An Intermediate Language for Relational Algebra

Kim S. Larsen[1]
Michael I. Schwartzbach[2]
Erik M. Schmidt[3]

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

Abstract

We present an intermediate language for relational algebra. This **factor** language is based on a term-algebra as opposed to standard relational algebra which is, primarily, a combinator-algebra. We demonstrate how queries that use standard operators can be translated into **factor** queries. The translation is very simple and intuitive. In fact, code in the intermediate language is usually shorter than the source code and can be evaluated more efficiently. The **factor** queries *never* get longer or less efficient. In addition, more query optimization analysis than usual can be performed on this intermediate language. Finally, the **factor** language could often be used directly as a query language, as it is very expressive and easy to use.

1 Introduction

We present an intermediate language for relational algebra; the **factor** language. This language is very simple, but strong enough to implement all the standard relational operators that are described in e.g. [**?**]. The language can without problems be extended in many directions in order to support special constructs in the source language under consideration. Here, we will mainly consider a source language containing only the standard operators. However, even with this simple language, the intermediate code is usually shorter and can sometimes be

[1]Internet address: kslarsen@daimi.dk
[2]Internet address: mis@daimi.dk
[3]Internet address: emschmidt@daimi.dk

evaluated more efficiently than the source code. The intermediate code is never longer or less efficient.

The intermediate language can be viewed as a new relational operator, **factor**, which is why it can be used directly in a query language instead of (or together with) the usual operators. When used like this, many computations on relations seem to be expressible in a more direct and intuitive fashion.

The **factor** language is inspired by the **group_by** operator [**?**], but it is far more general and fundamental in its nature.

A **factor** expression takes any number of relations as arguments together with some additional information. It is based on a unique *factorization* of relations.

We propose to evaluate **factor** queries in three steps.

The first step is to decompose the relational 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 for manipulating tuples and atomic values, as 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** language. In sections 5, 6, and 7, we demonstrate how all standard relational operators, and many more, can be translated into the **factor** language.

In section 8, we observe that **factor** queries 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. The **factor** language has been implemented[**?**] and tests show a very good performance. 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 $\{a_1, \ldots, a_k\} \subseteq \bigcap \sigma(R_j)$ 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 Θ_{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 ϕ_i, $+$ is interpreted as union, and $\times$ as Cartesian product of relations. These n equations can be concisely expressed as the following matrix equation

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

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. $\forall i \; \exists j : \; \tau(\Theta_{ij}) \neq \emptyset$

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

Proof We can find $\{\phi_1, \ldots, \phi_m\}$ as $\bigcup R_j \downarrow a_1, \ldots, a_k$, where $\downarrow$ is projection. Now, Θ_{ij} is found by selecting from R_j where $a_1, \ldots, a_k$ equals ϕ_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 ϕ_i belongs to $\bigcup R_j \downarrow a_1, \ldots, a_k$ then it must belong to some $R_j \downarrow a_1, \ldots, a_k$ and, hence, this particular Θ_{ij} must be non-zero. This demonstrates existence. For uniqueness, we observe that for any factorization the matrix equation implies $\bigcup R_j \downarrow a_1, \ldots, a_k \subseteq \{\phi_1, \ldots, \phi_m\}$. Since every row has a non-zero element we get the other inclusion, too. As the

Θ_{ij}'s are determined uniquely from the ϕ_i's, the factorization is unique up to a reordering of the ϕ_i's. □

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

$$\left(\begin{array}{|c|}\hline B\\\hline b_1\\\hline\end{array}, \begin{array}{|c|}\hline B\\\hline b_2\\\hline\end{array}, \begin{array}{|c|}\hline B\\\hline b_3\\\hline\end{array}, \begin{array}{|c|}\hline B\\\hline b_4\\\hline\end{array}, \begin{array}{|c|}\hline B\\\hline b_5\\\hline\end{array}\right)\left(\begin{array}{cc} \begin{array}{|c|}\hline A\\\hline a_1\\\hline\end{array} & \begin{array}{|c|c|}\hline C & D\\\hline\end{array} \\ \begin{array}{|c|}\hline A\\\hline a_2\\\hline\end{array} & \begin{array}{|c|c|}\hline C & D\\\hline c_1 & d_1\\\hline c_2 & d_2\\\hline\end{array} \\ \begin{array}{|c|}\hline A\\\hline a_3\\\hline\end{array} & \begin{array}{|c|c|}\hline C & D\\\hline\end{array} \\ \begin{array}{|c|}\hline A\\\hline a_4\\\hline\end{array} & \begin{array}{|c|c|}\hline C & D\\\hline c_3 & d_3\\\hline\end{array} \\ \begin{array}{|c|}\hline A\\\hline\end{array} & \begin{array}{|c|c|}\hline C & D\\\hline c_4 & d_4\\\hline\end{array} \end{array}\right) = (R_1, R_2)$$

3 The Implementation Language

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

e ::=	α	atom expressions
	$\mid$ `[`a`:`e`]` $\mid$ `[]`	tuple formations
	$\mid$ $e_1\ e_2$	tuple perturbations
	$\mid$ e`.`a	tuple inspections
	$\mid$ $e \setminus a$	tuple restrictions
	$\mid$ $\mathbf{0} \mid \mathbf{1}$	relation constants
	$\mid$ $\{e_1, \ldots, e_k\}$	relation formations
	$\mid$ b`?`e	gates
	$\mid$ f`(`e`)`	homomorphisms

We also provide two operations on relations

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

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. $\mathbf{0} = (\emptyset, \emptyset)$ and $\mathbf{1} = (\emptyset, \{\,[\,]\,\})$. 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 Operation

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

$$\begin{array}{l} \textbf{factor } R_1, R_2, \ldots, R_n \\ \textbf{on } a_1, a_2, \ldots, a_k \\ \textbf{do } e \end{array}$$

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

e	$::=$	**tup** $\mid$ **rel**(j)	factorization components

We allow the following variation: If one merely writes

$$\textbf{factor } R_1, R_2, \ldots, R_n \textbf{ do } e$$

then the factorization is performed on $\bigcap \sigma(R_j)$, 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 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 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

$$\textbf{factor}\ R_1, R_2, \ldots, R_n\ \textbf{on}\ a_1, a_2, \ldots, a_k\ \textbf{do}\ \{\textbf{tup}\} \times \textbf{rel}(j)$$

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).

2) the result is independent of the ordering of the ϕ_i's.

Proof

1) the schemas of ϕ_i and Θ_{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 ϕ_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 $\textbf{tup}.a$. Also, we shall write **rel** rather than $\textbf{rel}(1)$ 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

$$\begin{array}{|c|}\hline A\\\hline a_1\\\hline\end{array} \times \begin{array}{|c|c|}\hline C & D\\\hline\end{array} + \begin{array}{|c|}\hline A\\\hline a_2\\\hline\end{array} \times \begin{array}{|c|c|}\hline C & D\\\hline c_1 & d_1\\\hline c_2 & d_2\\\hline\end{array} + \begin{array}{|c|}\hline A\\\hline a_3\\\hline\end{array} \times \begin{array}{|c|c|}\hline C & D\\\hline\end{array} + \begin{array}{|c|}\hline A\\\hline a_4\\\hline\end{array} \times \begin{array}{|c|c|}\hline C & D\\\hline c_3 & d_3\\\hline\end{array} + \begin{array}{|c|}\hline A\\\hline\end{array} \times \begin{array}{|c|c|}\hline C & D\\\hline c_4 & d_4\\\hline\end{array}$$

which equals

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

5 Unary Operators

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

$$\textbf{factor } R \textbf{ on } a_1, \ldots, a_k \textbf{ do } e$$

where all the a_i's are attribute names of R. The standard unary relational operators can be translated as follows.

$$\begin{aligned}&\textbf{project } R \textbf{ on } a_1, \ldots, a_k \equiv\\ &\qquad \textbf{factor } R \textbf{ on } a_1, \ldots, a_k \textbf{ do } \{\textbf{tup}\}\end{aligned}$$

$$\begin{aligned}&\textbf{select } R \textbf{ where } b \equiv\\ &\qquad \textbf{factor } R \textbf{ do } b?\{\textbf{tup}\}\end{aligned}$$

$$\begin{aligned}&\textbf{rename } R \textbf{ by } a_1 \rightarrow a_2 \equiv\\ &\qquad \textbf{factor } R \textbf{ do } \{\textbf{tup}\backslash a_1 [a_2 : a_1]\}\end{aligned}$$

We can also define the translation of the following two non-standard operators [?, ?]

$$\begin{aligned}&\textbf{extend } R \textbf{ by } a := e \equiv\\ &\qquad \textbf{factor } R \textbf{ do } \{\textbf{tup}[a : e]\}\end{aligned}$$

$$\begin{aligned}&\textbf{group } R \textbf{ by } a_1, \ldots, a_k \textbf{ creating } a := f() \equiv\\ &\qquad \textbf{factor } R \textbf{ on } a_1, \ldots, a_k \textbf{ do } \{\textbf{tup}[a : f(\textbf{rel})]\}\end{aligned}$$

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

```
reduce $R$ by $a_1, \ldots, a_k \equiv$
    factor $R$ on $a_1, \ldots, a_k$ do rel
```

Another example is an **update** operator which works like **extend**, except that it assigns to an existing attribute

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

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

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 $\{a, b, c, d, x, y\}$

```
project
   extend
      select $R$ where $x>y$
   by $z := x+y$
over $a, b, c, d, x, z$
```

Using **factor** we can write

```
factor $R$ do $x>y$? $\{$tup[$z$: $x+y$] $\backslash y\}$
```

Two points are noteworthy in connection with this example. Firstly, the **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 **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 (and more efficient) but also considerably easier to program. This would be an argument for including **factor** in the query language as an operator.

6 Binary Operators

The usual binary operators, as well, appear as simple binary **factor** expressions.

We might be pleased with the standard union operator, but we could also get

$$\begin{aligned}&\textbf{union } R_1 \textbf{ and } R_2 \equiv\\ &\qquad \textbf{factor } R_1, R_2 \textbf{ do } \{\textbf{tup}\}\end{aligned}$$

which is 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 **rel**(1) equals **rel**(2) then they both equal **1**

$$\begin{aligned}&\textbf{intersect } R_1 \textbf{ and } R_2 \equiv\\ &\qquad \textbf{factor } R_1, R_2 \textbf{ do } \textbf{rel}(1) = \textbf{rel}(2)?\, \{\textbf{tup}\}\end{aligned}$$

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

$$\begin{aligned}&\textbf{join } R_1 \textbf{ and } R_2 \equiv\\ &\qquad \textbf{factor } R_1, R_2 \textbf{ do } \textbf{rel}(1) \times \{\textbf{tup}\} \times \textbf{rel}(2)\end{aligned}$$

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

$$\begin{aligned}&\textbf{difference } R_1 \textbf{ and } R_2 \equiv\\ &\qquad \textbf{factor } R_1, R_2 \textbf{ do } \textbf{rel}(2) = \mathbf{0}?\, \{\textbf{tup}\}\end{aligned}$$

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

$$\begin{aligned}&\textbf{combine } R_1 \textbf{ and } R_2 \equiv\\ &\qquad \textbf{factor } R_1, R_2 \textbf{ do } \textbf{rel}(1) \times \textbf{rel}(2)\end{aligned}$$

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

$$\begin{aligned}&\textbf{symdiff } R_1 \textbf{ and } R_2 \equiv\\ &\qquad \textbf{factor } R_1, R_2 \textbf{ do } \textbf{rel}(1) \neq \textbf{rel}(2)?\, \{\textbf{tup}\}\end{aligned}$$

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

```
check $R_1$ and $R_2$ $\equiv$
    factor $R_1, R_2$ on $a, b, c$ do rel$(1) \neq$ rel$(2)$? {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

```
project
    difference
        union $R_1$ and $R_2$
    and
        intersect $R_1$ and $R_2$
over $a, b, c$
```

Finally, we present an example of a two-level **factor**. The **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 it as

```
divide $R_1$ and $R_2$ $\equiv$
    factor $R_1, R_2$ do
        factor rel$(1)$ do
            {tup} $\times R_2 \subseteq R_1$? {tup}
```

which closely follows the definition. Together the two **factor**s 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 **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 [?] **divide** is derived from two **group_by**'s, but it involves renamings and projections, and it gets increasingly 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.

$$\textbf{union } R_1 \textbf{ and } R_2 \textbf{ and } \ldots \textbf{ and } R_n \equiv \\ \quad \textbf{factor } R_1, R_2, \ldots, R_n \textbf{ do } \{\textbf{tup}\}$$

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

$$\textbf{safejoin } R_1 \textbf{ and } R_2 \textbf{ using } R_3 \equiv \\ \quad \textbf{factor } R_1, R_2, R_3 \textbf{ do } \{\textbf{tup}\} \times \textbf{rel}(1) \times \textbf{rel}(2) \times \textbf{rel}(3)$$

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

8 Efficiency

The **factor** 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 $O(T\log(T))$.

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 expression can be evaluated 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** expression 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. All of this is treated in great detail in [**?**], where an optimal, linear-time decision algorithm is developed.

We observe that the expressions in **select** and in (legal) **extend** and **rename** are injective and that, consequently, these operators will run in time $O(T)$. 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** expression. 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 implement few operators on relations. This means, of course, that the implementation efforts can be concentrated on making these operators very efficient.

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 ϕ_i's and Θ_{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** language can conveniently express all standard relational operators, and many more, without any loss of efficiency. Hence, **factor** could be an interesting alternative implementation of relational database languages. This is especially attractive as very few operations on relations have to be implemented, and because more and new techniques for query optimization analysis can be applied. Furthermore, if a **factor** implementation is used, **factor** could also be introduced as an operator in the language, so that users could benefit from its expressiveness and often more intuitive query style.

In a concrete language proposal [**?**], we have combined **factor** with fully recursive polymorphic higher-order functions, which yields a very powerful tool. The language is not yet fully implemented, but the evaluation machine for **factor** expressions is, and early tests show a very good performance.

The **factor** language makes more and new techniques for query optimization analysis possible. This is demonstrated in [**?**] where injectivity analysis are performed. We have by an example demonstrated how unary queries can be "collapsed" to one **factor** expression. This will be treated formally in a later report. We are also interested in developing a full calculus of **factor** expressions to support further query optimizations.

We are working on a logical characterization of **factor** to determine its computational strength. We conjecture that **factor** is stronger than the collection of usual relational operators, but not strong enough to calculate e.g. transitive closure.

References