

iAnywhere Solutions Technical White Paper



Semantics and compatibility of Transact-SQL outer joins

G. N. Paulley

15 February 2002

Document Number 1017447

Abstract

Transact-SQL outer joins are specified using the special comparison operators ‘* =’ and ‘= *’. Using one (or more) of these operators in a query specifies an outer join between two tables in the `From` clause, though the semantics of such queries can be unclear. This is because TSQL outer join semantics were never formally defined, and in older releases of Adaptive Server Enterprise and Adaptive Server Anywhere the results of TSQL outer join queries could depend upon the access plan chosen by the optimizer. This iAnywhere Solutions’ Technical White Paper describes the semantics of TSQL outer joins in detail, and in particular outlines the differences in support for TSQL outer joins between ASE and ASA. Because of these differences, iAnywhere Solutions recommends that customers use ANSI outer join syntax, rather than Transact-SQL outer join syntax, in their applications.

Contents

1 Preliminaries	4
2 Simple TSQL outer joins	6
3 TSQL outer joins and more complex search conditions	8
4 Allowable forms of TSQL outer join expressions	11
5 TSQL outer joins and nested queries	12
6 Disjunctive TSQL outer join conditions	14
7 Star outer joins	14
8 Chained (nested) outer joins	16
9 Summary	17
A Automatically converting statements to ANSI syntax	17
References	18

1 Preliminaries

Left and right outer joins are two very useful relational operators whose characteristics are similar, yet quite different, to inner joins.

EXAMPLE 1

Consider the simple left outer join query

```
Select *  
From R Left Outer Join S On ( C )
```

where C denotes a search condition. The result of this left outer join can be explained as follows:

1. For each row of R , find a row from S that satisfies condition C .
2. If at least one such row from S is found, output each combination of this row from R and each matching row of S .
3. If no row in S exists such that C evaluates to *true*, output a result row comprising the attributes from R and substitute NULL values for each (missing) attribute of S .

Table R is termed the *preserved table*, since every row from R will appear in the result. Similarly, S is termed the *null-supplying table*, since NULL values are substituted for a row of S when no row from S can cause condition C to be satisfied. Note that left outer joins are trivially equivalent to right outer joins simply by reversing their operands.

The straightforward example above uses ANSI-defined join syntax. Transact-SQL (or TSQL) outer joins, on the other hand, are specified using the $* = \text{or} = *$ comparison operators in the query's **Where** clause, which imply a left or right outer join, respectively, between two quantifiers (base tables or views) in the query's **From** clause. Transact-SQL outer joins were originally implemented in a very early release of Sybase SQL Server, as it was then known, and before the semantics of outer joins were fully understood from a theoretical standpoint.

There are four specific problems with Transact-SQL outer join notation:

1. The syntax does not distinguish between predicates in the **On** condition and those in the **Where** clause (using ANSI terminology), and hence the semantics can be unclear to a user.
2. The outer join condition must be an equality condition; any other comparison operator is not permitted.
3. One is unable to specify nested outer joins without referring to views (though this is now supported as of ASE 11.5, it is still not supported by Adaptive Server Anywhere - see Section 8 below).

4. Full outer joins cannot be specified using Transact-SQL syntax.

In addition to these syntactic problems, confusion about TSQL outer join semantics is also due to two factors:

- In ASE releases before 12.0, and ASA releases before 7.0, a query containing TSQL outer joins could return a different result depending on the optimization strategy chosen for the query.
- Implementation changes in various ASE and ASA releases have both extended and restricted the class of valid TSQL outer join queries that can be executed.

This technical white paper summarizes Transact-SQL outer join behavior with respect to various ASE and ASA releases, using specific examples. It explains their semantics by converting TSQL outer joins into their ANSI-equivalent syntax using `Left Outer` or `Right Outer Join` with explicit `On` conditions. Such rewriting is precisely what both ASE 12 and ASA 7 and above now do with TSQL outer joins. This rewriting is performed prior to optimization, so that query results are now unaffected by the access plan chosen by the optimizer. Semantic differences in the behavior of TSQL outer joins still exist between the two products, however, in how specific classes of queries are converted.

To (roughly) summarize ANSI semantics as defined by the ANSI/ISO SQL-99 standard, the semantics of a query specification containing the following clauses (some optional)

```
Select [ Distinct ]
From
[ Where ]
[ Group By ]
[ Having ]
```

are:

1. Compute the extended Cartesian product of each independent table expression in the `From` clause. Note that with ANSI join syntax, ANSI outer joins appear in the `From` clause and hence their evaluation is part of this step.
2. Restrict the result of (1) such that each derived row satisfies the search condition specified by the `Where` clause (that is, the condition evaluates to *true*). Any *null-intolerant* predicate¹ that refers to attributes from a

¹ A null-intolerant predicate is a predicate that cannot evaluate to *true* if any of its inputs are `Null`. Most SQL predicates, such as comparisons, `Like`, or `In` predicates are null-intolerant. Examples of null-tolerant predicates are `Is Null` and any predicate *p* qualified by a null-tolerant truth value test, such as `p Is Not True`.

null-supplying table will eliminate `Null`-supplied rows from the result. In this case, the query is equivalent to a rewritten query using an inner join.

3. If `Group By` is specified, partition the result of (2) into groups of rows, with each group defined as a unique combination of values from the expressions listed in the `Group By` clause. If no `Group By` clause is specified, then the result of (2) is treated as a single ‘group’. Aggregate functions in the query’s `Select` or `Having` clauses can now be evaluated for each group.
4. If a `Having` clause is present, restrict the result of (3) such that the attribute values of each group must satisfy the search condition specified in the `Having` clause.
5. Project the result over the attributes listed in the query’s `Select` list.
6. If `Distinct` is specified, eliminate duplicate result rows.

2 Simple TSQL outer joins

The most significant issue with Transact-SQL outer joins is deciding what happens to additional `Where` clause predicates that refer to the null-supplying table. Whether or not these predicates are treated as part of an outer join’s `On` condition, or part of the query’s `Where` clause, depends on several factors. This discussion applies only to predicates that occur in the same block (query or view): semantically, Transact-SQL treats any view as a materialized table.

In the original implementation of TSQL outer joins, whether or not a predicate in the `Where` clause was treated as part of the outer join’s `On` condition was dependent on the placement of the predicate in the query’s access plan, as determined by the optimizer. This was true in ASE releases prior to 12.0, and ASA releases prior to 7.0. In general, this approach did not guarantee consistent results. In practice, however, for most queries consistency would be maintained because of two characteristics of both query optimizers:

1. preserved tables must precede any null-supplying tables in any valid access plan; and
2. predicates are evaluated as soon as possible; that is, predicates are placed in the plan at the earliest point that they can be evaluated, and are not deferred (say, after another join).

To illustrate the semantics of simple TSQL outer joins queries, we consider examples using the following schema:

```

Create Table T ( a Int Primary Key, b Int, c Int );
Create Table R ( x Int Primary Key, y Int, z Int );
Create Table S ( l Int Primary Key, m Int, n Int );
Create Table W ( d Int Primary Key, e Int, f Int );

Insert Into T Values( 1,2,3 );
Insert Into T Values( 2,4,5 );
Insert Into T Values( 3,4,5 );
Insert Into R Values( 3,4,5 );
Insert Into S Values( 3,0,0 );
Insert Into W Values( 3,4,5 );

```

EXAMPLE 2

Consider the query

```

Select *
From R, S
Where R.x * = S.l and S.m > 5.

```

This typical example of a TSQL outer join query is semantically equivalent to the ANSI formulation

```

Select *
From R Left Outer Join S On( R.x = S.l and S.m > 5 ).

```

Note the lack of a **Where** clause in this rewritten query; the additional condition on $S.m$ has been made part of the outer join's **On** condition. Any additional conjunctive conditions² on R would be considered to be part of the **Where** clause. However, if any predicate on R participates in a disjunctive clause with a predicate on S , then the disjunctive clause will become part of the **On** condition:

EXAMPLE 3

The query

```

Select *
From R, S
Where R.x * = S.l and R.y = 15

```

is equivalent to

```

Select *
From R Left Outer Join S On ( R.x = S.l )
Where R.y = 15.

```

2 A *disjunctive clause* is a logical expression composed of individual predicates connected using *or*; hence a *disjunctive condition* is a condition in a disjunctive clause. Similarly, a *conjunctive clause* is a logical expression composed of individual predicates connected using *and*; hence a *conjunctive condition* is a condition in a conjunctive clause.

EXAMPLE 4

The query

```
Select *
From R, S
Where R.x * = S.l and ( S.m > 5 or R.x = 15 )
```

is equivalent to

```
Select *
From R Left Outer Join S On ( R.x = S.l and ( S.m > 5 or R.x = 15 ) ).
```

Intuitively, these two examples make sense if you realize that early releases of ASE and ASA would place a predicate in the access plan as soon as it is possible to evaluate it. If the predicate happens to be placed adjacent to a table on the null-supplying side of a TSQL outer join, then the predicate will become part of that outer join's `On` condition. Hence individual predicates, or clauses, that do not refer to any column from the null-supplying table will never become part of an outer join's `On` condition in an ANSI formulation.

3 TSQL outer joins and more complex search conditions

With more complex conditions, earlier releases of both ASE and ASA could produce nondeterministic behavior if other tables are involved in the query, because of the inherent ambiguities with defining TSQL outer join semantics based on predicate placement. In an attempt to limit the number of cases where this could occur, ASE 11.0 and higher will reject a query that contains a table involved in both a TSQL outer join and an inner join.

EXAMPLE 5

The query

```
Select *
From R, S, T
Where R.x * = S.l and S.m = T.a
```

returns error (303,16) with all ASE 11 releases, and error (11054,16) with ASE 12.0. ASA 7 and 8.0 return `SQLCODE -680`. Arguably, this error is desirable if only to prevent misconceptions; if legal, the query above would be equivalent to a query containing only inner joins, because of the null-intolerant join predicate between tables *S* and *T*.

However, while conjunctive inner join conditions to null-supplying tables are not permitted, it is still possible to get nondeterministic behavior when the null-supplying table is referenced in a disjunctive clause.

EXAMPLE 6

Consider the query

```

Select *
From R, S, T
Where R.x * = S.l and R.x = T.a and ( T.b = 0 or S.m = 3 ).

```

Depending on the join strategy chosen by the optimizer, the final disjunctive predicate in the `Where` clause may or may not become part of the `On` condition. In this example, if the join strategy was RST , corresponding to the ASE 12.0 abstract plan notation

```

1  ( nl_g_join
2  ( t_scan R )
3  ( t_scan S )
4  ( t_scan T )
5  )

```

then the final disjunctive condition would be treated as part of the `Where` clause. In ANSI syntax, this would be equivalent to

```

Select *
From R Left Outer Join S On( R.x = S.l), T
Where R.x = T.a and ( T.b = 0 or S.m = 3 ).

```

With the tables R , S , and T populated as above, ASE 11.5 and 11.9.2 choose the join strategy RST , and, with the `Where` clause as above, the query returns the empty set. However, if we alter this query to add a redundant condition on column $T.a$, the query

```

Select *
From R, S, T
Where R.x * = S.l and R.x = T.a and
      T.a = 3 and ( T.b = 0 or S.m = 3 )

```

yields the result

RESULT	x	y	z	l	m	n	a	b	c
	3	4	5	NULL	NULL	NULL	3	4	5

because for this query ASE 11.5 and 11.9.2 choose the join strategy TRS , which corresponds to the ASE 12.0 abstract plan notation

```

6  ( nl_g_join
7  ( t_scan T )
8  ( t_scan R )
9  ( t_scan S )
10 )

```

In effect, ASE 11 interprets this query as

```
Select *
From T, R Left Outer Join S On( R.x = S.l and ( T.b = 0 or S.m = 3 ) )
Where R.x = T.a and T.a = 3.
```

Note that the effect of this is to introduce an outer reference to *T* in the **On** condition for the outer join.

ASE 12.0 will always treat disjunctive clauses involving tables apart from the ones being outer joined as part of the query's **Where** clause; hence, ASE 12.0 will also produce the empty set for this query. ASA 8.0, on the other hand, does not distinguish between disjunctive and conjunctive predicates and will return SQLCODE -680, instead of producing any result, for both of the queries in Examples 5 and 6. ASA 6 servers (or below) would yield ambiguous results in a manner similar to ASE releases prior to 12.0; however, since access plans can vary over time due to a variety of factors, the results are not predictable.

From the previous examples, it is easy to see that how one writes a SQL statement containing TSQL outer joins can affect the query's semantics. In both ASE 12.0 and ASA 8.0, the analysis performed to convert TSQL outer joins to their ANSI equivalents is based on the **Where** clause in the original statement; this is done prior to predicate normalization or the inference of additional conditions. What this means is that two variants of the same **Where** clause can yield different results, even though the search conditions are, on the surface, semantically equivalent.

EXAMPLE 7

At first glance, the following query

```
Select *
From R, T
Where R.x * = T.a and
      ( ( T.b = 2 and R.y = 1 ) or ( T.b = 4 and R.y = 1 ) )
```

should be semantically equivalent to

```
Select *
From R, T
Where R.x * = T.a and R.y = 1 and ( T.b = 2 or T.b = 4 ).
```

However, in all current ASE and ASA releases the first query returns

RESULT	<i>x</i>	<i>y</i>	<i>z</i>	<i>a</i>	<i>b</i>	<i>c</i>
	3	4	5	NULL	NULL	NULL

while the second returns the empty set. Why? This happens because the first query is interpreted as

```
Select *
From R Left Outer Join T On ( R.x = T.a and
    ( ( T.b = 2 and R.y = 1 ) or ( T.b = 4 and R.y = 1 ) ) )
```

while the second is interpreted as

```
Select *
From R Left Outer Join T On ( R.x = T.a and ( T.b = 2 or T.b = 4 ) )
Where R.y = 1.
```

When the conjunctive condition $R.y = 1$ is factored out of the first example, it only refers to the preserved table; hence it is not treated as part of the outer join's `On` condition.

4 Allowable forms of TSQL outer join expressions

Typical TSQL outer join predicates refer to base table columns or view columns; prior to ASA 6.0.3 build 2829, ASA rejected any query that contained a `* = (or = *)` TSQL outer join predicate that had a complex expression as either comparand. ASA now permits TSQL outer join conditions over arbitrary expressions, as long as each expression refers to only one table and contains no subqueries. ASE, on the other hand, not only permits complex expressions in outer join predicates, but also permits subqueries in the expression referencing the preserved table:

EXAMPLE 8

The query

```
Select *
From R, S
Where S.1 = * ( R.x + ( Select T.a From T Where T.a = R.x ) )
```

is a legal TSQL outer join in ASE, whereas

```
Select *
From R, S
Where R.x * = ( S.1 + ( Select T.a From T Where T.a = R.x ) )
```

returns the ubiquitous outer join syntax error (303,16) in ASE 11, and the error (11055,16) in ASE 12.0.

Certainly not all queries containing both TSQL outer joins and subqueries are illegal; see Examples 11 and 12 below for additional examples involving nested queries. However, ASA release 7 and above will return `SQLCODE -680` for both queries in Example 8, as ASA does not permit subqueries to be included in either comparison expression.

With both ASE and ASA, arithmetic expressions with constants or variables are permitted in TSQL outer join comparison predicates. However, one must be

careful to ensure that the expressions on either side of the TSQL outer join comparison only refer to a single table:

EXAMPLE 9

The query

```
Select *
From R, S
Where R.x * = (R.y + S.l)
```

will return a syntax error ((301,16) in ASE 11) and instead should be written as

```
Select *
From R, S
Where (R.x - R.y) * = S.l
```

which is legal.

If multiple outer join predicates refer to the same tables, then it is assumed that there is a single outer join with a conjunctive outer join condition, as follows:

EXAMPLE 10

The query

```
Select *
From R, S
Where R.x * = S.l and R.y * = S.m
```

is equivalent to the ANSI

```
Select *
From R Left Outer Join S On ( R.x = S.l and R.y = S.m ).
```

5 TSQL outer joins and nested queries

Another source of semantic problems for Transact-SQL outer joins is the presence of correlated subqueries, particularly if the subquery's correlation attribute(s) are from one or more null-supplying tables. Examples 11 and 12 below contain existentially-quantified subquery predicates using **Exists**, but similar difficulties occur with other forms of quantified subquery predicates involving **In**, **Some**, **Any** or **All**.

EXAMPLE 11

Consider the query

```
Select *
From R, S
Where R.x * = S.l and
      Exists( Select * From T Where T.a = S.l and T.b = R.y )
```

Correlated subqueries of this form were permitted in older ASE releases up to, and including, ASE 11.5. However, they suffered from semantic problems similar to those illustrated by Example 6 because of predicate placement. Moreover, if the subquery was converted to a join by the optimizer, then the rewritten query

```
Select *
From R, S, T
Where R.x * = S.l and T.a = S.l and T.b = R.y
```

will yield the error (303,16), as was the case with Example 5.

ASE releases above 11.5 now return the outer join syntax error (303,16) if a subquery contains a correlation variable from the null-supplying side of a TSQL outer join (ASA release 7 and above return `SQLCODE -680`). This restriction holds for any type of quantified subquery (`Any`, `In`, `Some`, `All`). Restricting subqueries in this manner also eliminates potential problems in rewriting nested queries as joins.

Interestingly, if the subquery is not part of a quantified predicate, then this restriction only applies to correlated nested queries.

EXAMPLE 12

The query

```
Select *
From T, R
Where T.a * = R.x and R.y = ( Select S.m From S Where S.l = 3 )
```

returns

	<i>a</i>	<i>b</i>	<i>c</i>	<i>x</i>	<i>y</i>	<i>z</i>
RESULT	1	2	3	NULL	NULL	NULL
	2	4	5	NULL	NULL	NULL
	3	4	5	NULL	NULL	NULL

ASE releases permit queries of this form, and in this case the subquery comparison predicate will become part of the outer join's `On` condition. If the subquery is converted to a join (i.e. the `Where` clause in the subquery contains an explicit or implied equivalence condition on the primary key of table *T*), then ASE 11 will convert the subquery to an inner join leading (again) to nondeterministic behavior depending on the join order selected by the optimizer. In contrast, ASE 12.0 will always make the subquery (or its equivalent) part of the outer join's `On` condition. Adding any correlations to the nested query block, however, will once again cause error (303,16). With ASA 7 and 8.0, all such queries are refused with syntax error -680.

6 Disjunctive TSQL outer join conditions

ASE releases permit a TSQL outer join condition to be contained in a disjunctive clause, as the following example illustrates.

EXAMPLE 13

Consider the query

```
Select *
From R, S
Where R.x = S.l or R.y > 5.
```

This query is equivalent to the ANSI

```
Select *
From R Left Outer Join S On ( R.x = S.l or R.y > 5 ).
```

TSQL outer joins contained within disjunctive clauses are also supported in ASA versions 7 and 8; previous ASA releases only considered conjunctive TSQL outer join predicates. A disjunctive clause is the only mechanism through which a predicate on the preserved table can become part of the outer join's `On` condition.

7 Star outer joins

Both ASE and ASA support *star* outer joins. With star outer joins, multiple references to a single table are interpreted as the same table reference, as long as the table references have the same correlation name.

EXAMPLE 14

In ASA, one can issue the query

```
Select *
From R, R
Where R.x > 15.
```

The `From` clause of this query does not specify a Cartesian product of *R* with itself, and is certainly not ANSI syntax. According to the ANSI SQL-99 standard, this query should yield a syntax error, as both references to *R* have the same correlation name. In contrast, ASA treats the second reference to *R* as an identical table reference, so the query is equivalent to

```
Select *
From R
Where R.x > 15.
```

This non-ANSI treatment of duplicate table references in ASA permits one to construct star joins (both inner and outer) more easily, without having to resort to views. Typically, the table in the middle of the 'star' is the preserved table in multiple outer joins, as described in this following example:

EXAMPLE 15

One can write

```
Select *
From R Left Outer Join S On( R.x = S.l ),
      R Left Outer Join T On ( R.x = T.a )
```

rather than the ANSI standard

```
Select *
From ( R Left Outer Join S On( R.x = S.l ) )
      Left Outer Join T On ( R.x = T.a )
```

This feature can be particularly useful if using `Key` or `Natural` joins where ambiguities would result due to multiple foreign key relationships or multiple columns with the same name.

EXAMPLE 16

Similarly, one can specify a star join with TSQL outer joins in both ASE and ASA:

```
Select *
From R, S, T
Where R.x * = S.l and R.x * = T.a
```

Here, the fact that there exists only one reference to *R* is clear based on the `From` clause, but there do exist two outer joins as there are two outer join predicates that do not relate the same pair of tables (contrast this with the query in Example 10).

Support differs between ASE and ASA if the table that is referenced more than once is the null-supplying table in an outer join.

EXAMPLE 17

Consider the following example:

```
Select *
From S Left Outer Join R On ( R.x = S.l ),
      T Left Outer Join R On ( R.x = T.a )
```

This outer join has a problem: we have duplicate references to table *R*, but the two `On` conditions remain independent (there are still two outer joins in this query). Hence, in ASA this query will return a syntax error (SQLCODE -137, table '*R*' requires a unique correlation name).

EXAMPLE 18

With ASE, the equivalent Transact-SQL syntax of the previous example

```
Select *
From R, S, T
Where S.l * = R.x and T.a * = R.x
```

is treated (with ASE 11) or converted (with ASE 12.0) to the ANSI-equivalent

```
Select *
From (S Cross Join T) Left Outer Join R On ( R.x = S.l and R.x = T.a )
```

which is clearly not the same thing as the query in Example 17 as it contains only one outer join. Queries such as this are not supported by any release of Adaptive Server Anywhere.

8 Chained (nested) outer joins

Because duplicate table references may exist in any `From` clause, both ASE and ASA must check for the existence of outer join cycles where the same table is both preserved and null-supplying in the same query block. ASA releases return a specific message (SQLCODE -136) regarding the existence of a cycle, whereas ASE release 11 and below return the standard outer join syntax error (301,16).

In ASE 11.02, nested outer joins could not be specified without using a view; the view provided the explicit mechanism for deciding whether or not the nested outer join was right-deep or left-deep nested. The ASE 11.5 release introduced *chained* TSQL outer joins so that a table can participate as both a preserved and null-supplying table to two (or more) outer joins. It is assumed that the outer joins are left-deep nested, as the following example illustrates:

EXAMPLE 19

Adaptive Server Enterprise 11.5 and above interprets

```
Select *
From R, S, T
Where R.x * = S.l and S.m * = T.b
```

as equivalent to the ANSI

```
Select *
From ( R Left Outer Join S On ( R.x = S.l ) )
      Left Outer Join T On ( S.m = T.b ).
```

It should be clear from the preceding examples that chained outer joins provide a significantly greater opportunity for semantic errors because of the ambiguity of the placement of conditions referencing table *S*. Because of this, ASA does not support chained outer joins, and will return SQLCODE -680 for such queries. ASE 12.0 converts such queries to ANSI syntax, placing such predicates at the ‘deepest’ level within the nested outer join expression, roughly corresponding to the predicate pushdown technique used in ASE 11, but doing so consistently (that is, the result is not affected by the optimization strategy chosen).

9 Summary

Even though the semantics of Transact-SQL outer joins are no longer affected by the optimizer's choice of access plan in current releases of ASA and ASE, iAnywhere Solutions does not recommend their use, particularly because their semantics remain ill-defined (and for the most part undocumented), and because ASE 12 now supports ANSI join syntax. While iAnywhere Solutions has yet to announce the deprecation of TSQL outer joins, we strongly encourage all customers to switch to ANSI join syntax in their applications in anticipation of this announcement.

A Automatically converting statements to ANSI syntax

One can eliminate the guesswork of determining how an ASA server converts TSQL outer joins to ANSI syntax through the use of the **Rewrite** function [1]. This function, introduced in the 7.0 release of Adaptive Server Anywhere, takes an SQL statement as an argument, and returns a rewritten SQL statement as its result. The rewritten SQL statement includes the following syntactic transformations:

1. conversion of **Key** and **Natural** joins to ANSI join syntax utilizing **On** conditions;
2. conversion of TSQL outer joins to ANSI join syntax;
3. removal of duplicate correlation names, resulting in a nested table expression in the **From** clause.

Beginning with ASA 8.0.1, an optional keyword parameter '**ANSI**' to the **Rewrite** function will restrict the set of transformations performed on the original SQL statement to the three above. Without this parameter, the server will apply the following additional transformations to the original statement:

1. rewrite nested queries as joins, possibly including an additional duplicate elimination step (the keyword **Distinct**);
2. merge views or derived tables into the referencing query;
3. eliminate **Distinct** keywords when the result is guaranteed to contain unique rows;
4. pushdown predicates into views or derived tables containing **Group by** or **Union**;
5. eliminate unnecessary joins, altering or inserting additional predicates as required;

6. convert outer joins to inner joins when the inner join will return the same result;
7. fully or partially normalize each search condition in a **Where** or **Having** clause, or in an **On** condition, into conjunctive normal form, simplifying the original condition(s) and adding inferred predicates when they can be exploited during query execution.

These semantic transformations are automatically applied each time the server optimizes any SQL query or **Insert**, **Update**, or **Delete** statement.

EXAMPLE 20

If we apply the **Rewrite** function to the query in Example 7:

```
Select Rewrite( 'Select *
                From R, T
                Where R.x * = T.a and R.y = 1
                   and ( T.b = 2 or T.b = 4 )',
                'ANSI' )
```

the result is the SQL statement (returned as a string)

```
Select *
From R Left Outer Join T On ( R.x = T.a and ( T.b = 2 or T.b = 4 ) )
Where R.y = 1.
```

References

- [1] iAnywhere Solutions, Waterloo, Ontario. *Adaptive Server Anywhere 8.0 Reference Manual*, December 2001.