \oplus R_2) \overset{p^{23}}{\rightarrow} R_3 \\
&= (R_1 \overset{p^{12}}{\bowtie} R_2 \oplus R_1 \oplus R_2) \overset{p^{23}}{\bowtie} R_3 \oplus R_1 \overset{p^{12}}{\bowtie} R_2 \oplus R_1 \oplus R_2 \\
&= R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3 \oplus R_1 \overset{p^{23}}{\bowtie} R_3 \oplus R_2 \overset{p^{23}}{\bowtie} R_3 \oplus R_1 \overset{p^{12}}{\bowtie} R_2 \oplus R_1 \oplus R_2 \\
&= R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3 \oplus R_2 \overset{p^{23}}{\bowtie} R_3 \oplus R_1 \overset{p^{12}}{\bowtie} R_2 \oplus R_1 \oplus R_2 \\
&= \{R_1R_2R_3, R_2R_3, R_1R_2, R_1, R_2\}.
\end{aligned}$$

■

2.3 A normal form

It turns out that join-disjunctions provide a normal form for join/outerjoin queries. We show that next.

Observation 4. Let G be a query graph partitioned into two connected subgraphs G_1, G_2. Let p be the conjunction of labels of all edges between G_1 and G_2 in G. Let $Q_1 = \{V_{11}, \ldots, V_{1n}\}, Q_2 = \{V_{21}, \ldots, V_{2m}\}$ be join-disjunctive queries on graphs G_1, G_2 respectively. Then the expressions $Q_1 \overset{p}{\bowtie} Q_2$, $Q_1 \overset{p}{\rightarrow} Q_2$, and $Q_1 \overset{p}{\leftrightarrow} Q_2$ can all be rewritten as join-disjunctive queries.

Proof. For $Q_1 \overset{p}{\bowtie} Q_2$, repeated application of identity 3 rewrites the expression as $(\bowtie(V_{11})) \overset{p}{\bowtie} (\bowtie(V_{21})) \oplus (\bowtie(V_{11})) \overset{p}{\bowtie} (\bowtie(V_{22})) \oplus \cdots \oplus (\bowtie(V_{1n})) \overset{p}{\bowtie} (\bowtie(V_{2m}))$. For each join $(\bowtie(V_{1i})) \overset{p}{\bowtie} (\bowtie(V_{2j}))$, p contains conjuncts for all conditions between relations in V_{1i} and V_{2j}, and perhaps more. If p contains a conjunct involving some relation not in V_{1i}, V_{2j}, then the result of $(\bowtie(V_{1i})) \overset{p}{\bowtie} (\bowtie(V_{2j}))$ is actually the empty relation, by predicate strongness, and it is removed; otherwise, $(\bowtie(V_{1i})) \overset{p}{\bowtie} (\bowtie(V_{2j}))$ is the term $\bowtie(V_{1i} \cup V_{2j})$ of G.

The outerjoin cases, $Q_1 \overset{p}{\rightarrow} Q_2$ and $Q_1 \overset{p}{\leftrightarrow} Q_2$, follow directly from identities 1, 2, and the join case. ■

Theorem 1. *Let Q be a join/outerjoin query. Q can be rewritten as a join-disjunctive query Q'. Such Q' is a normal form for Q.*

Proof. First, consider rewriting operator tree Q as a join-disjunction. We show by induction the stronger property that any subtree of Q can be rewritten as a join-disjunction. Each leaf of Q is a single relation, which is a trivial join-disjunction. Now, let $Q' = (Q_1 \overset{p}{\odot} Q_2)$ be a subtree of Q. By the connectivity assumption, $G' = \text{graph}(Q')$ is partitioned by connected subgraphs $G_1 = \text{graph}(Q_1), G_2 = \text{graph}(Q_2)$, and p is the conjunction of the labels of all edges between G_1 and G_2 in G. By induction hypothesis, Q_1, Q_2 can be rewritten as join-disjunctive queries. By observation 4, Q' can be written as a join-disjunction.
By observation 2, if rewriting two queries yields the same join-disjunction, those queries evaluate to the same result. By observation 3, if rewriting two queries yields different join-disjunctions, those queries evaluate to different results, in general. ∎

An immediate consequence of theorem 1 is a simple, transparent procedure to decide whether or not two join/outerjoin queries are equivalent —rewrite them as join-disjunctive queries and see if they are equal. This is done in the following example to prove identities.

Example 4 In example 1 we transformed query $(R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3$ into the join-disjunctive query $\{R_1R_2R_3, R_2R_3, R_1R_2, R_1, R_2\}$. Query $R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\rightarrow} R_3)$ can also be rewritten as join disjunction $\{R_1R_2R_3, R_2R_3, R_1R_2, R_1, R_2\}$. Therefore the associative identity $(R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\rightarrow} R_3)$ holds.

On the other hand $R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) = \{R_1R_2R_3, R_2R_3, R_1\}$, but $(R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\bowtie} R_3 = \{R_1R_2, R_1, R_2\} \overset{p^{23}}{\bowtie} R_3 = \{R_1R_2R_3, R_2R_3\}$. Therefore the associative identity $R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\bowtie} R_3$ does *not* hold. ∎

A list of identities for joins, left outerjoins, and full outerjoins is given in [GLR93]. An initial motivation for our join-disjunctions was to facilitate the proof of identities with outerjoins, which is somewhat involved if we transform left- into right-hand-side expressions algebraically —see [RGL90, OMO89], for example.

Our disjunctive normal form is in the worst case exponential on the number of relations of the query. This is to be expected, since it encodes operator trees of non-associative operators. Still, we believe it is an intuitive formalism, appropriate for reasoning about outerjoin queries.

3 Cyclic query graphs

3.1 Full disjunction and data merging

Example 4 showed that some associative identities involving joins and outerjoins do not hold. However, [GLR93] shows that the order of evaluation in many important cases can be changed using simplification techniques, and a *generalized outerjoin* operator —which we describe later in this paper. In this section we turn our attention to a different sort of "associativity problem" raised in the literature. The problem affects the intended answer of the query, rather than query evaluation.

Example 5 Recall the database of example 1. We have relations CUSTOMERS, ORDERS, and ITEMS; predicate p_1 to matches customers to orders, and p_2 to match orders to items. We add a new predicate p_3 matching each customer with items produced in the same location as his or her address.

Let CUSTOMERS$'$ be customers within certain age range, and ITEMS$'$ be luxury items. Our query is the full outerjoin of three relations under p_1, p_2, p_3, but note that:

$$\begin{aligned}
&(\text{CUSTOMERS}' \overset{p_1}{\leftrightarrow} \text{ORDERS}) \overset{p_2 \wedge p_3}{\leftrightarrow} \text{ITEMS}' \\
&= (\text{CUSTOMERS}' \overset{p_1}{\bowtie} \text{ORDERS}) \overset{p_2 \wedge p_3}{\bowtie} \text{ITEMS}' \oplus \text{CUSTOMERS}' \overset{p_1}{\bowtie} \text{ORDERS} \oplus \\
&\quad \text{CUSTOMERS}' \overset{p_2 \wedge p_3}{\bowtie} \text{ITEMS}' \oplus \text{ORDERS} \overset{p_2 \wedge p_3}{\bowtie} \text{ITEMS}' \oplus \\
&\quad \text{CUSTOMERS}' \oplus \text{ORDERS} \oplus \text{ITEMS}' \\
&= (\text{CUSTOMERS}' \overset{p_1}{\bowtie} \text{ORDERS}) \overset{p_2 \wedge p_3}{\bowtie} \text{ITEMS}' \oplus \text{CUSTOMERS}' \overset{p_1}{\bowtie} \text{ORDERS} \oplus \\
&\quad \text{CUSTOMERS}' \oplus \text{ORDERS} \oplus \text{ITEMS}' \\
&= \{\text{CUSTOMERS}'\ \text{ORDERS ITEMS}',\ \text{CUSTOMERS}'\ \text{ORDERS}, \\
&\quad \text{CUSTOMERS}',\ \text{ORDERS},\ \text{ITEMS}'\}.
\end{aligned}$$

The result does not lose tuples from the original relations, but it does miss some matches. Since ORDERS ITEMS$'$ is not a term, orders will not be matched with their items —unless they were ordered by a customer in CUSTOMERS$'$, in which case they are part of term ORDERS ITEMS$'$ CUSTOMERS$'$.

A second observation is that different associations of the query, namely ((ORDERS $\overset{p_2}{\leftrightarrow}$ ITEMS$'$) $\overset{p_3 \wedge p_1}{\leftrightarrow}$ CUSTOMERS$'$) and ((ITEMS$'$ $\overset{p_3}{\leftrightarrow}$ CUSTOMERS$'$) $\overset{p_1 \wedge p_2}{\leftrightarrow}$ ORDERS), all yield different results, but none contains all terms. Some tuple matches are still missing from the result. ■

Definition. The *full disjunction* of a query graph G is given by the join-disjunctive query $Q = \{V' \mid G|_{V'} \text{ is connected}\}$.

A full disjunction delivers all tuples of the base relations, combining them as a single tuple whenever they match. Clearly, on fully connected graphs, the full disjunction has exponentially many terms; but, in our view, full disjunctions capture the requirements of data merging queries.

The following theorem relates full disjunction with full outerjoins.

Theorem 2. *Assume the operators in query Q are all full outerjoins, and let G = graph(Q). Q computes the full disjunction of G if and only if G is acyclic.*

Proof. (Sketch) The if part is proved by induction on the subtrees that make up the operator tree Q. To prove the only-if part, the argument follows the join-disjunction rewrite of example 5: There is an outerjoin operator in Q that, when applied to the join-disjunction of its subtrees, will remove some terms.

An important consequence of our theorem is that, for acyclic query graphs, the full disjunction can be computed applying full outerjoins, in any order. For cyclic graphs, however, the full preservation —i. e. data merging query— *cannot* be computed by means of binary outerjoins.

The "outerjoin associativity" problem pointed out in [Mai83] is an instance of (natural) full outerjoins on a cyclic graph.

3.2 Implied graph edges

A related "associativity problem" affecting the intended answer of a query is the following. Conjuncts of join predicates may logically imply other conjuncts, e. g., using transitivity of equality. In terms of query graphs, implied conjuncts add edges, and thus modify the topology. If graph G_2 is obtained by adding or removing implied edges from G_1, then the join evaluation of both graphs yield the same result. But the full disjunction —i. e. data merging query— is *not*, in general, the same on both graphs. The following example, taken from [Dat86a], shows the problem.

Example 6 We want to compute the full outerjoin of the the following relations S, P, J, comparing their *city* attributes. Let predicate p^{sp} be $(S.city{=}P.city)$, p^{pj} be $(P.city{=}J.city)$, and p^{sj} be $(S.city{=}J.city)$. Outerjoin queries yield the following results.

S

$S\#$	*city*
S1	London
S2	Paris

P

$P\#$	*city*
P1	Paris
P2	Oslo

J

$J\#$	*city*
J1	Oslo
J2	London

$Q_1 = (S \overset{p^{sp}}{\leftrightarrow} P) \overset{p^{sj}}{\leftrightarrow} J$

$S\#$	$S.city$	$P\#$	$P.city$	$J\#$	$J.city$
S1	London	-	-	J2	London
S1	Paris	P1	Paris	-	-
-	-	P2	Oslo	-	-
-	-	-	-	J1	Oslo

$Q_2 = (S \overset{p^{sp}}{\leftrightarrow} P) \overset{p^{pj}}{\leftrightarrow} J$

$S\#$	$S.city$	$P\#$	$P.city$	$J\#$	$J.city$
S1	London	-	-	-	-
S1	Paris	P1	Paris	-	-
-	-	P2	Oslo	J1	Oslo
-	-	-	-	J2	London

Although the join evaluation of graphs $G_1 = \text{graph}(Q_1)$ and $G_2 = \text{graph}(Q_2)$ is the same, the full disjunctions of G_1 and G_2 on the above tables —computed by queries Q_1, Q_2— are not the same. ■

Definition. Given a query graph $G = (V, E)$, the *implied graph* $G' = (V, E')$ contains the maximum set of edges E' such that each edge $e' \in E'$ is logically implied by the predicates in E.

Observation 5. Let G' be the implied graph of G. The join evaluations $\bowtie(G')$ and $\bowtie(G)$ are the same. If the graphs G' and G are different, then G' has cycles.

Observation 6. If G is different from its implied graph G', then the full disjunction of G' cannot be computed by any outerjoin query.

Example 7 For the example 6 above, the query graph $G_1 = \text{graph}(Q_1)$ contains only two edges, p^{sp}, p^{sj}, but the implied graph G' of G_1 contains edges p^{sp}, p^{sj}, p^{pj} —and G' is also the implied graph of $G_2 = \text{graph}(Q_2)$. The full disjunction of G', which cannot be computed by full outerjoins only, is the following:

$$Q_2 = (S \overset{p^{sp}}{\leftrightarrow} P) \overset{p^{pj}}{\leftrightarrow} J$$

$S\#$	$S.city$	$P\#$	$P.city$	$J\#$	$J.city$
S1	London	-	-	J2	London
S1	Paris	P1	Paris	-	-
-	-	P2	Oslo	J1	Oslo

■

Date uses example 6 above to show that full outerjoin is not associative, in general[2], but asserts that natural full outerjoin is associative [Dat86a]. Maier gives an example of natural full outerjoin that does not associate in [Mai83]. Our analysis shows that cycles in the query graph are the source of the problem, and also identifies the relevance of implied edges, which are relatively unimportant on join queries.

4 Evaluation of join-disjunctions

4.1 Semijoin reduction programs

Semijoin reduction programs have been proposed in the evaluation of join queries in distributed databases [CP85]. The idea is to eliminate irrelevant tuples from base relations, before the actual query is evaluated. This reduction can be viewed as an optional preprocessing step, because, whether or not the base relations are reduced, it is still necessary to select a join evaluation order for the query. The preprocessing step pays off when its cost is lower than the cost reduction it brings to the evaluation of the actual query.

In general, for a given join/outerjoin (or simply join) query Q, the tuples of R_i needed to answer Q are given by $\pi_{\text{sch}(R_i)}Q$. When Q includes outerjoins, sometimes $R_i = \pi_{\text{sch}(R_i)}Q$, so all tuples of R_i are relevant to the result and no reduction is possible on that relation. The next example shows that some base relations in join/outerjoin queries do contain irrelevant tuples.

Example 8 In example 3, we saw that query $Q = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3$ is rewritten as the join-disjunction $\{R_1R_2R_3, R_2R_3, R_1R_2, R_1, R_2\}$. Since terms R_1 and R_2 are included, all the tuples of those relations are relevant to the result. But relation R_3 appears only in terms $R_1R_2R_3$ and R_2R_3. Furthermore, any tuple of R_3 in $\bowtie(R_1R_2R_3)$ is also in $\bowtie(R_2R_3)$. Therefore, we conclude that tuples of R_3 that do not match a tuple in R_2 are irrelevant to the result of Q. ■

Definition. Let V_1, V_2 be terms on graph G. Viewing terms as sets of relations, set containment induces a transitive relation on terms, that corresponds to a directed acyclic graph. We say V_1 is an *acestor* of V_2, denoted $V_2 \leq V_1$, if relations in V_2 are contained in V_1; if, in addition, $V_1 \neq V_2$, then V_1 is a *parent* of V_2, denoted $V_2 < V_1$. Given a collection of terms $\{V_1, \ldots V_n\}$, if $V_i < V_j$ and there is not V_k such that $V_i < V_k < V_j$, we say V_j is a *parent* of V_i. *Descendant* and *child* are the inverses of *ancestor* and *parent*, respectively.

[2]Actually, he deals with outer *equi*-join.

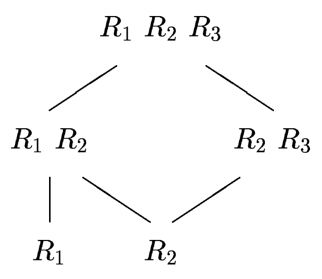

Figure 1: Parent-child relationship of terms.

Figure 1 shows the parent-child relationship of the terms of the query of example 3.

Observation 7. Let V_1, V_2 be terms of graph G, and S be the scheme of $\bowtie(V_1)$. If $V_1 \leq V_2$ then $\pi_S(\bowtie(V_2)) \subseteq \bowtie(V_1)$.

Observation 8. Relational projection distributes over minimum union.

Let Q be a join-disjunctive query on graph G. Let R_i be a relation used in Q, with scheme S_i. To reduce R_i, compute $R_i' = \pi_{S_i} Q$ as follows.

- Set $Q' = \{V \mid V \in Q, R_i \in V\}$. From observations 1 and 8, $\pi_{S_i} Q = \pi_{S_i} Q'$.
- Set $Q'' = \{V \mid V \in Q',\ V$ has no child in $Q'\}$. From observations 1, 7, and 8, $\pi_{S_i} Q' = \pi_{S_i} Q''$.
- Let $Q'' = \{V_1, \ldots, V_m\}$. Compute $R_{ij} = \pi_{S_i}(\bowtie(V_j))$, for $j = 1, \ldots, m$. Each R_{ij} can be computed by a semijoin program if $G|_{V_j}$ is acyclic [CP85]. The reduced relation R_i' is the union of relations $R_{i1}, \ldots, R_{im}$.

The above procedure can be used to find a reduction plan for every relation in the query. Since different the reduction plans may contain common work, it is important to schedule semijoins so that work is not duplicated. We do not consider that problem here.

4.2 Basic outerjoin algorithms

Whether or not base relations are reduced, we still have to evaluate outerjoins efficiently. To compute a single outerjoin $R_1 \xrightarrow{p} R_2$ we could follow the algebraic definition, and take the union of the join $R_1 \overset{p}{\bowtie} R_2$ and the antijoin $R_1 \overset{p}{\triangleright} R_2$. But it seems reasonable to expect common work in the computation of joins and antijoins so, instead of computing them separately, it has been suggested to modify join algorithms to compute outerjoin directly [Day87]. We summarize next two basic implementation ideas that have been proposed [RR84, GL92, PMC93], and then extend them to join-disjunctions.

"Simultaneous" computation of join and antijoin. Since there are no common tuples in the join $R_1 \bowtie R_2$ and null-padded antijoin $R_1 \triangleright R_2$, computing their union reduces to merging two streams of tuples. Some join algorithms are easy to modify so that, conceptually, they return two independent output streams, one with the result of the join, and the other with the result of the outerjoin. Then, merging these two streams yields the outerjoin.

In particular, many join algorithms distinguish an *outer* and an *inner* relation, and for each tuple in the outer relation they find all matching tuples of the inner relation —e. g. nested-loops, hash-join, index-join. Simple modifications to these algorithms can determine the tuples of the outer relation without a matching tuple in the inner, thus making possible to output efficiently the required streams, namely join and antijoin [RR84]. The limitation of this procedure is that, in general, it cannot compute the antijoin of the inner relation, only the outer.

Similarly, the sort-merge algorithm for joins can be modified so that it efficiently outputs both antijoins, in addition to the join [GL92]. These three output streams allow easy computation of full outerjoin. Unfortunately, the sort-merge algorithm cannot be used to compute joins with arbitrary predicates.

Mark and re-scan. Simultaneous computation of join $R_1 \bowtie R_2$ and semijoin $R_1 \rtimes R_2$ is easy, if duplicates in the semijoin result are irrelevant —simply channel to the semijoin output stream every tuple that is successfully matched with another. Conceptually, this semijoin result is used only to mark tuples of an input relation, say R_1. After marking is complete, and additional scan of R_1 finds the result of the antijoin $R_1 \triangleright R_2$ as those tuples that remain unmarked. Adding this antijoin to the previously computed join yields the outerjoin result. Tuple marking could be implemented, for example, using a hash table of tuple ids. This is the basis of the general outerjoin algorithm proposed in [PMC93][3]. As pointed out in that same reference, this second procedure is less efficient than the first procedure outlined above, due to the extra scans and temporary memory required, but it can compute arbitrary outerjoins.

4.3 Independent computation of terms

The strategy of computing outerjoin by following the definition —i. e. taking the union of independently computed join and antijoin— is generalized next to join-disjunctions.

Definition. Let Q be a join-disjunctive query whose result has scheme S, and V be a term of Q. Assume Q' is the join-disjunctive query containing all terms of Q, except V. Let R' be the result of eval(Q'), padded to scheme S. The *contribution* of V to Q is defined as eval$(Q) - R'$.

Observation 11. Let $Q = \{V_1, \ldots, V_n\}$ be a join-disjunctive query, and let S_i be the scheme of the result of $\bowtie(V_i)$. Assume $V_{i_1}, \ldots, V_{i_m}$ are the parents of V_i in Q. If $m = 0$, then the contribution (up to padding) of V_i to Q is $\bowtie(V_i)$; otherwise, the contribution (up to padding) of V_i to Q is $((\bowtie(V_i)) - \pi_{S_i}(\bowtie(V_{i_1}))) \cap \cdots \cap ((\bowtie(V_i)) - \pi_{S_i}(\bowtie(V_{i_m})))$. Each of the relational differences to be intersected corresponds to an antijoin.

The resulting intersections can also be written as antijoin sequences, as it is easy to show that $(R_1 \overset{p_1}{\triangleright} R_2 \cap R_1 \overset{p_2}{\triangleright} R_3) = (R_1 \overset{p_1}{\triangleright} R_2) \overset{p_2}{\triangleright} R_3 = (R_1 \overset{p_2}{\triangleright} R_3) \overset{p_1}{\triangleright} R_2$.

Theorem 3. *Let $Q = \{V_1, \ldots, V_n\}$ be a join-disjunctive query. Assume $C_1, \ldots, C_n$ are the respective contributions of $V_1, \ldots, V_n$ to Q. C_i is disjoint with C_j, for all $i \neq j$. Q is equal to the union $C_1 \cup \cdots \cup C_n$.*

[3]That reference also examines specific cases where the antijoin of the inner relation can be efficiently computed simultaneously with the join.

Proof. (Sketch) Assume, without loss of generality, that terms are arranged so that $V_i < V_j$ implies $j < i$. Define the collection of join-disjunctive queries $Q_k = \{V_i \mid V_i \in Q, i \leq k\}$ for $1 \leq k \leq n$. It can be shown, by induction on k, that Q_k is equal to the union of the disjoint contributions of its terms, for $1 \leq k \leq$. Since $Q_k = Q$, the theorem follows.

Example 9 Query $Q = (R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 = \{R_1R_2R_3, R_2R_3, R_1R_2, R_1, R_2\}$ of example 3 can be computed using contributions as

$$R_1 \overset{p^{12}}{\bowtie} R_2 \overset{p^{23}}{\bowtie} R_3 \uplus (R_2 \overset{p^{23}}{\bowtie} R_3) \overset{p^{12}}{\triangleright} R_1 \uplus (R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\triangleright} R_3 \uplus$$
$$R_1 \overset{p^{12}}{\triangleright} R_2 \uplus (R_2 \overset{p^{12}}{\triangleright} R_1) \overset{p^{23}}{\triangleright} R_3.$$

■

Similar to single outerjoin evaluation, the above strategy may duplicate work. It does provide a reference evaluation procedure for arbitrary disjunctions, based on joins, antijoins, and union of disjoint sets, rather than elimination of subsumed tuples.

4.4 Extending the basic outerjoin evaluation techniques

Strategies to compute outerjoins can be extended to compute join-disjunctions. In general, *generalized* semijoin and antijoin have to be computed simultaneously with join.

Definition. Let R_1, R_2 be relations with disjoint schemes, p be a predicate between R_1, R_2, and $S \subseteq (\mathrm{sch}(R_1) \cup \mathrm{sch}(R_2))$ a set of attributes. The *generalized semijoin* is the S-projection of the join, i. e. $R_1 \overset{S,p}{\ltimes} R_2 := \pi_S(R_1 \overset{p}{\bowtie} R_2)$. If, in addition, S is disjoint with $\mathrm{sch}(R_2)$, the *generalized antijoin* is the S-projection of R_1 that does not appear in the join, i. e. $R_1 \overset{S,p}{\triangleright} R_2 := \pi_S R_1 - (R_1 \overset{S,p}{\ltimes} R_2)$.

Assuming operators that compute join and *semi*join simultaneously, join-disjunctions are handled as follows. The contribution of each term is collected independently at the end, as the difference of tuples received from a "positive" stream minus those received from one of more "negative" (or "mark") streams. For a given term V, the "positive" stream comes from the computation of $\bowtie(V)$, and the "negative" streams come from semijoins obtained by operators that compute $\bowtie(V')$, for each parent V' of V.

Example 10 Figure 2 shows a way to compute the join-disjunctive query of example 3, $Q = \{R_1R_2R_3, R_2R_3, R_1R_2, R_1, R_2\}$ on predicates p^{12}, p^{23}. The flow of tuples is from left to right. Operators compute join and semijoin simultaneously; the join is sent through streams starting on a solid circle (●), and the semijoin is sent through streams starting on a hollow circle (○).

Figure 1 shows the child-parent relationship for the query. The contribution of R_1 is computed based on a "positive" stream coming from R_1, and a "negative" stream coming from a semijoin of operator $\bowtie_1$. The contribution of R_2 requires two "negative" streams, one from operator $\bowtie_1$, and another from operator $\bowtie_2$. Finally, operator $\bowtie_4$ computes join $\bowtie(R_1R_2R_3)$, and also generalized semijoins $\pi_{\mathrm{sch}(R_1R_2)}(\bowtie(R_1R_2R_3))$, $\pi_{\mathrm{sch}(R_2R_3)}(\bowtie(R_1R_2R_3))$.
■

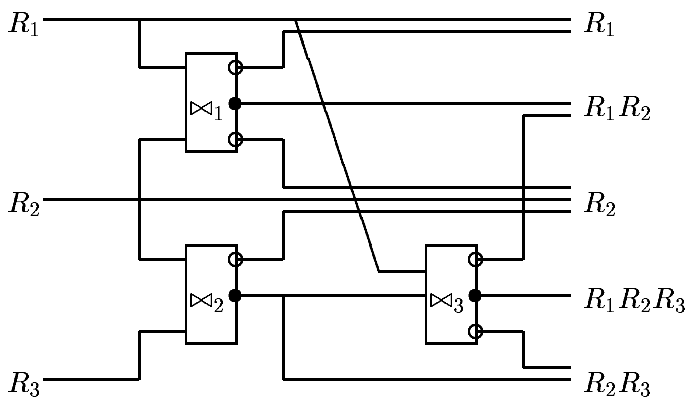

Figure 2: Computing with semijoin streams.

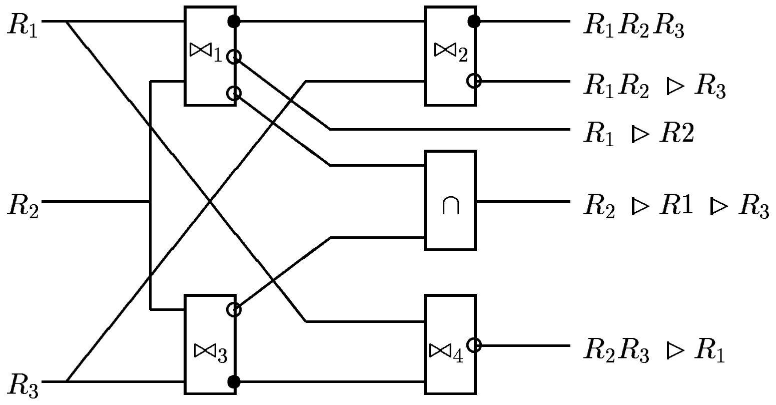

Figure 3: Computing with antijoin streams.

As with the single outerjoin case, each of the operators is relatively simple, but there is some memory and time overhead involved in marking and re-scanning —i. e. computing the difference of "positive" and "negative" streams above.

Assuming operators that compute join and *anti*join simultaneously, join-disjunctions are handled as follows. When given term V with child V' is evaluated, the antijoin $((\bowtie (V)) - (\bowtie (V')))$ is simultaneously computed, and antijoins results are intersected at the end to determine term contributions.

Example 11 Figure 3 shows a way to compute the join-disjunctive query of example 3, $Q = \{R_1R_2R_3, R_2R_3, R_1R_2, R_1, R_2\}$ on predicates p^{12}, p^{23}. Operators compute join and antijoin simultaneously; the join is sent through streams starting on a solid circle ($\bullet$), and the antijoin is sent through streams starting on a hollow circle ($\circ$).

Figure 1 shows the child-parent relationship for the query. The contribution of R_1 is computed as an atijoin in $\bowtie_1$. The contribution of R_2 is the intersection of two antijoins, one from $\bowtie_1$ and another from $\bowtie_3$. The antijoin on attributes of R_2R_3 cannot be computed by $\bowtie_2$, because those attributes intersect with the scheme of both input streams. For this reason, we are forced to add another operator, $\bowtie_4$, to compute the contribution of R_2R_3. ■

The antijoin-based strategy seems to impose less restrictions on the synchronization of operators, compared to the semijoin-based strategy that requires relational differences at the end. On the other hand, to implement an operator that computes join and antijoin simultaneously, we need a semijoin-based strategy, in the general case.

5 Conclusions and comparison with related work

Related work. After the definition given in [LP76], a number of papers have used outerjoins to formulate certain classes of queries, e. g. [DH84, Dat86b, Day87, GW87, Mur89, OMO89, WM90, Dav91]. But a key problem with outerjoins is their lack of associativity [Dat86b, Dat86a, Mai83], which makes more difficult the appropriate formulation of queries, as well as their evaluation. Papers dealing with evaluation of queries containing outerjoins include [RR84, Che90, RGL90, GLR92, GLR93, PMC93].

The current paper follows the trend of [RGL90, GLR92, GLR93] in that it studies queries that can be represented without an explicit order of evaluation, but, in contrast, those papers focus on query reordering and evaluation based on binary operator trees. Our focus here is an order-independent representation that can be used to prove identities and show equivalence easily (see section 2), study expressiveness (see section 3), and explore evaluation strategies other than binary operator trees (see section 4), while also removing the restriction of binary predicates on outerjoins found in [GLR93].

In terms of approach, [RGL90, GLR92, GLR93] could be called *bottom-up*, in the sense that they start with algebraic identities and then consider evaluation orders allowed by those identities. This paper uses a *top-down* approach, in the sense that it starts by describing which tuples must be in the result, and then considering how to obtain them.

Contributions. The core of this paper is the proposal of join-disjunctive queries as a formal, intuitive view of queries containing joins and outerjoins. Those queries form a natural and important class, as they correspond closely to SQL2 query blocks.

The number of terms in a join-disjunction is, in the worst case, exponential on the number of relations of the query, which may limit their direct use in query languages. Nevertheless, we believe they provide the appropriate primitives to *reason* both about specification and evaluation of queries containing outerjoins.

In terms of specification, we revisited the outerjoin associativity problems described in [Dat86b, Dat86a, Mai83], and showed that they arise exactly in the case of cyclic query graphs. In addition, we showed that the natural data-merging query on a cyclic graph cannot be computed using binary outerjoins only.

In terms of evaluation, we outlined the use of semijoin reduction programs, as well as other evaluation strategies based on operators with multiple output streams. These strategies seem more suitable for parallel or distributed systems, but the principles could also be used in a centralized system.

Semijoin reductions may prove to be particularly useful, because they are likely to eliminate irrelevant tuples *early*. Then, evaluation of the query on the reduced relations may be less sensitive to join ordering, as all the tuples that remain after the reduction do appear in the result. In addition, distributed databases may already implement semijoins.

Acknowledgements. I am grateful to Martin Kersten, Arjan Pellenkoft, Arnie Rosenthal, and Arno Siebes for encouragement and comments on earlier versions of this paper. They helped improve both the contents and presentation of this work.

References

[ANS92] ANSI. Working draft of sql2/sql3. Technical report, American National Standards Institute, 1992.

[Che90] A. L. P. Chen. Outerjoin optimization in multidatabase systems. In *2nd. Intl. Symposium on Databases in Parallel and Distributed Systems*, 1990.

[Cod79] E. F. Codd. Extending the relational database model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[CP85] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, New York, 1985.

[Dat86a] C. J. Date. *An Introduction to Database Systems*, volume II. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Dat86b] C. J. Date. *Relational Database Selected Writings.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Dav91] M. David. Advanced capabilities of the outer join. *ACM SIGMOD Record*, 21(1):65–70, March 1991.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton*, pages 197–208, 1987.

[DH84] U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering*, November 1984.

[GL92] C. Galindo-Legaria. *Algebraic Optimization of Outerjoin Queries.* PhD thesis, Harvard University, 1992. Technical report TR-12-92.

[GLR92] C. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *Proceedings of the Eighth International Conference on Data Engineering*, 1992.

[GLR93] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization, 1993. Submitted for publication.

[GW87] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queries revisited. In *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco*, pages 23–33, 1987.

[HM92] T. Hartley and T. Martyn. *ORACLE/SQL: A Professional Programmer's Guide.* McGraw-Hill, New York, 1992.

[LP76] M. Lacroix and A. Pirotte. Generalized joins. *ACM SIGMOD Record*, 8(3), September 1976.

[Mai83] D. Maier. *The theory of relational databases.* Computer Science Press, Rockville, MD, 1983.

[Mur89] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of the Fifteenth International Conference on Very Large Databases, Amsterdam*, pages 77–85, 1989.

[NPS91] M. Negri, G. Pelagatti, and L. Sbatella. Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 17(3):513–534, September 1991.

[OMO89] G. Ozsoyoglu, V. Matos, and Z. M. Ozsoyoglu. Query processing techniques in the summary-table-by-example database query language. *ACM Transactions on Database Systems*, 14(4):526–573, December 1989.

[PMC93] H. Pirahesh, C. Mohan, and J. Cheng. Sequential and parallel algorithms for unified execution of outer join and subqueries. Technical report, IBM Almaden Reseach Center, San Jose, June 1993.

[RGL90] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, New Jersey*, 1990.

[RR84] A. Rosenthal and D. Reiner. Extending the algebraic framework of query processing to handle outerjoins. In *Proceedings of the Tenth International Conference on Very Large Databases, Singapore*, 1984.

[Shi81] W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[SL90] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-base Systems*, volume II. Computer Science Press, Rockville, MD, 1989.

[WM90] Y. R. Wang and S. E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane*, pages 519–538, 1990.