A Universal Relational Operator

Kim S. Larsen
Erik M. Schmidt
Michael I. Schwartzbach

DAIMI PB – 297
December 1989
A Universal
Relational Operator

Kim S. Larsen¹
Michael I. Schwartzbach²
Erik M. Schmidt³

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Århus C, Denmark

Abstract

We present a single relational operator which, in combination with a simple
core language for manipulating atoms and tuples, generalizes all standard
unary and binary operators in relational databases, while permitting a more
intuitive query style. The new operator, factor, is based on a unique factorization
of relations. We present an example language and demonstrate
how the usual operators appear as simple and intuitive instances of factor.
We further show that many new operators and combinations of old ones
can be expressed in concise terms using factor. The factor versions will
always be evaluated as efficiently as the originals and will sometimes even
lead to a speed-up.

1 Introduction

We present a new relational operator, factor, which is universal in the
sense that it can replace all the usual operators of relational algebra that
are described in e.g. [Mai83].

The usual operators appear as simple and intuitively appealing special
cases, and the factor operator is quite different in flavor from the ones
it replaces. It is an n-ary operator based on a unique factorization of

¹E-mail address: kimskak@daimi.dk
²E-mail address: mis@daimi.dk
³E-mail address: emschmidt@daimi.dk
collections of relations. From this decomposition of the original relations one can specify the resulting relation by means of a very simple core language, allowing basic manipulations of atoms and tuples, together with Cartesian product, which is the simplest possible "horizontal" relational operator.

The factor operator is inspired by the group by operator [Gra81], but it is far more general and fundamental in its nature.

Apart from encompassing the usual relational operators, factor can also express "the next 700 variations", as well as combinations of existing ones and powerful new operators. Many computations on relations seem to be expressible in a more direct and intuitive fashion.

We propose to perform relational computations in three steps.

The first step is to decompose the relation arguments. The second step is to perform simple computations on these smaller components. The third and final step is to combine the individual results from step two.

The decomposition in step one belongs to a family of factorizations, as described in section 2. The computation in step two is specified by a small core language described in section 3. The combination in step three is always the union of the results from step two.

In section 4 we describe the full syntax and semantics of the factor operator. In sections 5, 6, and 7 we demonstrate how all standard relational operators, and many more, can be expressed in terms of factor.

In section 8 we observe that factor expressions can be evaluated efficiently. In fact, the implementation of all the usual operators in terms of factor will preserve their original complexities. One can even obtain a speed-up in certain situations. Also, factor expressions are well-suited for parallel evaluation.

2 Factorizations

A tuple is a finite partial function from attribute names to atoms. A relation is a pair $R = (\sigma(R), \tau(R))$ where $\sigma(R)$, the schema, is a finite set of attribute names, and $\tau(R)$ is a finite set of tuples with common domain $\sigma(R)$. 
The factorization is performed on a collection of relations, relative to a subset of their common attribute names. Operationally, the decomposition components can be found as follows. All tuples of all relations are projected onto the selected attribute names and duplicates are removed. This yields a set of component tuples. For each tuple in this set and for each relation argument, we determine a component relation, which contains exactly those complementary tuples that combined with the component tuple are contained in this relation argument.

**Definition 2.1** Let \( R_1, \ldots, R_n \) be relations and \( \{a_1, \ldots, a_k\} \subseteq \cap \sigma(R_j) \) a set of attribute names. The factorization of the \( R_j \)'s on the \( a_i \)'s consists of

- a sequence of component tuples \( \phi_1, \ldots, \phi_m \) with common domain \( \{a_1, \ldots, a_k\} \)
- for each \((i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}\), a component relation \( \Theta_{ij} \) with schema \( \sigma(R_j) - \{a_1, \ldots, a_k\} \)

such that

1) the following \( n \) equations hold

\[
\forall j : R_j = \sum_{i=1}^{m} \{\phi_i\} \times \Theta_{ij}
\]

where \( \{\phi_i\} \) denotes the singleton relation whose only tuple is \( \phi_i \), \( + \) is interpreted as union, and \( \times \) as Cartesian product of relations. These \( n \) equations can be concisely expressed as the following matrix equation

\[
(\{\phi_1\}, \{\phi_2\}, \ldots, \{\phi_m\}) \begin{pmatrix}
\Theta_{11} & \Theta_{12} & \cdots & \Theta_{1n} \\
\Theta_{21} & \Theta_{22} & \cdots & \Theta_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
\Theta_{m1} & \Theta_{m2} & \cdots & \Theta_{mn}
\end{pmatrix} = (R_1, R_2, \ldots, R_n)
\]

2) all the \( \phi_i \)'s are pairwise different, i.e. \( \forall i, j : i \neq j \Rightarrow \phi_i \neq \phi_j \)

3) no row of the \( (\Theta_{ij}) \) matrix has all "zeroes", i.e. \( \forall i \\exists j : \tau(\Theta_{ij}) \neq \emptyset \)
Proposition 2.2 The factorization always exists and is unique up to reordering of the $\phi_i$ sequence.

Proof We can find $\{\phi_1, \ldots, \phi_m\}$ as $\cup R_j \downarrow a_1, \ldots, a_k$, where $\downarrow$ is projection. Now, $\Theta_{ij}$ is found by selecting from $R_j$ where $a_1, \ldots, a_k$ equals $\phi_i$ and projecting this over $\sigma(R_j) - \{a_1, \ldots, a_k\}$ (this is the same as dividing $R_j$ by $\{\phi_i\}$). Clearly, this satisfies the matrix equation. Also, since each $\phi_i$ belongs to $\cup R_j \downarrow a_1, \ldots, a_k$ then it must belong to some $R_j \downarrow a_1, \ldots, a_k$ and, hence, this particular $\Theta_{ij}$ must be non-zero. This demonstrates existence. For uniqueness, we observe that for any factorization the matrix equation implies $\cup R_j \downarrow a_1, \ldots, a_k \subseteq \{\phi_1, \ldots, \phi_m\}$. Since every row has a non-zero element we get the other inclusion, too. As the $\Theta_{ij}$'s are determined uniquely from the $\phi_i$'s, the factorization is unique up to a reordering of the $\phi_i$'s. $\square$

Example: Let $R_1$ and $R_2$ be the two relations

\[
\begin{array}{|c|c|}
\hline
A & B \\
\hline
a_1 & b_1 \\
a_2 & b_2 \\
a_3 & b_3 \\
a_4 & b_4 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|c|}
\hline
B & C & D \\
\hline
b_2 & c_1 & d_1 \\
b_2 & c_2 & d_2 \\
b_4 & c_3 & d_3 \\
b_5 & c_4 & d_4 \\
\hline
\end{array}
\]

and

The factorization of $R_1, R_2$ on $B$ is

\[
\left( \begin{array}{cccc|c}
B & B & B & B & B \\
b_2 & b_3 & b_4 & b_5 \end{array} \right) = (R_1, R_2)
\]

\[
\left( \begin{array}{c|c|c|}
A & C & D \\
\hline
a_1 & & \\
A & C & D & \\
\hline
a_2 & c_1 & d_1 & \\
a_3 & c_2 & d_2 & \\
A & C & D & \\
\hline
a_4 & c_3 & d_3 & \\
A & C & D & \\
\hline
a_5 & c_4 & d_4 & \\
\end{array} \right)
\]
3 The Language

The set of expressions contains a minimal core language for manipulating atoms and tuples

\[ e ::= \alpha \quad \text{atom expressions} \\
| [a:e] \quad \text{tuple formations} \\
| e_1 e_2 \quad \text{tuple perturbations} \\
| e.a \quad \text{tuple inspections} \\
| e \setminus a \quad \text{tuple restrictions} \\
| 0 \mid 1 \quad \text{relation constants} \\
| \{e_1, \ldots, e_k\} \quad \text{relation formations} \\
| b:e \quad \text{gates} \\
| f(e) \quad \text{homomorphisms} \]

We also provide two operators on relations

\[ e_1 \times e_2 \quad \text{Cartesian products} \\
| \textbf{factor} \ldots \textbf{on} \ldots \textbf{do} \ldots \quad \text{factorization operators} \]

The atom expressions are left unspecified but are intended to be entirely standard; certainly, they will include the booleans.

A tuple formation denotes a partial function by its (singleton or empty) graph associating attribute names with values. Tuple perturbation is a binary operator on partial functions, where the left-hand function is updated with the definitions of the right-hand function. A tuple inspection merely applies a partial function to an argument. A tuple restriction removes an argument from the domain of a partial function.

The relation constants denote the 0- and 1-element for Cartesian product, i.e. \( 0 = (\emptyset, \emptyset) \) and \( 1 = (\emptyset, \{[]\}) \). A relation formation constructs a relation from a non-empty set of tuples with common domain.

In the gate expression \( b:e \) the expression \( b \) denotes a boolean and \( e \) denotes a relation. If \( b \) is true, then the result is \( e \); otherwise, the result is \( (\sigma(e), \emptyset) \).

Finally, a homomorphism \( f \) is a function from relations to atoms such that \( f(R_1 \cup R_2) \) equals \( f(R_1) \oplus_f f(R_2) \), where \( \oplus_f \) is an associative and commutative operator on the image of \( f \). The set of homomorphisms
is left unspecified but can include such functions as is-empty, and, or, min, and max.

4 The Factor Operator

The syntax of the factor operator is

\[
\text{factor } R_1, R_2, \ldots, R_n \\
on a_1, a_2, \ldots, a_k \\
do e
\]

where \( n \geq 1 \), the \( R_j \)'s are relations, \( k \geq 0 \), \( \{a_1, a_2, \ldots, a_k\} \subseteq \cap \sigma(R_j) \) is a set of attribute names, and \( e \) is an extended expression denoting a relation. An extended expression allows the following extra constructs

\[
e ::= \text{tup} | \text{rel}(j)
\]

factorization components

We allow the following variation: If one merely writes

\[
\text{factor } R_1, R_2, \ldots, R_n \text{ do } e
\]

then the factorization is performed on \( \cap \sigma(R_j) \), i.e. all the common attributes.

The semantics of factor is the function taking \( R_1, R_2, \ldots, R_n \) to the result of the following computation. Step one: A factorization of \( R_1, R_2, \ldots, R_n \) on \( a_1, a_2, \ldots, a_k \) is determined. Assume that this results in \( m \) component tuples. Step two: For each \( \phi_i \) and \( (\Theta_{i1}, \Theta_{i2}, \ldots, \Theta_{in}) \) the expression \( e \) is evaluated in an environment where \( \text{tup} = \phi_i \) and for each \( 1 \leq j \leq n \), \( \text{rel}(j) = \Theta_{ij} \). Step three: The result is the union of these \( m \) values. If \( m = 0 \) then the result is, of course, the empty relation with the appropriate schema, determined from \( e \).

Notice that both the decomposition and the combination can be expressed in terms of the two simplest relational operators, union and Cartesian product. In between, one can modify the components.

As a trivial example, where no modification takes place, observe that \( R_j \) equals

\[
\text{factor } R_1, R_2, \ldots, R_n \text{ on } a_1, a_2, \ldots, a_k \text{ do } \{\text{tup}\} \times \text{rel}(j)
\]
for any legal choice of $a_i$'s.

**Proposition 4.1** The factor operator is well-defined, i.e.

1) the schema of the value of $e$ is the same for each environment and can be statically determined (which is necessary to define the schema of an empty result).

2) the result is independent of the ordering of the $\phi_i$'s.

**Proof**

1) the schemas of $\phi_i$ and $\Theta_{ij}$ are independent of $i$. Hence, the schema of $e$ is the same in all environments. By a simple induction one can show that the domain of any tuple expression can be statically determined, as can the schema of any relation expression.

2) the result is independent of the ordering of the $\phi_i$'s since the factorization is unique up to such reorderings (Prop. 2.2) and union is associative and commutative.

A small amount of syntactic sugar will prove convenient. If an attribute name $a$ appears in an extended expression in place of an atomic value, then it denotes $\text{tup}.a$. Also, we shall write $\text{rel}$ rather than $\text{rel}(1)$ when factor takes only a single argument.

**Example:** If $R_1$ and $R_2$ are the two relations from section 2, then the result of

$$\text{factor } R_1, R_2 \text{ on } B \text{ do } \text{rel}(1) \times \text{rel}(2)$$

can be computed as

$$\frac{A}{a_1} \times \frac{C\ D}{a_2} + \frac{A}{c_1} \times \frac{C\ D}{d_1} + \frac{A}{c_2} \times \frac{C\ D}{d_2} + \frac{A}{c_3} \times \frac{C\ D}{d_3} + \frac{A}{c_4} \times \frac{C\ D}{d_4}$$
which equals

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_2</td>
<td>c_1</td>
<td>d_1</td>
</tr>
<tr>
<td>a_2</td>
<td>c_2</td>
<td>d_2</td>
</tr>
<tr>
<td>a_4</td>
<td>c_3</td>
<td>d_3</td>
</tr>
</tbody>
</table>

5 Unary Operators

To begin with, we investigate the simpler case of the unary factor operator

\[
\text{factor } R \text{ on } a_1, \ldots, a_k \text{ do } e
\]

where all the \(a_i\)'s are attribute names of \(R\).

We give a translation of the standard unary relational operators

\[
\text{project } R \text{ on } a_1, \ldots, a_k \equiv \\
\text{factor } R \text{ on } a_1, \ldots, a_k \text{ do } \{\text{tup}\}
\]

\[
\text{select } R \text{ where } b \equiv \\
\text{factor } R \text{ do } b:\{\text{tup}\}
\]

\[
\text{rename } R \text{ by } a_1 \rightarrow a_2 \equiv \\
\text{factor } R \text{ do } \{\text{tup}\backslash a_1[a_2:a_1]\}
\]

We can also define the following two non-standard operators [Gra84, Gra81]

\[
\text{extend } R \text{ by } a_1 := e \equiv \\
\text{factor } R \text{ do } \{\text{tup}[a:e]\}
\]

\[
\text{group } R \text{ by } a_1, \ldots, a_k \text{ creating } a := f() \equiv \\
\text{factor } R \text{ on } a_1, \ldots, a_k \text{ do } \{\text{tup}[a:f(\text{rel})]\}
\]

Many variations of these basic operators are readily available. One example is a reduce operator which removes the specified attributes

\[
\text{reduce } R \text{ by } a_1, \ldots, a_k \equiv \\
\text{factor } R \text{ on } a_1, \ldots, a_k \text{ do rel}
\]
Another example is an \texttt{update} operator which works like \texttt{extend}, except that it assigns to an existing attribute

\[
\text{update } R \text{ by } a := e \equiv \\
\text{factor } R \text{ do } \{\text{tup}[a := e]\}
\]

Notice that the translation is the same as that of \texttt{extend}. It is often the case that \texttt{factor} expressions turn out to be more general than one originally intended.

Many combinations of ordinary operators can conveniently be expressed by a single \texttt{factor} expression. Consider as an example the following expression where $R$ is a relation with schema $\{a, b, c, d, x, y\}$

\[
\text{project} \\
\text{extend} \\
\text{select } R \text{ where } x > y \\
\text{by } z := x + y \\
\text{over } a, b, c, d, x, z
\]

Using \texttt{factor} we can write

\[
\text{factor } R \text{ do } x > y: \{\text{tup}[z := x + y] \setminus y\}
\]

Two points are noteworthy in connection with this example. Firstly, the \texttt{factor} expression does not need to know the incidental attributes $\{a, b, c, d\}$. Secondly, the computation is clearly one that should be performed on each tuple individually. This is evident in the \texttt{factor} expression, which in this situation basically says "for all tuples in $R$ do ...". In the former expression one has to split this simple computation scheme out into operations on three different relations. In conclusion, this \texttt{factor} expression is not only shorter (and more efficient) but also considerably easier to program.

\section{Binary Operators}

The usual binary operators, as well, appear as simple binary \texttt{factor} expressions. Cartesian product is, of course, built-in.

We can write the union operation as
union \( R_1 \) and \( R_2 \) \( \equiv \)
\[
\text{factor } R_1, R_2 \text{ do } \{\text{tup}\}
\]

which is really an extension. If the two relations have different schemas, then it produces the union of the projections over the common attributes names.

Intersection is straightforward. Notice that if \( R_1 \) and \( R_2 \) have the same schema and \( \text{rel}(1) \) equals \( \text{rel}(2) \) then they both equal 1

**intersect** \( R_1 \) and \( R_2 \) \( \equiv \)
\[
\text{factor } R_1, R_2 \text{ do } \text{rel}(1) = \text{rel}(2) : \{\text{tup}\}
\]

Another way to obtain the intersection is as a special case of the **join** operator

**join** \( R_1 \) and \( R_2 \) \( \equiv \)
\[
\text{factor } R_1, R_2 \text{ do } \text{rel}(1) \times \{\text{tup}\} \times \text{rel}(2)
\]

This is a very intuitive presentation of **join**: The different parts of \( R_1 \) and \( R_2 \) are stuck together using the available “glue” – the common **tup**’s.

The difference of two relations is

**difference** \( R_1 \) and \( R_2 \) \( \equiv \)
\[
\text{factor } R_1, R_2 \text{ do } \text{rel}(2) = 0 : \{\text{tup}\}
\]

As before, this expression is very easy to understand: We take the **tup**’s that do not belong to \( R_2 \).

We can play the game of variations for binary operators, too. A very commonly emulated operator is **combine**, which joins two relations together while removing the “glue”. This is useful when a relation has been split in two by the introduction of an extra key attribute in each, and we want to recover the original relation

**combine** \( R_1 \) and \( R_2 \) \( \equiv \)
\[
\text{factor } R_1, R_2 \text{ do } \text{rel}(1) \times \text{rel}(2)
\]

Again, this expression follows directly from the definition of **join** and is easily understood.

The symmetric difference of two relations can be found as follows
symdiff \( R_1 \) and \( R_2 \) ≡ 
\[ \text{factor } R_1, R_2 \text{ do rel}(1) \neq \text{rel}(2): \{\text{tup}\} \]

A variation on this example shows a binary operator that factorizes on something beside all common attributes. Consider a relation in which the attributes \( a, b, c \) constitute a key. We have two different versions of what is intended to be the same relation, and we want to get the key values for which the information in the two relations disagree, i.e. we want to check for inconsistencies in our database

\[
\text{check } R_1 \text{ and } R_2 \equiv \\
\text{factor } R_1, R_2 \text{ on } a, b, c \text{ do rel}(1) \neq \text{rel}(2): \{\text{tup}\}
\]

This is almost a literal translation of: If the information is inconsistent, then include the key value. In comparison, using ordinary operators we would end up with the far less transparent expression

\[
\begin{align*}
\text{project} \\
\text{difference} \\
\text{union } R_1 \text{ and } R_2 \\
\text{and} \\
\text{intersect } R_1 \text{ and } R_2 \\
\text{over } a, b, c
\end{align*}
\]

Finally, we present an example of a 2-level \textbf{factor}. The \textbf{divide} operator is defined as

\[
R_1 / R_2 = \max \{ D \mid D \times R_2 \subseteq R_1 \}
\]

where \( \sigma(R_2) \subseteq \sigma(R_1) \). It is usually quite complicated to derive. We can write is as

\[
\text{divide } R_1 \text{ and } R_2 \equiv \\
\text{factor } R_1, R_2 \text{ do} \\
\text{factor rel}(1) \text{ do} \\
\{\text{tup}\} \times R_2 \subseteq R_1: \{\text{tup}\}
\]

which closely follows the definition. The two factors provide the \( \sigma(R_1) - \sigma(R_2) \) tuples of \( R_1 \). We then select those that combined with all of \( R_2 \) is contained in \( R_1 \). In comparison, a more standard derivation of \textbf{divide} is
difference
  project $R_1$ over $d_1, d_2, \ldots, d_k$

and

project
difference
  join
  project $R_1$ over $d_1, d_2, \ldots, d_k$

and

$R_2$

and

$R_1$

over $d_1, d_2, \ldots, d_k$

This is not very intuitive; furthermore, one needs explicit knowledge of
$\sigma(R_1) - \sigma(R_2) = \{d_1, d_2, \ldots, d_k\}$. In [Gra84] divide is derived from two
group by’s, but it involves renamings and projections, and it gets in-
creasingly complex with the size of $k$.

7 General Operators

The binary operators union, intersect, and others immediately scale up
to $n$-ary operators, e.g.

\[
\text{union } R_1 \text{ and } R_2 \text{ and } \ldots \text{ and } R_n \equiv \\
\text{factor } R_1, R_2, \ldots, R_n \text{ do } \{\text{tup}\}
\]

Apart from these handy generalizations one can write new operators that
are inherently more than binary. Consider as an example a novel safejoin
operator which takes as arguments $R_1, R_2, R_3$, where $\sigma(R_1) \cap \sigma(R_2) = 
\sigma(R_3)$. The result is the subset of the join of $R_1$ and $R_2$ for which the
projection onto the common attributes is contained in $R_3$, i.e. only the
glue mentioned in $R_3$ is “safe”. Using factor this looks like

\[
\text{safejoin } R_1 \text{ and } R_2 \text{ using } R_3 \equiv \\
\text{factor } R_1, R_2, R_3 \text{ do } \{\text{tup}\} \times \text{rel}(1) \times \text{rel}(2) \times \text{rel}(3)
\]

Such operators can, of course, generally be written as more cumbersome
combinations of binary operators.
8 Efficiency

The factor operator can be implemented quite efficiently. By sorting and merging one can compute the factorization of \( n \) relations each with \( T \) tuples in time \( O(nT \log(T)) \). The time for the entire factor operator must furthermore include the time for computing the union of the extended expressions. For example, the binary join can be computed in time \( O(T \log(T) + J) \), where \( J \) is the size of the result, and project can be computed in time \( O(T \log(T)) \).

One can make an obvious improvement by observing that if the extended expression \( e \) in a unary factorization on all attributes "factor \( R \) do \( e \)" is injective, then no sorting is needed, and the operator can be implemented in linear time. By injective we mean that the non-empty results of the extended expression are pairwise disjoint. Ignoring the properties of atom expressions, one can, using symbolic evaluation, statically determine when such a factor operator is injective; furthermore, the class of injective expressions can be extended by including knowledge about the different atom expressions, or about key attributes in the relations.

We observe that the expressions in select and in (legal) extend and rename are injective and that, consequently, these operators will run in time \( O(T) \). We also note that any clever implementation "tricks" for these operators, which make the running time sublinear, can be inherited by the implementation of factor. Hence, the factor version of every standard relational operator will have the same complexity as the original one.

We can, in fact, quite often do better. As demonstrated earlier, many combinations of standard operators can conveniently be expressed as a single factor operator. In general, we can gain a constant factor in these situations, since some temporary results are eliminated, and fewer sortings and copyings are needed. We can avoid sorting altogether if the combined query disguised an injectivity that is apparent in the single factor expression.

With this knowledge the factor technology can serve as the foundation for an interesting database implementation, where one needs only consider a single operator.

Furthermore, factor can be implemented efficiently on various multi-
processor architectures. The very formulation of the factorization as a vector/matrix product indicates the possibility for use of massive parallelism in the implementation. The basic operators on any architecture will be parallel sorting and merging combined with simultaneous computation on individual parts (the $\phi_i$'s and $\Theta_{ij}$'s). It seems that both vector processors, hypercubes, and a properly designed network of transputers should be able to support this sort of computation efficiently. Of course, the inherent parallelism in the definition of factorizations can also be exploited in a network of sequential machines supporting a distributed database.

9 Conclusion and Future Work

The factor operator can conveniently express all standard relational operators, and many more, without any loss of efficiency. Hence, one can without cost reap the benefits of our proposal: Greater expressiveness, a more intuitive query style, and hopes for increased efficiency.

In a concrete language proposal [LSSJ], we have combined factor with fully recursive polymorphic higher-order functions, which yields a very powerful tool. Implementation is currently being undertaken.

We are working on a logical characterization of factor to determine its computational strength. We already know that factor is stronger than the collection of usual relational operators, as the latter all yield linear\(^4\) functions and the former does not. On the other hand it not possible (we conjecture) to calculate e.g. transitive closure with the use of factor.

We are interested in developing a calculus of factor expressions, which will enable query optimizations. We are optimistic that this will be possible, since we only have a single operator, and the core language is quite expressive. Certainly, we can easily compact combinations of translations of standard unary operators into the appropriate single factor operator.

We also want to develop optimal algorithms for detecting injectivity under various assumptions about the atom expressions.

\(^4\)An $n$-ary function $f$ on relations is linear if for each argument $f(\ldots, R_1 \cup R_2, \ldots) = f(\ldots, R_1, \ldots) \cup f(\ldots, R_2, \ldots)$.
References


