Join Operations in Relational Databases with Automatic Attributes Renaming

Poliakov S, Buy D, Mohmmed Karam, Israa Jasim AL.kalafa

1Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
2Ministry of Science and technology, Ukraine

*Corresponding author
Poliakov S
Email: sergey.a.polyakov@gmail.com

Abstract: The paper considers inner and outer join of relation operations, as well as set-theoretical operations on relations. Unlike join operations in Codd’s relational algebra the join operations under consideration are based on Kleene and Priest three-valued logic used in SQL, that allows to give the semantics of join operations more precisely. In relational algebra, any relation that does not contain tuples is represented as an empty set. It is impossible to define outer joins for such relations provided we do not know what columns contain an empty relation. In the paper relations with schemas are introduced. That allows the operations to be defined even if one or both operands are empty relations unlike classical relational algebra.

Keywords: relational algebra, relational databases, program theory, programming language semantics, SQL, table join operations.

INTRODUCTION
Join operations play an important role in relational algebra as they allow to restore complex objects, joining many tables. Initially Codd introduced set-theoretical operations on relation and relation join operation in his article [1]. Then relational algebra was extended with operations of outer join. Now there exists a big set of join operations. See, for example, [2-4].

In [5, 6] another approach to SQL algebra was presented. In these works programming algebra of higher-order functions called operators in other programming languages was built for SQL. We describe the improved version of join operations of this program algebra. To avoid column name collision the join operators that are introduced in this article make renaming automatically. The relation definition was extended to schemes as well.

BASIC OPERATIONS ON RELATIONS
Let us clarify relations in terms of nominative sets. Fix two following sets: Atr, whose elements are called attribute names, and Dom - a universal domain of primitive data. We suppose that the special element nil belongs to universal domain and is called an undefined value. Thus, we call an arbitrary finite attribute set $S \subseteq Atr$ a scheme. The tuple of scheme $S$ is called a set of pairs $(a, d)$ where $a \in Atr$, $d \in Dom$ (i.e. a set of pairs (attribute, value)), whose projection by the first component is equal to $S$. Let’s denote operation of tuple projection of $t$ by first component as $pr_1(t)$ and scheme of tuple $t$ as $S(t)$. For given tuple $t$ it is easy to calculate its scheme making projection by the first component $S(t) = pr_1(t)$. Tuples can also be defined as finite mapping from attribute names to universal domain.

The relation of the scheme $S$ is called pair $(r, S)$, where $r$ is finite set of tuples of scheme $S$ which is called a relation state or state. Note, that in the most cases a relation scheme can be obtained from schemes of its tuples. But for an empty state it is impossible. The set of all tuples (relations) of scheme $S$ is denoted by $Tup(S)$ (respectively $Rel(S)$), and set of all tuples (relations) - $Tup$ (respectively $Rel$). Thus $Tup(S) \equiv \{ | pr_1(t) = S \}$, $Rel(S) \equiv \{ (r, S) | r \in 2^{Tup(S)} \}$, $Tup \equiv \cup_{S \in Atr} Tup(S)$, $Rel \equiv \cup_{S \in Atr} Rel(S)$

The scheme may be empty $\emptyset$. For such scheme there exists exactly one tuple denoted by $\varepsilon$ and two states - $\{ \varepsilon \}$ and $\emptyset$. The tuples will be denoted by $t_1, t_2, \ldots$, relations by $R$, $R_1, R_2, \ldots$, relation’s states by $r, r_1, r_2, \ldots$, and schemes by $S, S_1, S_2, \ldots$
Two relations \( R_1 \) and \( R_2 \) are supposed to be **consistent by scheme** if their schemes are equal. We will denote such relation by \( R_1 \equiv R_2 \). Similarly, \( t_1 \equiv t_2 \iff S(t_1) = S(t_2) \) is defined to be consistent by scheme tuples.

The binary relation of **tuple compatibility** \( t_1 \approx t_2 \) is defined as \( \forall a \in S(t_1) \& a \in S(t_2) \Rightarrow t_1(a) = t_2(a) \). In other words, two tuples are compatible if their attributes with the equal names have equal values. Tuples without equal attribute name are always compatible.

The binary operation \( U: Tup \times Tup \rightarrow Tup \) of the compatible tuple union is defined as follows: \( t_1 \cup t_2 = \{(a,d) | (a, d) \in t_1 \lor (a, d) \in t_2 \} \).

Its definition domain is \( dom \cup \equiv \{ (t_1, t_2) | t_1, t_2 \in Tup \lor t_1 \approx t_2 \} \).

Along with elementary attribute names we will consider complex names such as \( a_1.a_2 \). These names will be used to avoid name ambiguity in tuples. Let’s introduce special function \( a \Rightarrow Tup \rightarrow Tup \) that is parametrized by attribute name \( a \). It is defined by formula \( a \Rightarrow t = \{(a, a, d_j) | (a, d_j) \in t\} \).

This function will be called **the prefixation**.

The operation of **total union** of tuples \( \cup_{a_1, a_2}: Tup \times Tup \rightarrow Tup \) is total function parametrized by attribute names \( a_1 \) and \( a_2 \). It is defined as follows: \( t_1 \cup_{a_1, a_2} t_2 = \{t_1^* \cup t_2^* \mid t_1^* = (a_1 \Rightarrow t_1) \& t_2^* = (a_2 \Rightarrow t_2) \} \).

For schemes the prefixation is defined as follows: \( S_1 \cup_{a_1, a_2} S_2 = a_1 \Rightarrow (S_1) \lor a_2 \Rightarrow (S_2) \), where \( a_1 \Rightarrow (S) \equiv \{ a, a | a \in S \} \).

We will write \( t(a) \) to receive the value of \( a \).

We will say that a tuple belongs to a relation if it belongs to the relation’s state \( t \in R \equiv \exists r \exists S(R = (r, S) \land t \in r) \).

**SET-THEORETIC RELATION OPERATIONS**

Let’s \( R_1 \) be a relation \( (r_1, S_1) \) and \( R_2 \) be a relation \( (r_2, S_2) \) where \( S_1 = S_2 = S \).

Union of relations \( R_1 \cup R_2 = (r_1 \cup r_2, S) \)

Intersection of relations \( R_1 \cap R_2 = (r_1 \cap r_2, S) \)

Difference of relations \( R_1 \setminus R_2 = (r_1 \setminus r_2, S) \).

The domain of these relation definition is a set of consistent pairs of relations \( \{(R_1, R_2) | R_1 \equiv R_2 \} \).

The resulting relation state is calculated, respectively, as theoretical-set union, intersection and the difference of arguments states. Obviously, the scheme of resulting relation coincides with input relation schemes. We can say that these operations are restriction of the theoretical-set operation on pairs of consistent relations.

**INNER JOIN OPERATIONS**

We define **cross join operation** \( \times_{a_1, a_2} \) as total binary function of a type \( \times_{a_1, a_2}: Rel \times Rel \rightarrow Rel \) parametrized by attribute names \( a_1, a_2, a_1 \neq a_2 \).

Let \( R_1 \) be relation \( (r_1, S_1) \) and \( R_2 \) be relation \( (r_2, S_2) \). Then, cross join is defined by formula \( \times_{a_1, a_2} (R_1, R_2) = (r_3, S_1 \cup_{a_1, a_2} S_2) \), where \( r_3 = \{ t_1 \cup_{a_1, a_2} t_2 | t_1 \in r_1 \& t_2 \in r_2 \} \).

Natural join operation \( \bowtie_{a_1, a_2} \) or equijoin is total function of a type \( \bowtie_{a_1, a_2}: Rel \times Rel \rightarrow Rel \). Let \( R_1 \) be relation \( (r_1, S_1) \) and \( R_2 \) be relation \( (r_2, S_2) \). Then, natural join is defined by formula \( \bowtie_{a_1, a_2} (R_1, R_2) = (r_3, S_1 \cup_{a_1, a_2} S_2) \), where \( r_3 = \{ t_1 \cup_{a_1, a_2} t_2 | t_1 \in r_1 \& t_2 \in r_2 \& t_1 \approx t_2 \} \).

Next operation is **join by attributes** \( \bowtie^B_{a_1, a_2} \) of a type \( Rel \times Rel \rightarrow Rel \), where \( B \) is a set of attribute names and \( a_1, a_2 \) are names such that \( a_1 \neq a_2 \). This is partial parametric function with domain of definition \( dom \bowtie^B_{a_1, a_2} \equiv \{(R_1, R_2) | B \subseteq S(R_1) \land S(R_2)\} \)

It is defined by formula \( \bowtie^B_{a_1, a_2} (R_1, R_2) = (r_3, S_1 \cup_{a_1, a_2} S_2) \), where \( r_3 = \{ t_1 \cup_{a_1, a_2} t_2 | t_1 \in r_1 \& t_2 \in r_2 \& t_1(B) \subseteq t_2(B) \} \).

Let \( p: Tup \rightarrow Bool \) be, as generally accepted, a partial binary predicate. Under **join on predicate** \( p \) we understand a partial operation \( \bowtie^p_{a_1, a_2} \) of a type \( Rel \times Rel \rightarrow Rel \), where \( p \) is a predicate and \( a_1, a_2 \), such that \( a_1 \neq a_2 \) are attribute names. Let \( dom p \) be domain of definition of a predicate \( p \). The domain of definition of operation join on a predicate is \( dom \bowtie^p_{a_1, a_2} \equiv \{ (R, S) | (r, S) \subseteq dom p \} \). The operation is defined by formula \( \bowtie^p_{a_1, a_2} (R_1, R_2) = (r_3, S_1 \cup_{a_1, a_2} S_2) \), where \( r_3 = \{ t | t = t_1 \cup_{a_1, a_2} t_2 \& t_1 \in r_1 \& t_2 \in r_2 \& p(t) = True \} \).

Given above-mentioned join operation specification, we can see that cross-join, natural-join and equijoin are derived from join on predicate operation.

Let \( p \) be always true predicate \( p \equiv true \), then \( \times_{a_1, a_2} (R_1, R_2) \equiv \bowtie^true_{a_1, a_2} (R_1, R_2) \)

Let \( A \) be set of attribute names \( \{ a_1, a_2, ..., a_n \} \) and \( c_1 \) and \( c_2 \), \( c_1 \neq c_2 \) be attribute prefixes. Then, predicate \( p^A_{c_1, c_2} \) is defined as follows

\[
p^A_{c_1, c_2}(t) = \begin{cases} 
true, & \text{if } t(c_1, a_i) = t(c_2, a_i), i = 1...n \\
false, & \text{otherwise}
\end{cases}
\]

For given relations \( R_1 = (r_1, S_1) \), \( R_2 = (r_2, S_2) \) and attribute prefixes \( c_1, c_2 \) let’s designate
Results schemes are obtained by input relation schemes union $S_1 \cup a_{1,2} S_2$.

Under outer left join induced by operation $\varphi$, we understand the operation of a type $\varphi : Rel \times Rel \rightarrow Rel$, where $dom \varphi \subseteq dom \varphi_{a_{1,2}}$, that is calculated by a formula $\varphi((R_1, R_2)) = \varphi_{a_{1,2}}(R_1, R_2) \cup (R_1 \varphi a_{1,2} R_2) \times a_{1,2} ((t_{null}^N), S_2)$.

Under outer right join we consider the operation of a type $\varphi : Rel \times Rel \rightarrow Rel$, where $dom \varphi \subseteq dom \varphi_{a_{1,2}}$, that is calculated by a formula $\varphi((R_1, R_2)) = \varphi_{a_{1,2}}(R_1, R_2) \cup (R_2 \varphi a_{1,2} R_1)$.

Under outer full join we consider the operation of a type $\varphi : Rel \times Rel \rightarrow Rel$, where $dom \varphi \subseteq dom \varphi_{a_{1,2}}$, that is calculated by a formula $\varphi((R_1, R_2)) = \varphi_{a_{1,2}}(R_1, R_2) \cup (R_2 \varphi a_{1,2} R_1)$.

Under union join we consider the operation of a type $\varphi : Rel \times Rel \rightarrow Rel$, where $dom \varphi \subseteq dom \varphi_{a_{1,2}}$, that is calculated by a formula $\varphi((R_1, R_2)) = \varphi_{a_{1,2}}(R_1, R_2) \cup (R_1 \varphi a_{1,2} R_2) \times a_{1,2} ((t_{null}^N), S_2) \cup (R_2 \varphi a_{1,2} R_1)$.

Resulting relation schemes are still being built by input relation schemes union.

Thus, the left join is designed to consider the first(left) relation tuples that do not participate in inner join, the right join – to consider the tuples of the second right relation that do not participate in inner join, full join - to consider for both arguments not being involved in inner join. Finally, union join, unlike previous operations do not replenish the result of inner union and only builds appropriately extended tuples of both arguments that are not involved in union join.

**JOIN OPERATION STRUCTURE**

Let us return to the inner join operation SQL.

For cross join there is no sense in introducing outer join because the left, right and full joins coincide with it, and union join will always be an empty table (because all table-arguments strings are involved in forming the results of cross join). Moreover, in modern SQL, including the latest versions, outer union join is supported only for natural join. Family operation structure of SQL-like languages is given.

Note, that a preliminary design should be modified to consider the special element null role while determining compatible tuples: the tuples are
compatible if common attributes have the same values in tuples, the values differing from null.

The same objective can be achieved by considering the following clarification in three-valued logic.

\[
d_1 \equiv d_2 \iff \begin{cases} 
& \text{true, if } d_1 = d_2, d_1 \neq \text{null}, d_2 \neq \text{null} \\
& \text{false if } d_1 \neq d_2, d_1 \neq \text{null}, d_2 \neq \text{null} \\
& \text{unknown if } d_1 = \text{null} \lor d_2 = \text{null}
\end{cases}
\]

Where all \( d_1, d_2 \in D \); here unknown – an undefined Boolean value. Then, the compatibility relation that already meets the three-valued predicate is modified as follows: \( t_1 \equiv t_2 \iff \&_{\alpha \in \Phi} t_1(a) =_t t_2(a) \), where \( S_t \) is a scheme of \( t_i \), \( i = 1,2 \). Note, that in SQL-like languages the equality is specified as a predicate \( =_t \). In addition, a given definition is the manifestation of the overall situation in extending predicates on null-values: predicates keep null values, i.e. if at least one argument is equal to null, the result is Boolean value unknown.

In the same way, functions in extending on null-value keep it: if, at least, one argument coincides with null then the result is null. Thus, the situation is similar to natural extension of partial functions but in SQL extended functions are considered as the partial (i.e., division).

### Table 1: Operation join structure

<table>
<thead>
<tr>
<th>Inner join</th>
<th>Join operation predicate by predicate</th>
<th>Outer join</th>
<th>&lt;br&gt;left</th>
<th>right</th>
<th>full</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross join</td>
<td>( p(t) )</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Natural join</td>
<td>( p^*(t) )</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Join by attribute</td>
<td>( p(a) )</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Join on predicate</td>
<td>( p(t) )</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

### CONCLUSION

The join operations can be considered as a special type of Cartesian product on the case when arguments are set of tuples called relations. We want the result of the operation to be a relation too. To achieve this, an operation of join must concatenate two tuples into new tuple. It is easy to do when names of tuple attributes are not intersected. Otherwise we will have the ambiguity of names. To avoid this the join operations introduced in the paper make attributes renaming automatically.

### REFERENCES