High Level Efficiency in Database Languages

Kim S. Larsen

Computer Science Department, Aarhus University
Ny Munkegade, 8000 Aarhus C, Denmark

March 1993

Preface

The subject of this Ph.D. thesis is the design and implementation of database languages. The thesis consists of five articles [16, 57, 59, 60, 61] and this survey paper. In [61], a new query language design is proposed. The expressive power of the language is determined in [57] and all reasonable extensions are considered. In [59, 60], we focus on the optimization issue of avoiding unnecessary sorting of relations. The results in these papers are directly applicable to any algebra-based query language. In addition to the query language part, a database system also has to offer update facilities. The theory of standard tuple based updates is quite well developed in the sequential case. In [16], we discuss a new concurrent implementation of balanced search trees for that purpose.

This survey paper describes the results in the papers which form the thesis, and relates these results to each other and to the area in a broader sense than is customary in the introductions of individual papers. The paper is intended to be read in combination with the papers on which it is based.

The last section of this survey paper is a summary in Danish.

Acknowledgements

The work on this Ph.D. thesis was initiated in the late summer of 1989. On September 16, 1992, the thesis was officially submitted, and the thesis defense took place February 26, 1993.

First, I would like to thank my advisor, Erik Meineche Schmidt, for his advice and support, and the thesis committee consisting of Erik Meineche Schmidt (Aarhus), Mogens Nielsen (Aarhus), and Jens Clausen (Copenhagen) for careful reading of the thesis and for asking interesting questions at the defense. Mogens Nielsen deserves additional thanks for wakening my interest in research while advising me on my master's thesis.

At the Computer Science Department in Aarhus, there are many other people who have helped me in various ways during this period; some offered friendship, some helped with practical matters, and some contributed to interesting discussions. I would like to thank all of them.

I spent the fall of 1990 and the spring of 1991 at the University of Toronto visiting the group around Alberto Mendelzon. It was very helpful to spend those two semesters in an active database research environment, and I would like to thank everybody there who did their best to make me feel welcome.

I would also like to thank my wife, Joan Boyar, for inspiration, for interesting and helpful discussions, and most of all for putting up with me during this often stressful period.

Finally, my parents Lisbeth and Karsten Larsen have contributed greatly to the very existence of this thesis by instilling in me their belief in the importance of education. They have been very supportive of my educational and career choices and very helpful with matters less directly connected to this thesis. For all of this, I am very grateful.

1 Introduction

A database system consists of a large number of components which interact to create the best possible environment for storage and retrieval of data. It varies from system to system to what extent these components are integrated and exactly what borderlines or interfaces the designer has chosen to offer and/or enforce on the system. But usually components dealing with the following subjects are present: query languages, update facilities, transaction management, access control, failure recovery procedures, versioning mechanisms, concurrency, etc. Of these components, the query language and the update facilities are the high level or user-end components and these are the ones we focus on here.

Query languages have been developed guided by quite different strategies. Early on, network or hierarchical models were used. Later, the relational model was the basis for implementations. Newer approaches include semantics-oriented models [17], logic-based models [86], object-oriented models [11], and graph-based models [23]. However, since a number of papers in the late sixties led up to Codd's break-through paper in 1970 [21], the relational model has had a very central place in the area; both as a model in its own right and as a basis for the newer models. Only time will show if, or to what extent, newer models will be serious competitors to the relational one. An interesting description of the development of the area can be found in [85]. It is worth noting that, for a large part, the success of the relational model seems to be due to its simplicity and tractability in a mathematical framework. Via papers such as [8, 15], a rich theory of relations developed, which led to efficient implementations through design and optimization. In [57, 61], we deal with the design aspect in connection with aggregation in particular.

The ability to store and retrieve large amounts of data is what characterizes database systems and distinguishes them from other languages or systems. This means that it is often worth while to optimize queries and updates before or during the actual evaluation. Optimization can be carried out on many levels; from the highest levels of transforming queries into equivalent more efficient ones, to the lowest levels of organizing data optimally on the physical storage medium such that it can be fetched rapidly under various conditions. The paper [45] contains an excellent survey on optimization of

queries. In [59, 60], we focus on sorting, or rather, on how to avoid sorting. The results we obtain are based directly and solely on the analysis of queries. The results in [60], in particular, were inspired by the design papers [57, 61], but in both of [59, 60], we present the results in a general and standard framework since they can be applied to a great variety of systems.

The area of updating has been neglected for quite a while, but gradually the simplified view of updates being nothing more than a query language plus an assignment [1] is changing. However, whatever direction update languages are taking, it seems likely that the core of the matter will still be the ability to deal with insertion and deletion of tuples in an efficient way. In [16], we look at sorted organizations of tuples in the form of trees. In the sequential case, there is a well established theory, but results do not carry over immediately to the concurrent (but centralized) environment. Recently, in [72], Pugh conjectured:

> It might be possible to design concurrent balanced tree algorithms that allowed $O(n)$ busy writers with high efficiency, but the complexity of such algorithms probably would make their implementation prohibitive.

We believe that in [16], we prove that conjecture false. There is also a connection between [16] and [59] in that the clustering of data naturally present in a sorted structure implies that retrieval in sorted order is faster than other choices. This can be exploited in the algorithms of [59].

In [57, 61], we deal with design of query languages with aggregation, in particular, with emphasis on expressive flexibility (in [45], this is referred to as "user optimization"). In [59, 60], we look at optimization based on direct analysis of queries. In [16], the focus is on efficient design of the highest level of update organization. Thus, the aim of all five paper is to optimize performance via design and analysis of the high level parts of database systems. We believe, therefore, that an appropriate title for this collection of work is the chosen one: High Level Efficiency in Database Languages.

We have tried to include the relevant references each time we have introduced a new concept. If, despite our effort, some concepts have been discussed and have not been properly referenced, then we refer the reader to the excellent textbooks of Date [27], Kanellakis [48], Maier [64], and Ullman [86, 87].

Maier's book is the oldest and does not cover the newer areas, but it contains interesting results not included in the others. The older textbooks of Date [25, 26] and Ullman [84] cover the included material well, but all important results also appear in the textbooks mentioned above.

2 User Optimization

In [45], user optimization is defined as the subject dealing with "functional capabilities and usability of the query language". So, the concern here is primarily the ease with which users can formulate complex queries. In this section, the main focus is on aggregation, but we start with a brief summary of our view on the differences between relational algebra and relational calculus seen from a user's perspective.

With the exception of the natural join (and variants like semi-join [14], for example), the relational algebra operators take one or two arguments and produce a new relation which is related to the argument(s) in a very simple manner. Here, calculus queries are more complicated because of the more indirect formulation using existential and universal quantification. However, when the relevant tuples have been determined via the quantifiers, it is easy to formulate the result by forming a tuple using the attribute values from the tuples determined through the quantification. The algebra does not have this advantage. For example, unary queries which basically perform a number of simple manipulations on each tuple have to be formulated as a long sequence of operations on relations in the form of projections, selections, extensions [38], renamings, etc.

To some extent, we support the view of Merrett [65] who argues that there is nothing unnatural in trying to combine the algebra and the calculus. In fact, Merrett refers to the distinction and calls it "a historical accident that one branch of logic should be thought of as an algebra and the other called calculus".

The language which we present later in this section could be viewed as a blend of algebra and calculus. The language consists of an advanced algebra operator, which is combined with a tuple language, the purpose of which is to give us the advantages of the calculus.

Before we present the language, we wish to discuss aggregation and grouping. In the database world, aggregation usually refers to the computation of one value from a set (or a multiset) of values. Typical aggregate functions are *sum, count, max,* etc. Often aggregation is connected with some form of grouping, which allows a relation to be partitioned into a number of sets. The aggregate function is then applied to each of these partitions, and a new relation is formed using these individual results.

From early on, aggregation was included as a facility in query languages. It is present in the most famous languages like QUEL [82, 92] (INGRES [81, 82]), SQL [10] (System R [9, 18]), and QBE [91], as well as in less wide-spread languages like ASTRID [36, 37, 38], STRAND [46, 47], etc.

However, as it is also pointed out in [47], it can be quite cumbersome to express an aggregate query in these languages. This is partly due to heavy syntax (especially in QUEL) and partly due to the difficulties for the (inexperienced) user in understanding the concept of aggregation (and grouping in particular). In [46, 47], Johnson proposes a simpler language, STRAND, where syntax is reduced to a minimum and the grouping does not have to be formulated explicitly by the user. Unfortunately, the simplicity is obtained by sacrificing generality. As this approach has not been very successful, we will not describe the method in any detail. Briefly described, the user can only apply aggregate functions to (implicit) groupings over the join of a sequence of relations connected in the underlying entity-relationship model [20].

In [52], Klug develops a theoretical framework for incorporating aggregate functions into relational algebra and calculus and proves his extended algebra and calculus equivalent. For the algebra, expressions like

$$e\langle X, f\rangle$$

are added, where e is a relation expression, X is a list of attribute names (the group-by list), and f is an aggregate function. Here, X determines the partitions. Each maximal set of tuples agreeing on the attributes X constitutes one element of the partition. A more concrete way of looking at it is the following. First, we can form the relation $\pi_X(e)$. Each tuple t in this relations determines a partition $\{t\} \bowtie e$, where $\{t\}$ is the singleton relation containing only the tuple t. The aggregate functions that can be used are actually families of aggregate functions, as they are indexed by attribute

names (actually, Klug uses numbers as in Codd's original proposal, but we like attribute names better). So, for some attribute name $A \in schema(e) \setminus X$, f could be sum_A, $count_A$, etc. Now, each tuple $\{t\}$ in $\pi_X(e)$ is extended with one value calculated by applying the aggregate function to the specified column in $\{t\} \bowtie e$.

Specifying aggregation like this, makes it unnecessary to first project over the relevant attribute name (the A from above), require that duplicates be retained, and then perform the aggregation. This was the approach of QUEL and SQL among others, and as Klug argues, this goes completely outside the set-theoretic definition of the relational model. Also, Klug's calculus only allows safe [22, 31] expressions.

In [70], set-valued attributes are introduced directly on top of the algebra and the calculus of Klug. That paper also contains equivalence results.

In [24], several variants of Datalog [86] and GraphLog [23] are considered and compared. In particular, Consens and Mendelzon consider safe [74, 90] non-recursive Datalog with aggregation, which is equivalent in expressive power to relational algebra with aggregation. In addition to the usual rules, they allow rules of the form

$$p(t_1, \ldots, t_n)[X] \leftarrow s_1, \ldots, s_k$$

where a term t_i is now allowed to be an aggregate function applied to a variable and X is a sequence of variables (the group-by list). The t_i's which are variables should also appear in X. To ensure safety, all the variables in $t_1, \ldots, t_n$ and the variables in X should also appear in $s_1, \ldots, s_k$. The interpretation of the rule is the obvious one generalizing the semi-naive evaluation of Datalog programs [86].

In greater detail, this means the following. Consider a standard rule like

$$p(t_1, \ldots, t_n) \leftarrow s_1, \ldots, s_k$$

Assume that e is the relational expression obtained in the usual way by joining $s_1, \ldots, s_k$ and selecting on equality and inequality. Then the relational expression $\pi_{t_1,\ldots,t_n}(e)$ is usually associated with that rule and the expression is used in the fixed point calculation. In the new aggregate rule, if t_i is an

aggregate function applied to an attribute name, $f(A)$, and remaining t_j's are simply attribute names, then the associated relational expression would be

$$\pi_{t_1,\ldots,t_n}(\mathit{group}\ e\ \mathit{by}\ X\ \mathit{creating}\ t_i := f_A())$$

The fact that Klug's presentation is so simple together with the fact that radical extensions can be made without ruining the algebra/calculus equivalence and the fact that the simplest possible formulation of aggregation in the standard logic-based language, Datalog, gives the same expressive power, strongly supports the point of view that Klug's extension to standard relational algebra is very natural.

We believe, however, that there are some artificial limits both with respect to the grouping and with respect to the use of aggregate functions on the grouping. First, it is not given a priori that grouping should be a unary operation. Second, it is not given a priori that aggregate functions should be applied directly to the grouping components. We give some examples later, which will clarify these objections. As we have argued that Klug's basic approach is the natural one, further extensions should of course have Klug's grouping and aggregation as a special case.

For some applications where the use of aggregation is not central, it might be a good idea to simplify the grouping aspect through restrictions on the language, as it has been done with STRAND as discussed above. However, in applications where the need for aggregation comes up frequently, we believe that the power of grouping and aggregation should not be restricted.

In [61], we have *based* a language called FACTOR on grouping. Although this places an initial burden on the user, we believe that for applications where the concept of grouping is indispensable, we might as well try to increase our benefits by exploiting the grouping mechanism intensively. Before discussing the advantages we obtain, let us stress that this language should not necessarily be used directly as a query language. Only experience with a prototype implementation can give some hints as to what the best variants of this language might be. Such an implementation has been completed very recently [58], but we have not yet had much response from users.

Based on our discussion in this section, we now account for the advantages of the FACTOR language as we see them. We refer the reader to [57, 61] for

a precise definition of the FACTOR language, its semantics, and the simplest examples expressing the standard operators of relational algebra. In the following, we just give a short and informal description of FACTOR. This description is followed by some examples to illustrate the more important aspects of the language. The name FACTOR is motivated in [57].

A FACTOR expression can take any number of arguments. The notation used here is

$$r_1, r_2, \ldots, r_n : \ exp$$

The group-by list is implicitly defined as the intersection of the schemas of the arguments.

In the following, compare with the semantics of the Klug expression giving earlier. A FACTOR expression is evaluated by running through all the tuples of $\bigcup_i \pi_X(r_i)$, where $X = \bigcap_i R_i$, evaluating *exp* each time, and taking the union. So, the implicit union present in Klug's grouping/aggregation is of course also present in FACTOR. In *exp*, we are allowed to use the current tuple from $\bigcup_i \pi_X(r_i)$, called #, and the corresponding relation parts from the n relations, called @(1),...,@(n).

In the most basic FACTOR language described in [61], we only have a small tuple language. A constant tuple can be defined by writing $[A : v]$, where A is an attribute name and v is a value. Tuples can be concatenated and we use the notation $t_1 t_2$ for this, where t_1 and t_2 are tuples. There is also tuple restriction, $t \backslash A$, which denotes the same tuple as t except that the attribute name A and its associated value does not appear in $t \backslash A$. To extract associated values from a tuple, we use the notation $t.A$. The value in the formation of a constant tuple is a constant or an attribute name from # (that is, from $\bigcap_i R_i$). It can also be an expression if we allow computation on domain values.

As values from # are used very frequently, it is natural to allow a more convenient notation for this. In the following, an attribute name, A, used as a *value* will be short for $\#.A$.

A singleton relation can be formed by surrounding a tuple with curly brackets. In addition, we have a gate construct (or a guard). This is denoted $b?\ e$, where b is a boolean expression and e a relational expression. The expression

evaluates to the value of the relational expression provided the boolean expression evaluates to true. Otherwise the value is the empty relation with the schema of the relational expression. The boolean expressions are the same as the ones used in ordinary selections. Finally, we use Cartesian product. Let us immediately remark at this point that using Cartesian product will not make it more expensive to evaluate queries because this operator will typically be used on the small parts of the relations where the natural join is also basically the Cartesian product (the parts which have identical values in the intersection of the schemas).

This is the basic language. In [57], this language is proved equivalent in expressive power to relational algebra. In the same paper, it is shown how the language can be extended significantly and at the same time remain equivalent to relational algebra. For aggregation and computation on domain values, the equivalence only holds, of course, if relational algebra is extended similarly (for instance, using Klug's definition). See [57] for details.

It makes perfect sense and it is sometimes convenient to group on less than the intersection of the schemas of the relational arguments to FACTOR. For notational convenience, we shall use the symbols $|\!+$ and $|\!-$ to indicate a positive or negative list of attribute names relative to this intersection of schemas. Additionally, we shall allow @ to be used for @(1).

We stated earlier that it is not given a priori that grouping should be a unary operation. This is illustrated by the following example. Consider the relations *Employees* with schema {*Name, Dept, Salary*} and *Offices* with schema {*Location, Dept, Size*}. The attribute *Name* is a key for *Employees*, and *Location* is a key for *Offices*. The relation *Employees* contains information about the staff of some institution. For each person, the person's department and salary is listed. The relation *Offices* contains a listing of all the offices in the institution: their exact location, which department uses the offices, and how many people they can hold. In connection with deciding which departments are most needy of office space, we are interested in calculating the ratio of office space per employee. We need to aggregate over two arguments at the same time.

Employees, Offices: {#[*Ratio: sum*(@(**2**), *Size*) *count*(@(**1**), *Name*)]}

It would be quite awkward to formulate this in standard relational algebra

with some form of a unary grouping mechanism and an extension operation with arithmetic. There are two main reasons why this is so easy to formulate in FACTOR. First, grouping in FACTOR is a general operation, i.e., it is not restricted to a single relational argument. Thus, the grouping components in the two relations *Employees* and *Office* associated with a certain *Dept* value can be accessed at the same time. Second, the strong tuple language ensures that there is an effective way of combining the aggregate values $sum(@(\mathbf{2}), Size)$ and $count(@(\mathbf{1}), Name)$.

Also, notice how easily computations on domain values fit into the tuple part. This is also high-lighted by the following query. Let the schema of r be $R = \{A_1, A_2, A_3, A_4, B_1, B_2\}$. In relational algebra, a unary query with a domain value computation would typically look something like

$$\pi_{A_1,A_2,A_3,A_4,B_1,B}\ (extend\ \sigma_{B_1>B_2}(r)\ with\ B := B_1 + B_2)$$

In the FACTOR language, we can write

$$r : B_1 > B_2?\ \{\#\backslash B_2[B : B_1 + B_2]\}$$

We cannot argue that this is simpler than the relational algebra formulation as this is clearly a matter of taste. If one is very familiar with the algebra, then the standard algebra query is probably the simplest to understand. However, our opinion is that using the tuple language with an explicit naming of a (generic) tuple from the argument relation makes the formulation of the query easier.

We stated earlier that it is not given a priori that aggregate functions should be applied directly to the grouping components. This is illustrated by the following example.

Assume that $r, R = \{A, B\}$, is a representation of a directed graph. We want to count the number of 2-cycles for each node, i.e., for each node v, we want to count the number of distinct nodes w such that (v, w) and (w, v) are arcs in the graph. In relational algebra with group_by, this can be done by

$$\pi_{A,N}(group\ \sigma_{A=C}(r \bowtie \delta_{A\leftarrow B,B\leftarrow C}(r))\ by\ A,\ C\ creating\ N := count())$$

Avoiding the inefficient natural join, the FACTOR version looks like

$$r, \delta_{A \leftarrow B, B \leftarrow A}(r) \mid\!+ A : \{\#[N : \mathit{count}(@(1) \cap @(2))]\}$$

Once again, it is the blend of algebra and calculus, which makes this query easy to formulate. In addition, it also becomes more efficient. Because the grouping is still an algebra operation, standard algebra operations can be used freely in combination with the basic FACTOR language. The power of the calculus-inspired tuple language is, of course, dependent on the explicit naming of the grouping components, # and the @'s. Only via direct references to the grouping components is it possible to build a flexible and efficient tuple language.

To underline the naturalness of the FACTOR language, we demonstrate how easily we can extend the language to deal with one of the Non First Normal Form (N1NF) models. Various complex object models [2, 13, 42], such as the Verso model [3], can be simulated, but the world of nested relations fits the FACTOR language best. We take a closer look at the nest and unnest operators of [43, 44] (in the their NF^2 model, they refer to these operators as ν_A and μ_A).

Again, grouping is the fundamental concept. In fact, no new notation or constructs are needed. All we need to do is to allow relations as values. We assume the following semantics of *nest* and *unnest*. The operation nest is used as $\mathit{nest}_A(r)$. The schema R of r must contain at least two attribute names, one of which is A. Operationally, $\mathit{nest}_A(r)$ is then the following relation. Each maximal set of tuples from r agreeing on $R \setminus \{A\}$ is replaced by a single tuple. This tuple has the value they all have on $R \setminus \{A\}$, whereas on A, the value is the relation with schema $\{A\}$ containing all the A-values from the set of tuples under consideration.

More formally, the equality $\pi_{R \setminus A}(r) = \pi_{R \setminus A}(\mathit{nest}_A(r))$ holds. Additionally, for each tuple t in $\pi_{R \setminus A}(r)$, $\{t\} \bowtie \mathit{nest}_A(r)$ is a singleton, and the A value of the single tuple it contains is $\pi_A(\{t\} \bowtie r)$.

The operator *unnest* is simply the reverse operation. Formally, this means that $\mathit{unnest}_A(r)$ has schema R, the equality $\pi_{R \setminus A}(r) = \pi_{R \setminus A}(\mathit{unnest}_A(r))$ holds, and each tuple in $\pi_{R \setminus A}(r)$ has A value $\pi_A(\{t\} \bowtie \mathit{unnest}_A(r))$. Thus, the argument r is required to have relational values in the A column.

Now we can write $\mathit{nest}_A(r)$ as

$$r\ |{-}A : \{\#[A : @]\}$$

and $unnest_A(r)$ as

$$r : \{\#\backslash A\} \times A$$

When our relations are extended to include relational values, we will probably be interested in extending our set of aggregate functions as well. For instance, if $R = \{A, B, C\}$ and B contains relational values, then we can allow

$$r\ |{+}A : \{\#[D :\ intersect(@, B)]\}$$

As mentioned earlier, the FACTOR language has been implemented, and so far, a simple version has been described [58].

Finally, let us remark that an implementation of the FACTOR language will not be less efficient than existing implementations of languages of equivalent expressive power. There are some limits to the efficiency which is enforced by the grouping [7], but we claim that similar costs would be present in any equivalent relational algebra or calculus query. In other words, the FACTOR language has the same asymptotic complexity as equivalent languages. Most optimization techniques aimed at obtaining constant speed-up factors can be used equally well in a FACTOR implementation. This is discussed in greater detail for some special cases in the next section and in [59, 60, 61].

3 Query Optimization

Query optimizations range from simple standard techniques, like pushing down selections, to sophisticated expression analysis. A great number of optimization techniques are now entirely standard and are a part of almost any database system. We are thinking of techniques like, as already mentioned, pushing down selections [6, 71], combining sequences of unary operators [41], using multiway sort-merging [5], etc. These techniques were developed very early because they in many ways are obvious and guaranteed to improve the complexity of queries. Nonlocal expression analysis, on the other hand, often

involves computationally hard problems, such as detecting equivalent (common) subexpressions [33, 40], and have to rely heavily on heuristics. This can also be the case when the efficiency of an optimization technique depends on the concrete data values of the involved relations, like choosing the best order in which to join a number of relations.

In this section, we focus on the time-consuming task of sorting. As discussed in [59] in great detail, there are two main reasons for sorting relations. One is to remove duplicates, and the other is to prepare for the merge, when sort-merge algorithms are used to implement binary operators. In both cases, it can be quite difficult to decide if sorting (or to what extent sorting) is required. A major part of the results in this section was motivated by the FACTOR language, but the results apply fully to standard languages. In [59, 60], we have chosen to present the results in a standard framework, partly to point out the general applicability of the results and partly to extend our audience to as large a part of the community as possible. In this paper, however, we will take the opportunity to give a short explanation of how these results can be applied to the FACTOR language.

When trying to minimize sorting, the most interesting place to start is with the unary queries (here, we are talking about unary queries without aggregation). This is because in order to evaluate a unary query, there is not as such any reason to sort. The only reason to sort is to remove duplicates afterwards. So, if it can somehow be established that the unary query does not produce duplicates, then sorting can be avoided entirely, and we reduce the complexity of the query from $O(n \log n)$ to $O(n)$, where n is the size of the relation.

In [60], we have examined unary queries including operations on domain values in a quite general framework. In that paper, we use the notation,

$$[A_1 : exp_1, \ldots, A_n : exp_n]$$

to specify a new relation with schema $\{A_1, \ldots, A_n\}$. The values in this relation are specified by the expressions in which we can use the attribute names from the argument relation as well as constants. Operators on the domain can be applied freely to both attribute names and constants or combinations hereof. We also refer to this type of queries as "for all" queries.

We use this notation for its simplicity. The unary queries are only a part of

the total relational system, so their syntax has to depend on the overall design. In languages where the unary operators are not integrated, the system can translate sequences of unary operators into expressions like the above before our results are applied. If a notation similar to the above is used, as is the case in the FACTOR language, then one would allow additional notation. As it is now, renaming A_3 to B in a relation with schema $\{A_1, A_2, A_3, A_4, A_5\}$ would look like, for example

$$[A_1 : A_1,\ A_2 : A_2,\ A_4 : A_4,\ A_5 : A_5,\ B : A_3]$$

If one allows # as short for the argument tuple $[A_1 : A_1, \ldots, A_n : A_n]$ and allows restriction, then instead we can write

$$\#\backslash A_3[B : A_3]$$

The system can easily translate back to the original notation.

A specification

$$[A_1 : exp_1, \ldots, A_n : exp_n]$$

can be considered a function from sets of tuples to sets of tuples. The query is guaranteed not to produce duplicates if and only if it, considered as a function, is injective.

As an example, consider a relation r with schema $\{A_1, A_2\}$. The query $[B : A_1]$ is not injective since. For example, the two tuples $[A_1 : 5, A_2 : 47]$ and $[A_1 : 5, A_2 : 19]$ both map to $[B : 5]$. This is what we would expect, of course, as the query contains (in algebra terms) a projection (and also a renaming). However, the query $[A_1 : A_1,\ B : A_1 + A_2]$ also contains a projection. In relational algebra, we would write something like

$$\pi_{A_1,B}(extend\ r\ with\ B := A_1 + A_2)$$

And this query is in fact injective.

Notice that although selection is a unary operator, it is uninteresting in connection with this problem, as it can never be the cause of creating duplicates.

So, the unary queries we focus on are the ones which in relational algebra would be written using combinations of projection, renaming, and some form of an extend. In translating from relational algebra to our notation, intermixed selections can simply be ignored in the analysis.

It is, of course, easy to see that the above query is injective, but when queries become a little more difficult, it soon becomes less apparent. The query

$$[\text{Circ: } 2\ (\text{H} + \text{W}),\ \text{Ratio: } (\text{H} + \text{W})/\text{H},\ \text{Wide: } \text{W} > 50]$$

is used in [60] to demonstrate the techniques.

In order to decide injectivity of such queries, we need to have information about the involved operators $+$, $/$, etc. If we capture the full meaning of the operators and the relationships between them in an equational theory, then it is undecidable whether a function is injective or not. So, we have to aim lower. A natural and modular choice is to list information about each operator separately in the form of functional dependencies, e.g., for integer division, we imagine that a relation with schema $\{A, B, C\}$ lists all tuples such that $A/B = C$. This gives rise to the functional dependencies $A, B \to C$ and $A, C \to B$, but *not* $B, C \to A$ (because equations such as $x/3 = 5$ have several solutions; here $x \in \{15, 16, 17\}$).

In [60], we present algorithms which are capable of fully exploiting this kind of operator information. This means, first of all, that if our algorithm answers "proved injective", then the query really is injective. As the problem of determining if a function is injective is undecidable in general, this means, of course, that sometimes when our algorithm answers "no proof", the function is in fact injective; we just failed to find a proof. However, whenever our algorithm answers "no proof", there exists other operators than the ones used in the query, which have the same functional dependencies, and which makes the function not injective. As an example, plus and minus have the same functional dependencies, but

$$[C : A + B,\ D : A - B]$$

is injective, whereas

$$[C : A + B,\ D : A + B]$$

is not.

In [60], we present two algorithms which both solve the problem. The first algorithm produces a propositional Horn clause program from the query, guided by the functional dependencies. We use an algebraic framework [89] to prove the correctness of this method. We obtain the answer by checking whether a few functional dependencies can be deduced. This is done by using the pebbling technique of [32]. The second algorithm transforms the query into a tableau [64, 86] having the functional dependencies which constitute the operator information imposed on its relations. We obtain the answer using this algorithm by checking for validity of a certain functional dependency. This can be done by applying the result of [53], which is based on a chase technique [63] ([53] contains a further development of the results in [51]). Examples of both approaches can be found in [60]. With respect to complexity, the propositional Horn clause algorithm is linear in the size of the query. The chase algorithm also seems to be linear when restricted to handle only what is necessary here. Which one to choose would mostly depend on what software is available.

One interesting open problem is whether these results can be generalized to some extent to avoid sorting when unary operators are mixed with binary ones. Tableaux are a particular kind of conjunctive queries [19, 75], so maybe by formulating the solution in that framework a generalization could be found. As the full relational data model can be expressed using sets of tableaux [76], this is another direction in which a generalization may be found.

As injectivity is undecidable, it is not clear how far one should go in specifying operator information. We believe that using the functional dependencies might be exactly the right level, because this means that we can both incorporate and propagate functional dependencies. That is, functional dependencies which are known to hold for the argument relation can be used in the algorithm, and this information will improve our chances of deeming the function "injective". On the other hand, we can also use the functional dependencies that we collect in the process to check validity of new functional dependencies, which could then we attached (propagated) to the resulting relation.

Unary queries were the obvious starting point, but sorting is a primary cost

in evaluating queries in general. In the following, we focus on sort-merge implementations of the binary operators. This implementation has been used extensively and is described in almost any textbook. It is often advocated that sort-merge is the best implementation of joins [50, 66, 67], and in systems where the decision is made to avoid duplicates in relations, sort-merge is also the superior technique for evaluating the remaining binary operators [28].

As for the unary queries, we base our analysis directly on the query, or rather on the operator graph, which is a special kind of query graph. In [59], we follow the programming languages tradition and refer to the graph as a syntax tree.

A very simple example which illustrates the problem is the following. Consider the query $(r_1 \cup r_2) \bowtie r_3$, where $R_1 = R_2 = \{A, B\}$ and $R_3 = \{B, C\}$. If $r_1 \cup r_2$ is evaluated first without any information about the context in which it appears, then we might choose to sort both relations according to the sort order AB and then merge to obtain the partial result. Proceeding to the join, we observe that the schema of the union intersected with R_3 equals $\{B\}$, so for the purpose of merging, r_3 should be sorted according to BC, and the result of the union according to BA. This means that the result of the union has to be resorted. Of course, this could have been avoided by initially choosing the sort order BA for the relations r_1 and r_2.

This problem of choosing the appropriate sort orders is considered in [80]. Clearly, the problem is primarily how to pass information about required sort orders around in the syntax tree such that whenever there are a number of possible choices, the one which fits in best with the rest of the query is chosen. Smith and Chang try to achieve this by making two *passes* through the tree; first a pass up and then a pass down. By doing so, they can spread information from any leaf to any leaf. There are however, a number of problems with this proposal.

First, it takes both passes to move information about preferred/required sort orders from one leaf to another. But when a leaf v receives information from other leaves, this restricts the possibilities for v, so that its set of possible sort orders changes. That information ought to be spread to other leaves depending on v. The information in *these* leaves will then change and so on.

Second, when Smith and Chang have attached information to the nodes in the syntax tree, they still have to select one sort order for each node among

the remaining possibilities. This they do locally, so they cannot make choices in different parts of the tree which are consistent with one another. This often leads to an unnecessary sorting at the node where these inconsistent choices meet.

Third, they do not in any way recognize the problem of having two or more appearances of the same relation name in a query. Therefore, their algorithm might well assign different sort orders to the same relation. Again, that might be the cause of unnecessary sorting.

Fourth, there are no proofs showing how well their algorithm performs. Given the number of problems which can be identified, it would have been nice to see a proof of the efficiency of their proposal.

In [59], we use similar operator requirements, but instead of just using two passes through the tree, we form an inequational system, which captures the full information. The ultimate goal is to find sort orders such that argument relations are sorted once (unless we have been able to choose orders according to which some of the relations are already sorted), and then no more sorting has to be done—only merging. We solve the inequational system using fixed point techniques [83]. This gives us all possible solutions and we select one afterwards. Thus, we are guaranteed to find a solution if one exists, which we prove in the paper. Formally, we have managed to prove that the running time of the algorithm is bounded by the size of the query squared. As queries are usually short, this is quite acceptable. Furthermore, it seems like most queries can be analyzed even faster. We have not yet come across families of examples, which would make our algorithm use more than linear time in the size of the query. It is therefore possible that the bound can be improved. We conjecture, however, that families of queries with time complexity matching our bound do exist. In [59], we have given the algorithm for standard relational algebra, but it can easily be extended in various directions: other operators can be used, k-way merges [54] can be used to evaluate collections of binary operators, etc.

In addition to fewer sortings, we obtain different degrees of pipelining. In [79], this is referred to as parallel processing. This means that the whole query, or parts of it, can be computed by one large merge using all the input relations (and maybe small buffers). In the best cases, we can then make do with only writing one relation on the disk, i.e., no temporary relations would

have to be created.

If no solution to the entire system exists, then we have to sort after at least one of the merges during the evaluation. We would like to find a minimum number of places to do this. This problem seems quite difficult. In [59], we conjecture that this problem is NP-complete [34] and discuss an approximation algorithm.

4 Update Optimization

When dealing with toy database systems, updates are best handled by using query languages and allowing assignment of relational values to relation names. This is not a feasible approach, however, when dealing with larger and more realistic systems. Whichever update model will eventually be the basis for update languages, like the relational model is the basis for query languages, we believe that the ability to insert, delete, or modify individual tuples will be very central.

When tuples are equipped with one key, the standard approach is to organize the data in some form of a search tree. The advantages of relying on search trees are obvious. As search trees are of interest in many different applications, it has been an on-going research topic for decades. So, a great variety of proposals with different properties exist and most of the complexity issues concerning the proposed structures have already been dealt with. However, the well-established theory of search trees for the sequential case does not carry over directly to the parallel case.

It is natural, though, to choose some of the best sequential structures and attempt to generalize these to a parallel environment. Before we describe our results which are based on red-black trees [39], we shortly describe previous proposals based on other well-known data structures for sequential processing.

Concurrent use of AVL-trees [4] is considered in [49]. Balance information is kept in the nodes of the data structure in the form of a "carry bit", which is set if there is a non-optimal height difference between the subtrees of that node. In that way, the work of keeping the structure optimally balanced can be divided between all of the search and update operations. The advantage

of this is a more evenly distributed overhead, as opposed to the usual sequential case where a few insertions or deletions are the cause of a significant part of the rebalancing. Kessels refer to this as on-the-fly optimization in analogy to the well-known concept of on-the-fly garbage collection [30]. Though this is not proved in the paper, it is also possible to separate the updating and the rebalancing completely, letting background processes do all the rebalancing. Of course, the tree can then become totally unbalanced, if too little rebalancing is done.

A similar algorithm is proposed in [69]. Here, the "carry bit" can hold any integer value greater than or equal to -1. They claim that this improves on the result of Kessels, but no proof of this is included.

The most extensively investigated data structure in the database area is probably B-trees [12]. In the last few years, work has also been done concerning the concurrent use of this, or variants of this, data structure. The first and probably most simple attempt was presented in [78]. Using semaphores [29], all the nodes on a path down to an update are locked. The obvious problem with this approach is that nodes close to the root are locked very frequently and for long periods of time, thereby preventing a high degree of concurrency. There are ways to improve on this without fundamentally changing the idea. In [56], locks are held up to the first insertion or deletion safe node. The number of locks can still be proportional to the height of the tree, though.

The paper [62] introduces an implementation, where at any time only a constant number of locks have to be hold for any single update. The structure which is used is called a B^{link}-tree. This is only a slight modification of B^*-trees [88], where links to the sibling to the right are maintained. Still, overflow is taken care of by the inserting process. As only a constant number of locks are used, a complicated scheme with process queues has to be designed to ensure that the process can traverse the path from the leaf to the root requesting locks and at the same time avoid deadlocks. Sagiv [77] improves slightly on these results by using fewer locks and by allowing background processes to rebalance using compressions.

Finally, in [69], rebalancing is separated entirely from the updating. A "tag bit" is used to register unbalance, and background processes deal with these problems in parallel with searches and updates. The disadvantage, of course, is that the tree can become totally degenerated if not enough background

processes do rebalancing. The advantages are that a high degree of concurrency is allowed as no paths are locked. In fact, not even stepwise locking down paths has to be used, where "stepwise locking" is the term often used as a name for (variations of) the following technique: Assume that a path from the root down to a leaf has node $v_1, v_2, \ldots, v_k$. Assume that at some point, two consecutive nodes v_{i-1} and vi are locked. Now, the "step" consists of first locking u_{i+1} and then releasing v_{i-1}. In this way, it is possible to obtain exclusive access to a leaf while only having at most three locks in effect at any given time. Though this is clearly better than locking more nodes, processes wanting to access a sibling nodes to v_k, say, are still blocked for a long time.

The major problem, which this paper has in common with most of the papers mentioned above, is that none of them actually prove the complexity of their proposals, and only a few test their proposal in comparisons with proposals of others.

Skip lists [73] is a newer and very interesting data structure. It is probabilistic, so it is fundamentally different from the other data structures, which are guaranteed to be balanced or at least become balanced at some point after updating has stopped. However, in addition to exhibiting a very fine average performance, the risk of very bad performance of any particular search is really quite insignificant. The main disadvantage in using skip lists is that each element uses a variable amount of space. This is a problem in main memory, and a disaster in designing an efficient storage plan for secondary memory. Though the data structure is definitely intended for main memory use, reasonable performance is required if the structure, or parts of it, has to be placed (temporarily) on secondary storage. One can, of course, allocate maximum space for all elements, but then thirty-two extra words have to be allocated per element (using the constants suggested by Pugh).

In [68], the data structure chromatic search tree is presented. This data structure is based on red-black trees [39]. It often turns out that constraints on trees which have to be kept invariant has the effect of requiring that a non-constant number of locks be used when implementing the structure directly in a concurrent environment. Typically, all nodes on a path from the root to a leaf have to be locked during updates. To avoid this, chromatic search trees were defined to be similar to red-black trees, only with weaker constraints. Local transformations on the search tree were then proposed

as a means of gradually changing the tree until the original red-black tree constraints would hold. A concurrent rebalancing algorithm does this in parallel with the updates in the tree. There have been earlier, but somewhat different, proposals for local locking of nodes; see [55], for example. Thus, chromatic search trees can also become unbalanced if not enough processes do rebalancing.

Chromatic search trees were presented in [68] without any proofs of their complexity. We have, however, performed a careful analysis which showed that there were indeed efficiency problems. In [16], we presented a new set of local transformations on chromatic search trees, fixing the problems present in [68]. The new proposal was accompanied by a proof of the complexity of the algorithm. The result we obtain is a worst-case complexity of less than $\log N$ rebalancing operations per update, where N is roughly the number of nodes in the tree. Additionally, only a small constant number of nodes need to be locked when rebalancing occurs and each element, including data values, pointers and balancing information, can be stored in five words.

All this can be achieved with a conceptually very simply algorithm which can be programmed in simple independent parts: First, a local problem is found. This can be done by constantly having processes traverse the structure at random. However, it is more efficient to enter problems into a queue when they are created and then simply deal with them in order. This technique has been described in [35]. Second, the involved nodes are locked. The local problem is going to be fixed by carrying out a local transformation involving a small constant number of nodes around the problem. As usual these have to be locked to ensure serializability. Third, the transformation is carried out. The transformations are rotations, double rotations, or weight adjustments as known from red-black trees. Fourth, if a problem queue is used, then it should be checked at this point whether there is still a problem at this location. If so, then this (new) problem is entered into the queue. Finally, the nodes are unlocked.

In conclusion, if both space and time resources are of concern, then the structure presented here seems to be superior to other types of search trees for database implementations.

Summary in Danish

Denne afhandling omhandler design og implementation af databasesprog. Afhandlingen består af fem artikler [16, 59, 57, 60, 61] samt denne oversigtsartikel. Dette afsnit indeholder et sammendrag.

Man begyndte meget tidligt at konstruere databasesystemer og designe databasesprog. De positive og negative erfaringer, man fik gjort sig, udmøntede sig blandt andet i, at man i løbet af 60'erne begyndte at arbejde sig hen imod en fælles datamodel. Dels for organisering af data og dels for opdeling af databaseprogrammer i flere mere eller mindre veldefinerede lag, som kan programmeres hver for sig med på forhånd fastlagte grænseflader som udgangspunkt.

Man kan nok tillade sig at sige, at kulminationen på databaseverdenens anstrengelser med at udvikle en datamodel kom med Codd's [21] artikel i 1970, hvor han præsenterede den relationelle algebra. Selv om andre modeller var i brug tidligere, og selv om nye modeller er blevet foreslået senere, er der ingen tvivl om, at denne model til stadighed samler og repræsenterer området i højere grad end nogen anden model.

Med denne grænseflade for øje har man udviklet de underliggende lag, d.v.s. filsystemer med versionskontrol, sikkerhedskontrol, transaktionskontrol, o.s. v. Fordelen har selvfølgelig været, at man i langt større udstrækning end tidligere har kunnet gøre dette uafhængigt af de øverste lag i et databasesystem. De nederste lag skal blot kunne modtage information i en form svarende til den relationelle model og på rimelig effektiv vis også levere informationen til de øverste lag på denne form.

De øverste lag udgøres så af de egentlige databasesprog. Det vil først og fremmest sige forespørgselssprog, men også i høj grad opdateringssprog.
Disse sprog kan nu tillade sig at operere med relationer og operationer på disse, som de er defineret i den relationelle algebra. Denne abstraktion fra den konkrete repræsentation af data har i høj grad været afgørende for udviklingen af mere vidtfavnende og generelle databasesprog og har åbnet vejen for teoretiske analyser af programfragmenter for eksempel med henblik på semantik-bevarende transformationer i effektivitets øjemed.

I denne afhandling beskæftiger vi os netop med de øverste lag af database-

systemer, hvor vi koncentrerer os om sprogdesign og effektivitet. Før den egentlige gennemgang af resultaterne i de enkelte artikler, vil vi lige i overskriftsform trække hovedlinierne op.

I [57, 61] beskæftiger vi os med design af spørgesprog med særlig vægt på formuleringen af forespørgsler indeholdende aggregering. I [59, 60] ser vi på optimering af evalueringen af forespørgsler gennem direkte analyser af forespørgselsteksten. I [16] fokuserer vi på effektivt design af det mest abstrakte niveau i organiseringen af tuplerne i en relation med henblik på senere opdatering.

Vi beskæftiger os altså i alle artiklerne med det højeste niveau af spørge- og opdateringssprog med vægt på effektivitet i såvel formulering som afvikling af forespørgsler og opdateringer. Dette har vi forsøgt at afspejle i titlen på denne artikel: Effektivitet på de højeste niveauer af databasesprog.

Artiklerne [57, 61] omhandler som sagt aggregering, som i database sammenhæng oftest refererer til det at beregne en enkelt værdi udfra en mængde af værdier. Funktionerne sum, antal, maximum, o.s.v. er typiske. Ofte indgår aggregeringsfunktioner i et samspil med en form for gruppering, som tillader en at opdele en relation i et antal mindre dele med fælles egenskaber. Aggregeringsfunktionerne anvendes så på de enkelte grupperinger og en ny relation formes som samlingen af disse enkelte resultater.

I artiklerne præsenterer vi et nyt sprog kaldet FACTOR, som på flere leder er utraditionelt. Først og fremmest fordi designet direkte baserer sig på gruppering af relationer. Dette betyder, at aggregering kan formuleres meget naturligt i sproget, som dog ikke udtryksmæssigt ændrer styrke i forhold til relationel algebra. Dette er bevist i [57]. En simpel version af sproget er implementeret, og sproget er beskrevet i [58].

En anden speciel egenskab ved FACTOR sproget er, at det vanskeligt kan karakteriseres som værende enten algebra- eller kalkylebaseret. Fordelen ved algebraen er, at operatorerne er begrebsmæssigt meget simple. De tager alle et eller to argumenter og producerer et resultat, der er relateret til argumenterne på en simpel måde. Her er kalkyleforespørgsler typisk mere komplicerede, fordi de tupler, man ønsker at udvælge, skal specificeres v.h.a. sekvenser af eksistentielle og universelle kvantorer. Efter denne udvælgelse kan det endelige resultat dog let specificeres ved at samle attributværdier for de tupler, der er blevet udvalgt. Denne fordel har algebraen til gengæld

ikke. Et godt eksempel på dette er de unære forespørgsler. Man ønsker at angive en funktion, der skal anvendes på hvert tupel, men tvinges i stedet til at formulere dette som en lang kæde af unære relationsoperatorer.

Andre forskere har også argumenteret for det synspunkt, at der ikke er noget unaturligt i at blande algebraen og kalkylen. Merrett [65] refererer til denne skelnen mellem algebra og kalkyle og kalder det (frit oversat): "en historisk fejl, at en del af logikken betragtes som algebra og en anden del som kalkyle".

Lad os kort beskrive hovedideen i FACTOR sproget. Et udtryk ser ud som følger:

$$r_1, \ldots, r_n : exp$$

Fællesmængden af relationernes skemaer, $X = \bigcap_i R_i$, angiver grupperingslisten. En denotationel semantik er givet i [57], men her vil vi nøjes med en operationel forklaring på, hvordan resultatet beregnes. Først formes relationen $\bigcup_i \pi_X(r_i)$. I *exp* kan specialsymbolerne #, @(1),. . . ,@(n) optræde. For hvert tupel $t \in \bigcup_i \pi_X(r_i)$ beregnes *exp* med # bundet til t, og med @(j) bundet til $\pi_{R \setminus X}(\{t\} \bowtie r_j)$. Til sidst tager man foreningen af alle disse evalueringer af *exp*. Det er værd at bemærke, at som et specialtilfælde opnår vi for $n = 1$ samme gruppering som Klug [52].

Det, at der er én fastholdt måde at gruppere på, er typisk for et algebrabaseret sprog. Men i *exp* tillader vi, at delresultater specificeres via # og aggregatfunktioner anvendt på @'erne v.h.a. et tupelsprog. Dette tupelsprog er meget lig det, der anvendes i kalkylerne. I oversigtsartiklen samt i [57, 61] findes en række eksempler på, hvordan denne blanding af algebra og kalkyle har vist sig frugtbar.

Lad os også påpege, at der ser ud til at være muligheder for at forfølge FACTOR idéen i andre retninger. For eksempel håndteres indlejrede relationer meget elegant. Det er ikke nødvendigt at introducere ekstra operatorer (for eksempel *nest* og *unnest*), som det eller normalt gøres. Det samme gælder i øvrigt for naturlige udvidelser af samlingen af aggregatfunktioner til også at omfatte mængdeoperationer så som fællesmængde og foreningsmængde.

Med hensyn til kompleksiteten af FACTOR er den asymptotisk den samme som tilsvarende spørgesprog. Desuden kan de fleste optimeringsteknikker, der retter sig mod en forbedring af køretiderne med en konstant faktor, anvendes

på FACTOR sproget med et lige så resultat. I det følgende vil vi beskrive et optimeringsresultat, der blev udviklet direkte inspireret af FACTOR.

Som det fremgår af det ovenstående, kan forespørgsler i FACTOR formuleres v.h.a. et tupelsprog, og de unære forespørgsler kan formuleres udelukkende v.h.a. dette delsprog. De unære forespørgsler er specielt interessante i systemer, hvor man ikke tillader forekomsten af dubletter i relationer; d.v.s. flere identiske tupler i samme relation. En unær forespørgsel som sådan giver nemlig ikke anledning til, at der skal sorteres på noget tidspunkt. Begrebsmæssigt skal man jo blot løbe alle tupler i argumentrelationen igennem, anvende en funktion på hvert af disse tupler, og endelig samle alle disse resultattupler sammen til en relation. Men hvis den funktion, man anvender, ikke er injektiv, så risikerer man jo netop at introducere dubletter, som så må fjernes bagefter. Dette vil koste en sortering eller tilsvarende. Beregningstiden bringes da op med mere end en konstant faktor fra $O(n)$ til $O(n \log n)$, hvor n er størrelsen af relationen.

Det er med andre ord af interesse at kunne afgøre injektivitet af funktioner, da en sortering kan spares, hvis svaret er positivt. Det er velkendt, at dette problem generelt set er uafgørligt, så man kan altså kun håbe på at løse en del af problemet. I [60] udvikler vi en teknik, der kan anvendes på alle standard unære forespørgsler samt visse kombinationer af beregninger på domæneværdierne. Vi specificerer operatorers injektivitetsegenskaber v.h.a. funktionelle afhængigheder. Vi antager for eksempel at funktionaliteten af heltals division er opskrevet i en relation. D.v.s. at relationen indeholder tupler som [16, 3, 5], hvilket angiver, at $16/3 = 5$. Denne operator får da de funktionelle afhængigheder $1, 2 \rightarrow 3$ og $1, 3 \rightarrow 2$ tilknyttet sig. Her angiver den sidste altså, at første og tredie element i tuplet entydigt fastlægger det andet element. Den sidste mulige afhængighed $2, 3 \rightarrow 1$ gælder altså ikke, da ligningen $x/3 = 5$ har mere end en løsning (nemlig $x \in \{15, 16, 17\}$). Relationen behøver ikke faktisk at eksistere. Vi angiver blot informationen som om den gjorde.

I [60] præsenterer vi to algoritmer, der begge, givet information på ovenstående form samt en forespørgsel, giver et bud på, om funktionen er injektiv eller ej. Da problemet er uløseligt, svarer algoritmerne altså ikke altid rigtigt, men det er dog sådan, at når de svarer, at funktionen er injektiv, så er den det virkelig. Derimod kan algoritmerne fejlagtigt svare, at en funktion ikke er injektiv.

Den ene algoritme er baseret på en transformation af forespørgslen til et logikprogram uden variable. Der er i [32] teknikker til at afvikle disse programmer i lineær tid. Den anden algoritme transformerer problemet til et spørgsmål om beviselighed af funktionelle afhængigheder. Der er så teknikker, kaldet chase, der kan afgøre sådanne spørgsmål. Disse teknikker er af samme kompleksitet som teknikkerne til evaluering af logikprogrammerne .

Når det gælder forespørgsler i almindelighed i stedet for de unære alene, kan det selvfølgelig også være rart at spare nogle sorteringer. Dette vil ikke bringe kompleksiteten ned fra $O(n \log n)$ til $O(n)$ som i det unære tilfælde, da sorteringer på et eller andet tidspunkt under evalueringen er nødvendige, hvis man vil undgå dubletter. Det er dog stadig af stor interesse at spare sorteringer blot for at bringe evalueringstiden ned med en konstant faktor.

I [59] ser vi på generelle forespørgsler, og vi forsøger gennem en analyse af selve forespørgselsteksten at arrangere den senere evaluering sådan, at nogle af sorteringerne kan undgås. Vore resultater kan anvendes af systemer, der baserer sig på den meget udbredte *sort-merge* evalueringsteknik. Som navnet antyder, evaluerer man et simpelt udtryk indenholdende en operator ved først at sortere argumenterne og derefter flette sig til resultatet. Det minder en del om den velkendte sorteringsalgoritme ved (næsten) samme navn.

Problemet kan illustreres ved et meget simpelt eksempel. Betragt forespørgslen $(r_1 \cup r_2) \bowtie r_3$, hvor skemaer er $R_1 = R_2 = \{A, B\}$ og $R_3 = \{B, C\}$. I en naiv implementation, hvor man evaluerer $r_1 \cup r_2$ først uden at tage hensyn til den kontekst, den optræder i, kunne man finde på at vælge at sortere r_1 og r_2 i den leksikografiske orden AB, hvorefter man fletter sig frem til resultatet $r_1 \cup r_2$. Dette delresultat vil nu også være sorteret ifølge AB. Nu kræver operatoren $\bowtie$ imidlertid, at argumenterne er sorteret således, at det fælles skema, det vil altså sige $\{B\}$, er det mest betydende. Så nu skal r_3 samt resultatet af at beregne $r_1 \cup r_2$ sorteres. Man observerer selvfølgelig her, at denne sortering af $r_1 \cup r_2$ kunne have været undgået, hvis man til at begynde med havde valgt at sortere r_1 og r_2 ifølge BA.

I ovenstående eksempel er det let at se, hvordan man arrangerer sig optimalt, men i det generelle tilfælde med store forespørgsler og muligvis med flere forekomster af samme relationelle argument kan det være yderst vanskeligt. Det bedste, man kan håbe på, er, at man kun behøver at sortere alle argumenterne til en stor forespørgsel, hvorefter alle delresultater kan bereg-

nes ved blot at flette på passende vis. Så opnår man også, hvad vi kunne kalde *perfekt gennemstrømning*, idet delresultater faktisk ikke behøver at blive beregnet. I stedet kan man udføre en mere kompliceret fletning, der beregner det endelige resultat med det samme. Dette vil spare ekstra meget tid, da dyre skrivninger på eksternt lager kan undgås.

I [59] præsenterer vi en algoritme, der finder en sådan perfekt gennemstrømning, hvis der findes en sådan. Tidskompleksiteten for denne algoritme er kvadratisk i forespørgslens størrelse. Løsningsteknikken baserer sig på det at finde et maksimalt fixpunkt i et gitter.

Hvis der ikke findes en perfekt gennemstrømning, er man selvfølgelig interesseret i at få det næstbedste, nemlig en evaluering, hvor der sorteres så få gange som muligt i løbet af evalueringen. Vi mener, at dette problem er NP-hårdt, og i [59] foreslår vi en approksimationsalgoritme for dette problem.

Lad os yderligere nævne, at information, der opbevares om relationers tilstand m.h.t. sortering, også kan udnyttes. Der er jo den oplagte mulighed, at en relation allerede er sorteret, eller at den har et indeks, der gør, at det er hurtigere at gennemløbe tuplerne i en orden end i en anden. Endelig kan det være, at tuplerne er anbragt i en træstruktur. I de fleste implementationer af træstrukturer vil det betyde, at tupler, der har nøgleværdier tæt på hinanden, også vil have en tendens til at være placeret på samme sektor på et pladelager. En træstruktur, der blandt andet har den egenskab, er beskrevet nedenfor.

Det, at uddrage information fra en database, er et meget vigtigt element i design af databasesystemer, men opdatering er som regel næsten lige så væsenligt. Næsten underordnet hvilken opdateringsmekanisme, man ønsker at understøtte, bliver det centrale punkt det at genfinde og indsætte enkelte tupler hurtigt. Hvis relationer er gemt på den naive måde som en uordnet sekvens af tupler, er den eneste måde at genfinde et tupel på en lineær søgning gennem sekvensen. For store relationer er dette uhyre tidskrævende. I årenes løb har man derfor udviklet en stor rigdom af træstrukturer med mange forskellige egenskaber. Fælles for dem er dog, at søgetiden gerne skulle holde sig omkring log n, hvor n er størrelsen af den relation, man søger i.

I databasesystemer, der kun understøtter, at en proces af gangen har adgang til træstrukturen, kan man altså benytte sig af den uhyre veludviklede teori

og vælge sig et AVL-træ, et rød-sort træ eller et B-træ med passende egenskaber i forhold til anvendelsen. I multiprocessystemer er det ikke slet så nemt. Den veludviklede teori fra det sekventielle tilfælde kan ikke umiddelbart føres over.

I [16] ser vi på en variant af rød-sorte træer, kaldet kromatiske træer. Definitionen i denne artikel er en videreudvikling af forslaget fra [68]. Desuden indeholder [16] i modsætning til hovedparten af lignende forslag et bevis for kompleksiteten af de involverede operationer. Udover selve opdateringsoperationerne drejer det sig som altid om operationer, der skal holde træet rimeligt balanceret, sådan at søgetiden så vidt muligt hele tiden holdes nede på log n. I [16] lykkes det at definere en konstruktion, hvor en opdatering giver anledning til mindre end log n rebalanceringsoperationer, hvoraf højst en ændrer træets struktur. Dette er af afgørende betydning for effektiviteten af andre søgninger, der måtte finde sted samtidigt med omtalte rebalancering. Desuden bruger denne struktur meget lidt plads – i særdeleshed i forhold til andre strukturer med rimelig hurtige opdaterings- og rebalanceringstider.

References

[1] Serge Abiteboul. Updates, a New Frontier. In *ICDT,* pages 1-18, l988.

[2] Serge Abiteboul and Catriel Beeri. On the Power of Languages for the Manipulation of Complex Objects. Rapports de Recherche 846, INRIA, 1988.

[3] Serge Abiteboul and Nicole Bidoit. Non First Normal Form Relations To Represent Hierarchically Organized Data. In *ACM PODS,* pages 191-200, 1984.

[4] G. M. Adel'son-Vel'slciĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Dokl. Akad. Nauk SSSR,* 146:263-266, 1962. In Russian. English translation in *Soviet Math. Dokl.,* 3:1259-1263, 1962.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[6] Alfred V. Aho and Jeffrey D. Ullman. Universality of Data Retrieval Languages. In *ACM POPL,* pages 110-120, 1979.

[7] Lars Arge, Mikael Knudsen, and Kirsten Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. Personal communication. To appear as MS thesis, Computer Science Department, Aarhus University.

[8] W. W. Armstrong. Dependency Structures of Data Base Relationships. In *Proc. IFIP Congress,* pages 580-583. North-Holland, 1974.

[9] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM TODS,* 1(2):97-137, 1976.

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

[11] Malcolm Atkinson, François Banchilhon, David Dewitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. Altair Tech. Report 30-89, Rocquencourt, France, August 1989.

[12] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Inf.,* 1(3):97-137, 1972.

[13] Catriel Beeri. Data Models and Languages for Databases. In *ICDT,* pages 19-40, 1988.

[14] P. A. Bernstein and D.-M. Chiu. Using Semi-Joins to Solve Relational Queries. *J. ACM,* 28(1):25-40, 1981.

[15] Philip A. Bernstein. Synthesizing Third Normal Form Relations from Functional Dependencies. *ACM TODS,* 1(4):277-298, 1976.

[16] Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. In O. Nurmi and E. Ukkonen, editors, *LNCS 621: Algorithm Theory - SWAT'92,* pages 151-164. Springer-Verlag, 1992.

[17] Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors. *On Conceptual Modelling.* Topics in Information Systems. Springer-Verlag, 1984.

[18] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A History and Evaluation of System R. *Comm. ACM,* 24(10):632-646, 1982.

[19] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *ACM STOC,* pages 77-90, 1977.

[20] P. Chen. The Entity-Relationship Model-Toward a Unified View of Data. *ACM TODS,* 1(1):9-36, 1976.

[21] E. F. Codd. A Relational Model of Data for Large Shared Data Bases. *Comm. ACM,* 13(6):377-387, 1970.

[22] E. F. Codd. Relational Completeness of Data Base Sublanguages. In Randall Rustin, editor, *Data Base Systems,* pages 65-98. Prentice-Hall, 1972.

[23] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a Visual Formalism for Real Life Recursion. In *ACM PODS,* pages 404-416, 1990.

[24] Mariano P. Consens and Alberto O. Mendelzon. Low Complexity Aggregation on GraphLog and Datalog. In *ICDT,* pages 379-394. Springer-Verlag, 1990.

[25] C. J. Date. *An Introduction to Database Systems, Vol. 1.* The Systems Programming Series. Addison-Wesley, 1975.

[26] C. J. Date. *An Introduction to Database Systems, Vol. 2.* The Systems Programming Series. Addison-Wesley, 1983.

[27] C. J. Date. *An Introduction to Database Systems, Vol. 1.* Addison-Wesley, 1990.

[28] Bipin C. Desai. *An Introduction to Database Systems.* West Publishing Company, 1990.

[29] E. W. Dijkstra. Co-Operating Sequential Processes. In F. Genuys, editor, *Programming Languages.* Academic Press, 1968.

[30] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: an exercise in cooperation. *Comm. ACM,* 21:966-975, 1978.

[31] R. A. DiPaola. The Recursive Unsolvability of the Decision Problem for a Class of Definite Formulas. *J. ACM,* 16(2):324-327, 1969.

[32] William F. Dowling and Jean H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Logic Programming,* 1(3):267-284, 1984.

[33] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the Common Subexpression Problem. *J. ACM,* 27(4):758-771, 1980.

[34] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman, 1979.

[35] N. Goodman and D. Sasha. Semantically Based Concurrency Control for Search Structures. In *ACM PODS,* pages 8-19, 1985.

[36] P. M. D. Gray and R. Bell. Use of Simulators to Help the Inexpert in Automatic Program Generation. In P. A. Samet, editor, *Euro IFIP,* pages 613-620. North-Holland, 1979.

[37] Peter 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.

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

[39] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *IEEE FOCS,* pages 8-21, 1978.

[40] P. A. V. Hall. Common Subexpression Identification in General Algebraic Systems. Report UKSC0060, IBM UKSC, Peterlee, Co. Durham, England, November 1974.

[41] P. A. V. Hall. Optimization of Single Expressions in a Relational Data Base System. *IBM J. Res. Devel.,* 20(3):244-257, 1976.

[42] Richard Hull and Chee K. Yap. The Format Model: A Theory of Database Organization. In *ACM PODS,* pages 205-211, 1982. Extended abstract. Full version in: Tech. Report, Dept. of Comp. Sci., University of Southern California, Sept. 1981.

[43] G. Jaeschke. The Theory of One Attribute Nesting. Technical note, Heidelberg, Scientific Center, 1982.

[44] G. Jaeschke and H.-J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *ACM PODS,* pages 124-138, 1982.

[45] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys,* 16(2):111-152, 1984.

[46] Rowland R. Johnson. Modelling Summary Data with the Entity Relationship Model. Technical Report 10647, Lawrence Berkeley Laboratory, l980.

[47] Rowland R. Johnson. Modelling Summary Data. In *Proc. ACM SIGMOD,* pages 93-97, 1981.

[48] Paris C. Kanellakis. Elements of Relational Database Theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Models and Semantics,* volume B, chapter 17, pages 1073-1156. Elsevier Science Publishers, 1990.

[49] J. L. W. Kessels. On-the-Fly Optimization of Data Structures. *Comm. ACM,* 26:895-901, 1983.

[50] W. Kim. A New Way to Compute the Product and Join of Relations. In *ACM SIGMOD,* pages 179-187, 1980.

[51] A. Klug. Calculating Constraints on Relational Expressions. *ACM TODS,* 5(3):260-290, 1980.

[52] Anthony Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *J. ACM,* 29(3):699-717, 1982.

[53] Anthony Klug and Rod Price. Determining View Dependencies Using Tableaux. *ACM TODS,* 7(3):361-380, 1982.

[54] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.

[55] H. T. Kung and Philip L. Lehman. Concurrent Manipulation of Binary Search Trees. ACM TODS, 5(3):354-382, 1980. First published as an abstract: A Concurrent Database Manipulation Problem: Binary Search Trees, *Very Large Data Bases,* page 498, 1978.

[56] Yat-Sang Kwong and Derick Wood. A New Method for Concurrency in B-Trees. *IEEE Trans. Software Eng.,* 8(3):211-222, 1982.

[57] Kim S. Larsen. On Aggregation and Computation on Domain Values. PB 414, Computer Science Department, Aarhus University, 1992.

[58] Kim S. Larsen. RASMUS User's Manual. MD 60, Computer Science Department, Aarhus University, 1992.

[59] Kim S. Larsen. Strategies for Expression Evaluation Using Sort-Merge Algorithms. PB 415, Computer Science Department, Aarhus University, 1992.

[60] Kim S. Larsen and Michael I. Schwartzbach. Injectivity of Unary Queries With Computation on Domain Values. Computer Science Department, Aarhus University, 1992. Revised version of PB 311.

[61] Kim S. Larsen, Michael I. Schwartzbach, and Erik M. Schmidt. A New Formalism for Relational Algebra. *IPL,* 41(3):163-168, 1992.

[62] Philip L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM TODS,* 6(4):650-670, 1981.

[63] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM TODS,* 4(4):455-469, 1979.

[64] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[65] T. H. Merrett. The Extended Relational Algebra, a Basis for Query Languages. In Ben Shneiderman, editor, *Databases: Improving Usability and Responsiveness,* pages 99-128. Academic Press, 1978. Proc. Int. Conf. Databases: Improving Usability and Responsiveness.

[66] T. H. Merrett. Why Sort-Merge Gives the Best Implementation of the Natural Join. *SIGMOD Record,* 13(2):39-51, 1983.

[67] T. H. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of Page-Fetches in Join Operations. In *VLDB,* pages 488-498, 1981.

[68] O. Nurmi and E. Soisalon-Soininen. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In *ACM PODS,* pages 192-198, 1991.

[69] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency Control in Database Structures with Relaxed Balance. ln *ACM PODS,* pages 170-176, 1987.

[70] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos. Extending Relational Algebra and Relational Calculus with Set-valued Attributes and Aggregate Functions. *ACM TODS,* 12(4):566-592, 1987.

[71] Frank P. Palermo. A Data Base Search Problem. In Julius T. Tou, editor, *Informations Systems COINS IV,* pages 67-101. Plenum Press, 1974.

[72] William Pugh. Concurrent Maintenance of Skip Lists. Technical Report CS-TR-2222.1, Dept. of Comp. Sci., University of Maryland, April 1989. Revised June, 1990.

[73] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *LNCS 382: Algorithms and Data Structures,* pages 437-449. Springer-Verlag, 1989. Proc. WADS'89.

[74] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of Recursive Horn Clauses With Infinite Relations. In *ACM PODS,* pages 328-339, 1987.

[75] D. J. Rosenkrantz and H. B. Hunt. Processing Conjunctive Predicates and Queries. In *VLDB,* pages 64-72, 1980.

[76] Y. Sagiv and M. Yannakakis. Equivalences Among Relational Expressions With the Union and Difference Operators. *J. ACM,* 27(4):633-655, 1980.

[77] Yehoshua Sagiv. Concurrent Operations on B^*-Trees with Overtaking. *J. Comp. and System Sci.,* 33:275-296, 1986.

[78] Behrokh Samadi. B-Trees in a System with Multiple Users. *IPL,* 5(4):107-112, 1976.

[79] J. W. Schmidt. Parallel Processing of Relations: A Single-Assignment Approach. In *VLDB,* pages 398-408, 1979.

[80] John Miles Smith and Philip Yen-Tang Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Comm. ACM,* 18(10):568-579, 1975.

[81] Michael Stonebraker. Retrospection on a Database System. *ACM TODS,* 5(2):225-240, 1980.

[82] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The Design and Implementation in INGRES. *ACM TODS,* 1(3):189-222, 1976.

[83] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math,* 5:285-309, 1955.

[84] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[85] Jeffrey D. Ullman. Database Theory: Past and Future. In *ACM PODS,* pages 1-10, 1987.

[86] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems,* volume 1. Computer Science Press, 1988.

[87] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems,* volume 2. Computer Science Press, 1989.

[88] H. Wedekind. On the Selection of Access Paths in a Data Base System. In J. W. Klimbie and K. L. Koffeman, editors, *Data Base Management,* pages 385-397. North-Holland, 1974.

[89] Martin Wirsing. Algebraic Specification. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Models and Semantics,* volume B, chapter 13, pages 675-788. Elsevier Science Publishers, 1990.

[90] C. Zaniolo. Safety and Compilation of Nonrecursive Horn Clauses. In *Proc. Expert Database Systems,* pages 167-178. Benjamin-Cummings, 1986.

[91] M. M. Zloof. Query-by-Example: a data base language. *IBM Systems Journal,* 16(4):324-343, 1977.

[92] W. Zook, K. Youssefi, N. Whyte, P. Rubinstein, P. Kreps, G. Held, J. Ford, R. Berman, and E. Allman. *INGRES Reference Manual,* 1977.