A New Formalism for Relational Algebra

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

Computer Science Department, Aarhus University, Aarhus, Denmark

Keywords: databases, relational algebra, query languages

1 Introduction

We present a new formalism for relational algebra, the FC language, which is based on a novel factorization of relations. The acronym stands for *factorize and combine*. A pure version of this language is equivalent to relational algebra in the sense that semantics preserving translations exist in both directions [4].

Advantages of the new proposal include more concise and elegant expressions for many queries, new possibilities for query analysis, and the ability to include arithmetic and aggregate functions in a natural way.

The FC language is based on one operator, **factor**, which takes any number of relations as arguments and returns a single relation as result. It is related to the **group_by** operator [2], though more general.

A **factor** expression is evaluated in three steps: The first step is to factorize the relational arguments. The second step is to perform simple computations on the smaller components obtained hereby. The third and final step is to combine the individual results from step two.

The computation in step two is specified by a small core language for manipulating tuples and atomic values. The combination in step three is always the union of the results from step two.

We demonstrate how all standard relational operators, and others, can be translated into the FC language. The translation from FC to relational algebra is more difficult; we refer the reader to [4] for the definition of this translation and proofs of correctness.

2 Factorizations

Recall that a *tuple* is a finite partial function from attribute names to atoms. A *relation* $R(r)$ is a finite set of tuples r, over a common domain R which is also referred to as the *schema* of the relation. As usual, we overload notation and let r refer to the relation $R(r)$.

A 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 $X \subseteq \bigcap R_j$ a set of attribute names. The *factorization* of the r_j's *on* X consists of

- a sequence of component tuples $\phi_1, \ldots, \phi_m$ with common domain X
- for each $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$, a component relation Θ_{ij} with schema $R_j \backslash X$

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 the only tuple of which is ϕ_i, $+$ is interpreted as union, and $\times$ as Cartesian product of relations. These n equations can also be depicted as the following matrix equation

$$(\{\phi_1\}, \{\phi_2\}, \ldots, \{\phi_m\}) \begin{pmatrix} \Theta_{11} & \Theta_{12} & \cdots & \Theta_{1n} \\ \Theta_{21} & \Theta_{22} & \cdots & \Theta_{2n} \\ \vdots & \vdots & & \vdots \\ \Theta_{m1} & \Theta_{m2} & \cdots & \Theta_{mn} \end{pmatrix} = (r_1, r_2, \ldots, r_n)$$

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

3) no row of the (Θ_{ij}) matrix has all "zeroes", i.e., $\forall i\ \exists j :\ \Theta_{ij} \neq \emptyset$

Proposition 2.2 A factorization always exists and is unique up to reordering of the ϕ_i sequence. □

Example: Let r_1 and r_2 be the two relations

A	B
a_1	b_1
a_2	b_2
a_3	b_3
a_4	b_4

and

B	C	D
b_2	c_1	d_1
b_2	c_2	d_2
b_4	c_3	d_3
b_5	c_4	d_4

The factorization of r_1, r_2 on B is

$$\left(\begin{array}{|c|}\hline B \\ \hline b_1 \\ \hline \end{array}\ , \begin{array}{|c|}\hline B \\ \hline b_2 \\ \hline \end{array}\ , \begin{array}{|c|}\hline B \\ \hline b_3 \\ \hline \end{array}\ , \begin{array}{|c|}\hline B \\ \hline b_4 \\ \hline \end{array}\ , \begin{array}{|c|}\hline B \\ \hline b_5 \\ \hline \end{array} \right) \left(\begin{array}{|c|}\hline A \\ \hline a_1 \\ \hline \end{array}\ \begin{array}{|c|c|}\hline C & D \\ \hline \end{array} \right.$$

$$\begin{array}{|c|}\hline A \\ \hline a_2 \\ \hline \end{array}\ \begin{array}{|c|c|}\hline C & D \\ \hline c_1 & d_1 \\ \hline c_2 & d_2 \\ \hline \end{array}$$

$$\left. \begin{array}{|c|}\hline A \\ \hline a_3 \\ \hline \end{array}\ \begin{array}{|c|c|}\hline C & D \\ \hline \end{array} \right) = (r_1, r_2)$$

$$\begin{array}{|c|}\hline A \\ \hline a_4 \\ \hline \end{array}\ \begin{array}{|c|c|}\hline C & D \\ \hline c_3 & d_3 \\ \hline \end{array}$$

$$\begin{array}{|c|}\hline A \\ \hline \end{array}\ \begin{array}{|c|c|}\hline C & D \\ \hline c_4 & d_4 \\ \hline \end{array}$$

3 The FC Language

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

e ::=	α	atom expressions
	$\mathtt{[}A\mathtt{:}e\mathtt{]}$ | $\mathtt{[]}$	tuple formations
|	$e_1\ e_2$	tuple perturbations
|	$e\,.\,A$	tuple inspections
|	$e \backslash A$	tuple restrictions
|	$\mathbf{0} \mid \mathbf{1}$	relation constants
|	$\{e_1, \ldots, e_k\}$	relation formations
|	$b\mathtt{?}e$	guards
|	$f\mathtt{(}e\mathtt{)}$	homomorphisms

We also provide two operations on relations

|	$e_1 \times e_2$	Cartesian products
|	**factor** ... **on** ... **do** ...	factorization operations

The atom expressions are left unspecified, but are intended to be entirely standard; certainly, they will include the booleans. They could also include integers as well as arithmetic on the integers.

A tuple formation $\mathtt{[}A\mathtt{:}e\mathtt{]}$ denotes the function which is undefined on all values except A where its value is e. If e_i denotes the function f_i, $i = 1, 2$, then the tuple perturbation $e_1\ e_2$ denotes the function which is equal to f_2 whenever f_2 is defined and otherwise equal to f_1. If e denotes f, then the tuple inspection $e\,.\,A$ denotes $f(A)$. If e denotes f, then the tuple restriction $e \backslash A$ denotes the function which is equal to f except that it is undefined on A.

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

In the guard expression $b\mathtt{?}e$, the expression b denotes a boolean and e denotes a relation. If b is true, then the result is e; otherwise, the result is $E(\emptyset)$—the empty relation with the schema of e.

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 **and**, **or**, **min**, and **max**.

4 The Factor Operator

The *syntax* of the **factor** operator is

$$\textbf{factor}\ r_1, r_2, \ldots, r_n\ \textbf{on}\ A_1, A_2, \ldots, A_k\ \textbf{do}\ e$$

where $n \geq 1$, the r_j's are relations, $k \geq 0$, $\{A_1, A_2, \ldots, A_k\} \subseteq \bigcap R_j$ is a set of attribute names, and e is an *extended* expression denoting a relation. An extended expression allows the following *extra* constructs (in addition to the constructs from the core language):

$$e \ ::= \quad \textbf{tup} \mid \textbf{rel}(j) \qquad \text{factorization components}$$

We allow a variation: if one merely writes **factor** $r_1, r_2, \ldots, r_n$ **do** e, then the factorization is performed on $\bigcap R_j$, i.e., on 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 ϕ_i and $(\Theta_{i1}, \Theta_{i2}, \ldots, \Theta_{in})$, the expression e is evaluated in an environment where $\textbf{tup} = \phi_i$ and for each $1 \leq j \leq n$, $\textbf{rel}(j) = \Theta_{ij}$. 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 (set-)operators, union and Cartesian product. In between, one can modify the components.

As a trivial example, observe that r_j equals

$$\textbf{factor}\ r_1, r_2, \ldots, r_n\ \textbf{on}\ A_1, A_2, \ldots, A_k\ \textbf{do}\ \{\textbf{tup}\} \times \textbf{rel}(j)$$

for any legal choice of A_i's.

Proposition 4.1 The **factor** operation 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), and 2) the result is independent of the ordering of the ϕ_i's. □

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 **tup**.A. Also, we shall write **rel** rather than **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

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

can be computed as

A
a_1

$\times$

C	D

$+$

A
a_2

$\times$

C	D
c_1	d_1
c_2	d_2

$+$

A
a_3

$\times$

C	D

$+$

A
a_4

$\times$

C	D
c_3	d_3

$+$

A

$\times$

C	D
c_4	d_4

which equals

A	C	D
a_2	c_1	d_1
a_2	c_2	d_2
a_4	c_3	d_3

5 Relational Operators

We have chosen to present FC in a style which is more like a language than like standard relational algebra notation. We also use verbose notation for the standard relational operators.

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

$$\textbf{factor } r \textbf{ on } A_1, \ldots, A_k \textbf{ do } e$$

This implies, of course, that all the A_i's are attribute names of r. The standard unary relational operators can be translated as follows:

$$\textbf{project } r \textbf{ on } A_1, \ldots, A_k \equiv \textbf{factor } r \textbf{ on } A_1, \ldots, A_k \textbf{ do } \{\textbf{tup}\}$$

select r **where** $b \equiv$ **factor** r **do** b?{**tup**}

rename r **by** $A_1 \leftarrow A_2 \equiv$ **factor** r **do** {**tup**\$A_1$[$A_2$:$A_1$]}

We can also define the translation of the following two nonstandard operators [3, 2]:

extend r **by** A:=$e \equiv$ **factor** r **do** {**tup**[A:e]}

group r **by** $A_1, \ldots, A_k$ **creating** A:=f() $\equiv$
factor r **on** $A_1, \ldots, A_k$ **do** {**tup**[A:f(**rel**)]}

It turns out that if the core language contains neither operations on atom expressions (like arithmetic on the integers) nor homomorphisms, then FC and relational algebra are equivalent. If operations on atom expressions are added to FC, then this is equivalent to relational algebra with **extend** (based on the same operations). If homomorphisms are added to FC, then this is equivalent to relational algebra with **group_by** (and the same homomorphisms). See [4] for further details.

Many combinations of ordinary operators can conveniently be expressed by a single **factor** expression. Consider as an example the following expression where r is a relation with schema $\{A_1, A_2, A_3, A_4, B_1, B_2\}$:

project
  extend
    select r **where** B_1>B_2
  by B:=B_1+B_2
over $A_1, A_2, A_3, A_4, B_1, B$

Using **factor** we can write:

factor r **do** B_1>B_2? {**tup**[B: B_1+B_2] \ B_2}

Two points are noteworthy in connection with this example. Firstly, the **factor** expression does not need to know the incidental attributes A_1, A_2, A_3, A_4.

Secondly, the computation is clearly one that should be performed on each tuple individually. This is evident in the **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 **factor** translation is not only shorter, but also considerably easier to program. In a naive implementation, it would also automatically be more efficient.

A short discussion of optimization is appropriate in connection with this example. The tuple language of FC lead us to consider new methods of query analysis. The expression above is one out of a large class of expressions which can be determined to be injective, in the sense that the tuple language expression, viewed as a function from tuples to tuples, is injective. This implies that no duplicates are produced when the query is evaluated. A linear-time algorithm to decide membership of this class is presented in [5]. The results can to some extent be inherited by the binary queries.

Though these results are inspired by the tuple language of FC, they can be applied equally well to languages like SQL and the earlier SEQUEL [1]. Instead of analyzing an SQL query directly, we analyze an equivalent FC query.

We now turn our attention to the usual binary operators. Apart from the standard union operator, we can also get

$$\textbf{union } r_1 \textbf{ and } r_2 \equiv \textbf{factor } r_1, r_2 \textbf{ do } \{\textbf{tup}\}$$

which is an extension: if the two relations have different schemas, then this expression produces the union of the projections over the common attributes names.

Intersection is straightforward:

$$\textbf{intersect } r_1 \textbf{ and } r_2 \equiv \textbf{factor } r_1, r_2 \textbf{ do } \textbf{rel}(1) = \textbf{rel}(2)? \{\textbf{tup}\}$$

Notice that if r_1 and r_2 have the same schema, then **rel**(1) and **rel**(2) equal either **0** or **1**. Thus the guard can be evaluated efficiently.

Of course, another way to obtain the intersection is as a special case of the **join** operator. In general **join** looks as follows:

$$\textbf{join } r_1 \textbf{ and } r_2 \equiv \textbf{factor } r_1, r_2 \textbf{ do rel}(1) \times \{\textbf{tup}\} \times \textbf{rel}(2)$$

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:

$$\textbf{difference } r_1 \textbf{ and } r_2 \equiv \textbf{factor } r_1, r_2 \textbf{ do rel}(2) = \textbf{0?}\, \{\textbf{tup}\}$$

As before, this expression is very easy to understand: we take the **tup**'s that do not belong to r_2.

Consider a relation in which the attributes A, B, C constitute a key. Suppose we have two versions of what is intended to be the same relation. We can obtain the key values for which the information in the two versions disagree:

$$\textbf{factor } r_1, r_2 \textbf{ on } A, B, C \textbf{ do rel}(1) \neq \textbf{rel}(2)\textbf{?}\, \{\textbf{tup}\}$$

This is almost a literal translation of: if the information is inconsistent, then include the key value.

Finally, we present an example of a two-level **factor**. The **divide** operator is defined as

$$r_1/r_2 = \max\{r \mid r \times r_2 \subseteq r_1\}$$

where $R_2 \subseteq R_1$. It is usually quite complicated to derive; however, we can write it as:

$$\textbf{factor } r_1, r_2 \textbf{ do factor rel}(1) \textbf{ do } \{\textbf{tup}\} \times r_2 \subseteq r_1\textbf{?}\, \{\textbf{tup}\}$$

which closely follows the definition. Together the two **factor**s provide the $R_1 \setminus R_2$ part of the 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 **divide** is:

difference
    project r_1 **over** $A_1, A_2, \ldots, A_k$
and
    project
        difference
            join
                project r_1 **over** $A_1, A_2, \ldots, A_k$
            and
                r_2
        and
            r_1
    over $A_1, A_2, \ldots, A_k$

This is not very intuitive; furthermore, one needs explicit knowledge of the set $R_1 \backslash R_2$, i.e., the A_i's. In [3] **divide** is derived from two **group_by**'s, but it involves renamings and projections, and becomes increasingly complex with the size of k.

6 Efficiency

The FC language can be implemented 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 an expression containing exactly one **factor** 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 the unary **project** can be computed in time $O(T \log(T))$.

A unary expression **factor** r **do** e can be evaluated in linear time when e is injective, since no sorting is then required. This property can often be determined statically; an optimal, linear-time algorithm is presented in [5]. Since **select**, **extend**, and **rename** yield such expressions, the **factor** versions of all standard relational operators preserve the complexity of the originals. As discussed in the previous section and in [5], we can sometimes even do better, since combinations of standard operators can be expressed as a single **factor** expression, for which we may detect an injectivity that was

disguised in the original. The results in [5] are fully applicable to standard languages like SQL.

References

[1] M. M. Astrahan and D. D. Chamberlin. Implementation of a Structured English Query Language. *Communications of the ACM*, 18(10):580–588, 1975.

[2] P. M. D. Gray. The GROUP_BY Operation in Relational Algebra. In S. M. Deen and P. Hammersley, editors, *Databases*, pages 84–98. Pentech Press Limited, 1981.

[3] P. M. D. Gray. *Logic, Algebra and Databases*. Ellis Horwood Limited, 1984.

[4] K. S. Larsen. *Equivalence of FC and Relational Algebra*. In preparation.

[5] K. S. Larsen and M. I. Schwartzbach. *Optimal Detection of Query Injectivity*. Computer Science Department, Aarhus University, 1990.