ISSN 0105-8517

OPTIMIZING THE EVALUATION OF CALCULUS EXPRESSIONS IN A RELATIONAL DATABASE SYSTEM

by

Svend-Erik Clausen

DAIMI PB-97
April 1979

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

OPTIMIZING THE EVALUATION OF CALCULUS EXPRESSIONS IN A RELATIONAL DATABASE SYSTEM

by

Svend-Erik Clausen ‡

Computer Science Department
Aarhus University, Aarhus, Denmark

‡ Present address: Jydsk Telefon A/S, Sletvej 30, 8310 Tranbjerg, Denmark.

OPTIMIZING THE EVALUATION OF CALCULUS EXPRESSIONS IN A RELATIONAL DATABASE SYSTEM

CONTENTS

ABSTRACT

There is an interesting search strategy (due to James B. Rothnie) for efficient implementation of a limited kind of selection criterion for a relational database. This strategy is here generalized to arbitrary relational calculus expressions, and an analysis of the resulting improvement of performance is given. The strategy is used in a relational database system TGR and an overview of the architecture of this system is presented. TGR uses microprogrammed database primitives for searching the database. This approach is very similar to the use of a database processor although it also allows flexible change of processor design. The behaviour of TGR in evaluating typical queries is analyzed and the results are used for pointing out the bottlenecks in a relational database system with a particular type of structure. As a conclusion the construction of a database processor with the database primitives from TGR as instruction set is recommended. This would be a step towards getting acceptable performance in a relational database system.

1. INTRODUCTION

Since 1970 when Codd suggested the relational model for database systems [CODD70], there has been a great deal of discussion of how efficiently working systems supporting this model can be implemented.

The relational database system TGR was designed to investigate these performance problems. It was implemented as a Danish candidate thesis project by Birgitte Madsen, Erling Madsen and the author [TGR78].

The purposes of the project were the following:

1. Design and implementation of an <u>intermediate language</u>, based on relational calculus, which can be used as a basis for implementing a full database system.

2. Design and implementation of a <u>microprogrammed database-machine</u> with a number of database primitives which can be used for the implementation of the intermediate language.

3. Design, implementation and investigation of <u>optimization features</u> intended to speed up retrieval from the database.

A few notes can be given about these purposes:

1) Intermediate language:

It has <u>not</u> been our intention to implement a full database system and therefore TGR lacks a lot of the facilities a commercial database system has to include, e.g. concurrency, security, integrity checks, backup/recovery etc. However, proposals of how to implement these features through the use of the intermediate language are given in [TGR78].

2) Microprogrammed database machine:

There is a close analogy between our use of microprogrammed database primitives and the design of special hardware ([OZKA75], [SU75], [LIN76], etc.). As a tool for research, microprogramming is much more flexible than the use of specialized hardware because the micro-programmed primitives can be changed until the most suitable form is reached. A hardware database machine can then be constructed using the experience gained through the development of microprogrammed primitives.

3) Optimization features:

This is the main subject of this paper. The investigation of optimization features is directed toward strategies not usually used in relational database systems. However, such generally used optimization features as use of indices and multilists can be included in TGR, too.

The rest of the paper is organized as follows:

Section 2 contains an outline of TGR. The various parts of the system are described and an overview of the optimization features is given.

Sections 3 and 4 describe one of the optimization strategies implemented in TGR. It is a generalization of a strategy developed by Rothnie [ROTH74] for a limited kind of calculus-expressions.

Section 5 gives a description of the current status of the system - what is now implemented and what is intended to be implemented in the near future.

Section 6 investigates the performance of TGR using the optimization strategy presented in section 4. Two queries are executed on example databases and time measurements are given for these queries.

In section 7 our experiences with TGR are presented as the conclusion of this report.

Throughout the paper some terminology introduced in [ROTH74] is used. However, most of the terms are self-explanatory.

Notation used in all figures is:

flow of control and data	$\rightarrow$
flow of data.	$\Rightarrow$

2. AN OUTLINE OF THE DATABASE SYSTEM

A full database system based on TGR would look like:

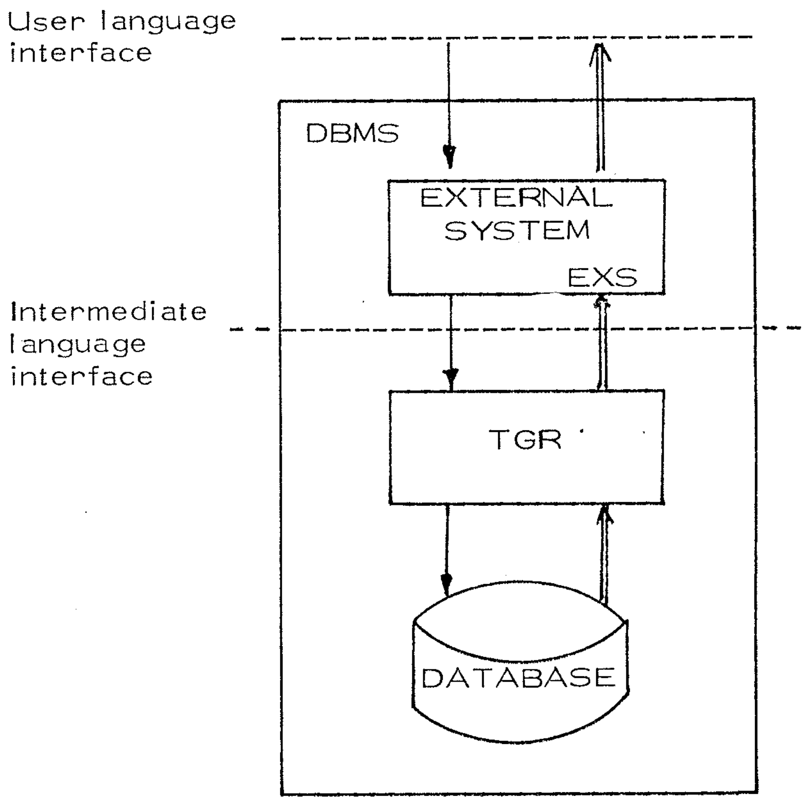


Figure 2.1.

The intention with TGR is that it should be used as a "bottom end" in a full DBMS. TGR only implements basic retrieval and update operations - other features like security, integrity checks, recovery etc. are implemented in EXS, the external system. Precisely what to implement in EXS and how to implement it is up to the designer of the full DBMS containing TGR.

The interface between EXS and TGR is ILT, the Intermediate Language of TGR.

ILT is a set-oriented, relationally complete language based on the relational calculus. It is very similar to the ALPHA language [CODD71] and is intended to be used as a target language for a user language (like SEQUEL [ASTR76] or QUEL [STON76]) implemented in EXS. A precise definition of the syntax of ILT is found in [TGR78].

For ease of implementation the format of ILT is very restricted:

- prenex normal form
- conjunctive normal form
- without the negation operator,

but none of these restrictions causes a loss in the descriptive power of ILT.

The rest of this paper will mainly deal with the retrieving of data, since this is the most interesting part of the system. Update and data definition facilities are described in [TGR78].

The choice of the relational model for TGR gives a very high degree of data independence which is necessary to achiève the goals mentioned in the introduction. In Fig. 2.2 everything below the one-variable expression interface can be changed, without affecting the higher levels of the system (e.g. the usage of different logical or physical storage structures or a hardware implemented database machine).

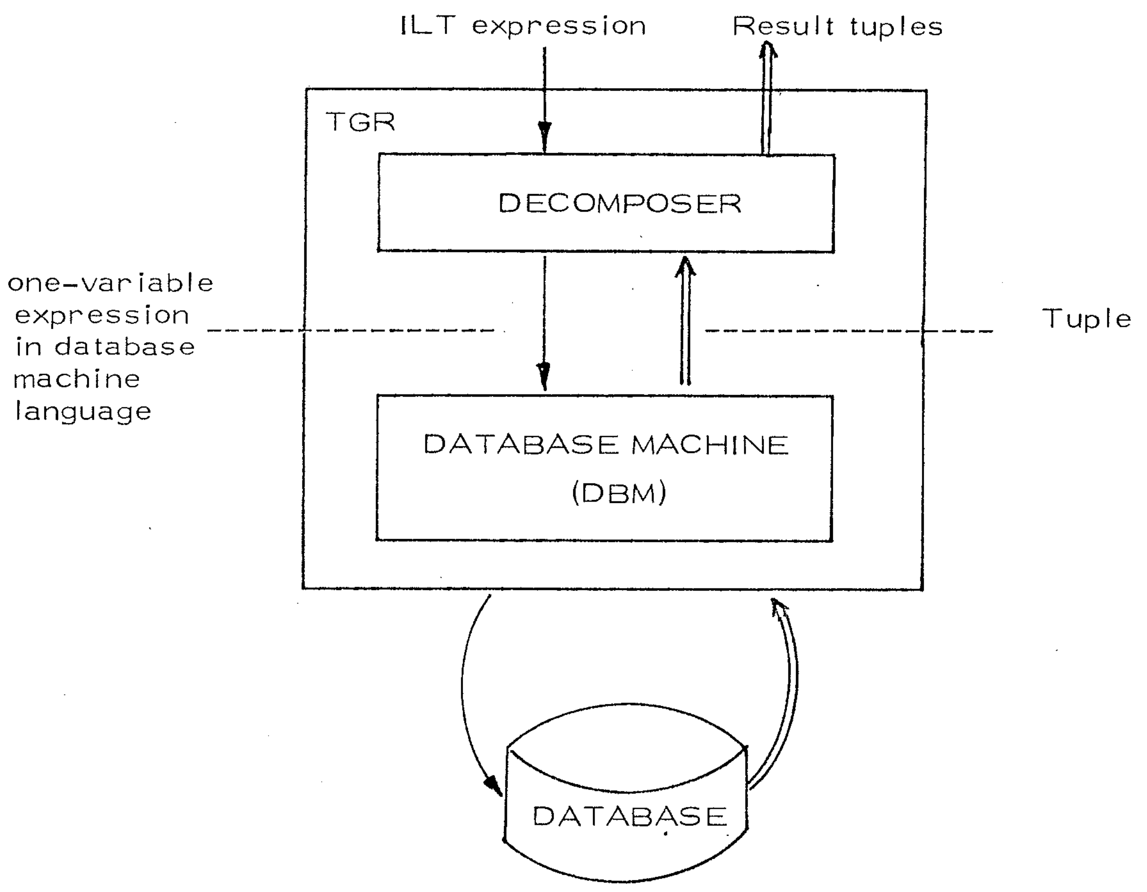


Figure 2.2

TGR functions in the following way: An ILT expression, possibly including several tuple-variables, is decomposed by the DECOMPOSER to a sequence of one-variable expressions. Each one-variable expression is executed by the DATABASE MACHINE (DBM) and this execution transfers some tuples to the DECOMPOSER. These tuples are used by the DECOMPOSER both to select the next one-variable expression to submit to the DBM and to do the feed-back optimization described in sections 3 and 4. Also, some of the tuples are transferred to EXS as the value of the ILT expression.

The modularity in this approach gives the possibility to separate the DBM into modules, one for each storage structure supported, as described in [ROTH74] and to create and remove modules without affecting the DECOMPOSER. However, when dealing with complicated expressions joining several relations, it has a great impact on the performance to choose the right sequence of one-variable expressions for the DBM. The approach in TGR is to use the knowledge of existing storage structures (or use of special hardware) in the decomposition to find the optimal sequence of instructions for the DBM. This is based on a generalization of ideas presented by Pecherer in [PECH76] but only partially implemented. The approach will not be further discussed in this paper but a detailed description is found in [TGR78].

As mentioned before the structure of the DBM can vary a lot within the design of TGR. The only demand is that the one-variable interface be supported. The DBM can be software, firmware or hardware implemented and for instance the hardware implemented "search processor" developed at Braunschweig [LEIL78] could be used directly in TGR, since it supports a one-variable expression interface.

Our design of the DBM uses a mixture of software and firmware implementation and it is our hope that this design can eventually be used as a guideline for a hardware implementation of some parts of the DBM.

The structure of the DBM is:

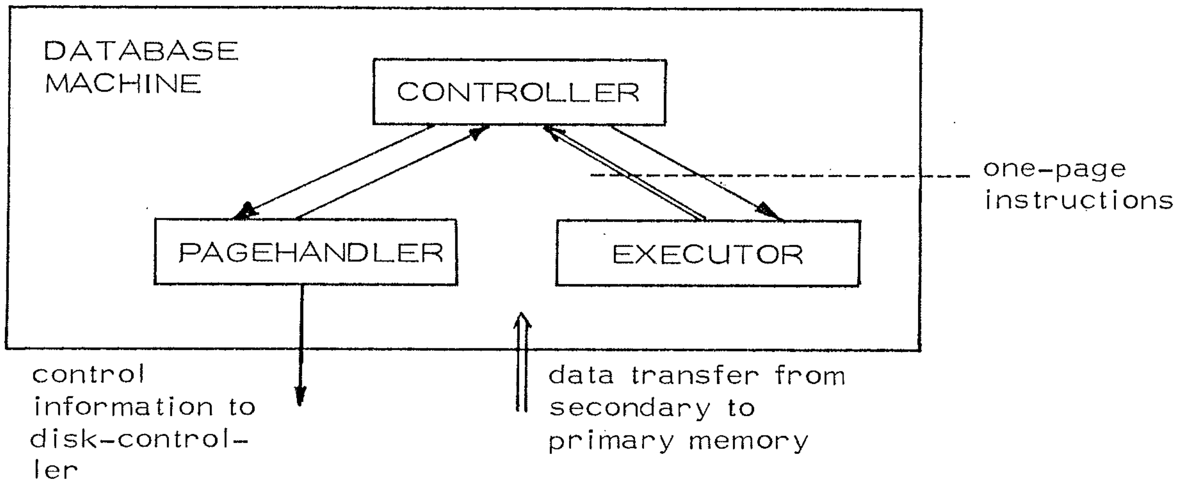

DATABASE MACHINE
CONTROLLER
PAGEHANDLER
EXECUTOR
one-page instructions
control information to disk-controller
data transfer from secondary to primary memory

Figure 2.3

The components in Figure 2.3 are:

- The CONTROLLER: translates a one-variable expression to a sequence of <u>one-page</u> instructions for the EXECUTOR at the same time controlling the PAGEHANDLER.

- The PAGEHANDLER transfers data pages, which are going to be examined by the EXECUTOR, between secondary and primary memory.

- The EXECUTOR: executes the one-page instructions.

TGR is implemented on a system comprising two microprogrammable computers, RIKKE and MATHILDA ([STAU74], [KORN75]) belonging to the experimental equipment at DAIMI, University of Aarhus. A short description of this system is found in appendix A.

The EXECUTOR of the DBM is microprogrammed on MATHILDA to simulate a database processor. The rest of TGR is programmed in BCPL on RIKKE.

3. OPTIMIZATION OF TWO-VARIABLE EXPRESSIONS

In this section the basic ideas behind feedback-optimization, one of the optimization strategies used in the decomposer of TGR, will be outlined. These ideas are the same as used by Rothnie in DAMAS [ROTH74], the only difference being their adaption for TGR.

Let us look at a very simple example:

Example 3. 1:

R1 is the relation

A1	A2
3	6
4	5
10	2
15	0

R2 is the relation

A1	A2
2	8
5	7
9	6
8	3

The query

GET $R1.A1 : \forall R2\ (R1.A1 > R2.A1 \wedge R1.A2 < R2.A2)$

is executed as follows:

For the tuple $<3,6>$ from R1, relation R2 is scanned to see if the qualification of the query is satisfied (i. e. $\forall R2\ (3 > R2.A1 \wedge 6 < R2.A2)$). This is not the case since the tuple $<5,7>$ from R2 does not satisfy $(3 > R2.A1 \wedge 6 < R2.A2)$. From this we gain the following information:

If the condition $F1 = (T1.A1 \leq 5 \vee T1.A2 \geq 7)$ is true for a tuple T1 in R1 then the qualification for this tuple is false.

It is clear that the above statement is true because we know that the tuple $<5,7>$ exists in R2 and F1 is the negation of the condition from the qualification.

The information is used to eliminate those tuples from R1 which cannot cause a true qualification. For each of the tuples eliminated,a partial scan of R2 is saved. The elimination is done by creating an elimination-filter on R1 containing F1 as condition. For every tuple found in a scan of relation R1 the elimination-filter is checked and, if the condition is fulfilled, the tuple is eliminated.

F1 is actually true for the next tuple $<4, 5>$ from R1 and this tuple is therefore eliminated.

The tuple $< 10, 2>$ does not satisfy the filter so R2 is scanned. The qualification is true for this tuple which means that $< 10>$ is in the result. The scan of R2 gives the information that every tuple T2 in R2 satisfies $(10> T2.A1 \wedge 2<T2.A2)$. Using this information some tuples can be found for which we can deduce, without scanning R2, that the qualification is true, namely the tuples T1 from R1 satisfying $F2 = (T1.A1 \geq 10 \wedge T1.A2 \leq 2)$. Therefore a true-filter on R1 is established, containing F2 as condition. If a tuple T1 from R1 fulfills F2, it is known in advance that the qualification for T1 is true, and there is no need for a scan of R2. In this way a complete scan of R2 is saved for every tuple from R1 satisfying F2.

In fact, the last tuple $<15, 0>$ from R1 satisfies the true-filter giving $< 15>$ as a result item. After that the execution of the query is finished with $< 10>$ and $< 15>$ as result.

Example 3. 1 was a very straightforward example of the use of Rothnie's optimization method. Treating more complex queries is rather complicated expecially when dealing with multi-variable expressions, which is the subject of the next section.

Of course, the use of these optimization methods will not always give a faster execution of a query. Therefore the setting and clearing of options is used to decide whether or not to use a specific method. In DAMAS option A and option B are used, corresponding to the use in TGR of respectively true- and elimination-filters. In example 3. 1 the true-filter would only be created if option A was set and equivalent with option B and elimination-filters.

In DAMAS and TGR an option C is implemented, too. Option C is concerned with the elimination of duplicates. For each duplicate the calculation of the qualification is avoided. Option C will not be further discussed in this paper.

When to set an option is a very difficult problem which is not yet solved. In section 6 a little bit more is mentioned about this subject.

4. OPTIMIZATION OF ARBITRARY (MULTI-VARIABLE) EXPRESSIONS

The generalization of the feedback-optimization technique from section 3 has been done with the following intention. When evaluating a (multi-variable) query in TGR, all information gained from a search of a relation should be used, where advantageous, to avoid some searches later in the evaluation, thereby speeding up retrieval from the database.

Section 4 describes how this goal is achieved, when and how information is collected and how it is used. Furthermore an explanation is given of a division of each of the two-variable options A and B, mentioned in section 3, into three parts to encompass the new situations in multi-variable expressions.

Example 4.1

R1 is the relation

A1
13
5
4
9

R2 is the relation

A1
8
3

R3 is the relation

A1
7
9

R4 is the relation

A1
7
4
11

The query

$$Q1 = \text{GET}\ (R1.A1,\ R2.A1):$$
$$\forall R3\ \exists R4\ (R1.A1 < R3.A1 \wedge R2.A1 > R4.A1 \wedge R3.A1 \neq R4.A1)$$

is executed by an algorithm using nested loops, and a stage in this execution is represented by a diagram like Fig. 4.1. An arrow means that the search of a relation has come to this particular tuple called the <u>current tuple</u> of the relation. If no arrow is pointing at any tuples in a relation a search of that relation is not in progress and the current tuple is undefined.

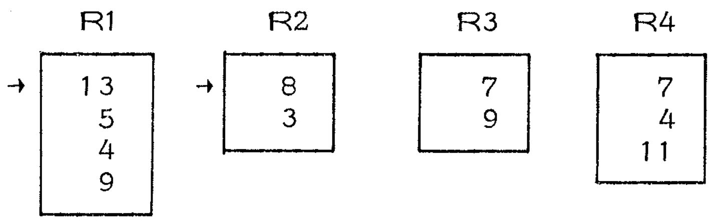


Figure 4.1.

Fig. 4.1 represents the start situation where searches are opened on R1 and R2 and the next thing to happen is to search R3 and R4 to check the qualification of Q1 for the combination (<13>, <8>) from R1 and R2.

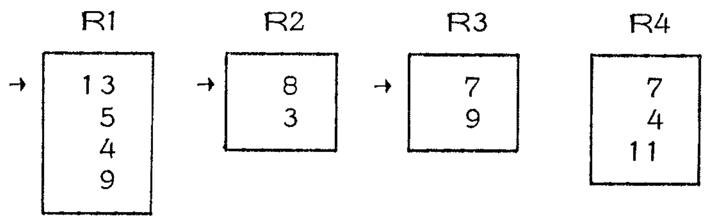


Figure 4.2

In Fig. 4.2 a search of R3 is opened, to check if all tuples from R3 satisfy the condition $C1 = [13 < R3.A1]$ (from $R1.A1 < R3.A1$ in the qualification of Q1). C1 is immediately false for the tuple T3 = <7> and therefore the qualification for (<13>, <8>) is false. T3 can be used to establish the information that a tuple from R1 satisfying $F1 = [R1.A1 \geq 7]$ always gives a false qualification. Therefore an eliminination-filter is created on R1 with the condition F1 (in fact, the filter is on the combination of R1 and R2 but as it only concerns R1 it can be put on R1). This is an example of a use of option B1all, a special instance of option B concerned with the universal quantifier.

The next interesting situation is:

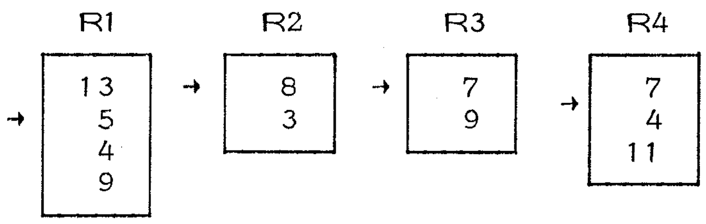


Figure 4. 3

where the qualification for (<5>, <8>) is examined. The search of R4 checks if there exists a tuple T4 from R4 such that: $[8 > T4.A1 \wedge 7 \neq T4.A1]$. This is the case for the tuple <4> from R4. Having the knowledge that the tuple <4> with this property exists in R4, a true-filter $F2 = [R2.A1 > 4 \wedge R3.A1 \neq 4]$ is put on the combination of R1, R2, and R3.

This is an example of option Asome, an instance of option A concerning the existential quantifier.

In the situation

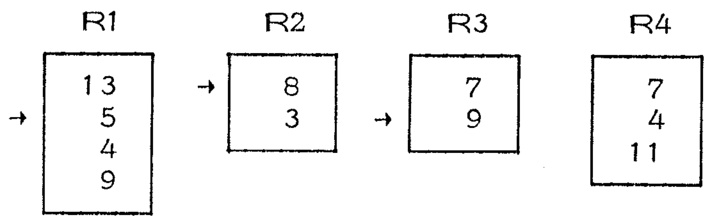


Figure 4. 4

the combination <5>, <8> and <9> from R1, R2, and R3 satisfies the true-filter F2 and therefore R4 is not searched. The search of R3 has now been finished and the qualification for <5>, <8> is found to be true. This can be used in the following way: If a combination of tuples T1, T2 from R1 and R2 by insertion in the qualification creates the <u>same</u> or a <u>weaker</u> condition than the condition created by (<5>, <8>) then the qualification is <u>true</u> for (T1, T2). Therefore a true-filter $F3 = [T1.A1 \leq 5 \wedge T2.A1 \geq 8]$ is created for the combination of R1 and R2.

The option used here is called <u>option Aall</u>, an instance of option A concerning the universal quantifier.

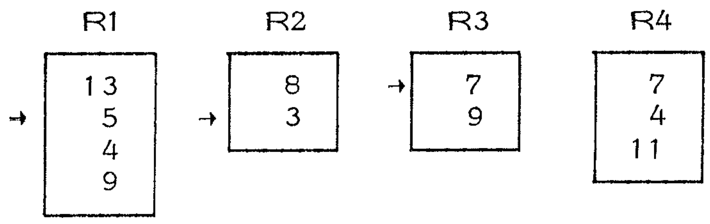


Figure 4.5

In the situation presented in Fig. 4.5 a search of R4 for a tuple satisfying $[3 > R4.A1 \wedge 7 \neq R4.A1]$ is opened which eventually yields a false. At this point an <u>elimination-filter</u> $F4 = [R2.A1 \leq 3 \wedge R3.A1 = 7]$ is created to ensure that the search of R4 is avoided when a search is opened with a <u>"stronger"</u> condition than $[3 > R4.A1 \wedge 7 \neq R4.A1]$. This option is called <u>option B2some</u>.

The search of R3 is also stopped since the condition was false for the tuple <7>. Again an elimination-filter is created, this time on R1 and R2 with the condition $F5 = [R1.A1 \geq 7 \vee R2.A1 \leq 3]$. In <u>option B2all</u>, as this option is called, as in option Asome and option B1all, the tuple which causes the conclusion of the search, is used in the creation of the filter to give as strong a condition as possible. Using the other options no such tuple can be used because information is obtained concerning an entire relation.

The filter F5 is the last filter created in this example and the filters F1, F2, F3, F4, and F5 are shown in Fig. 4.6.

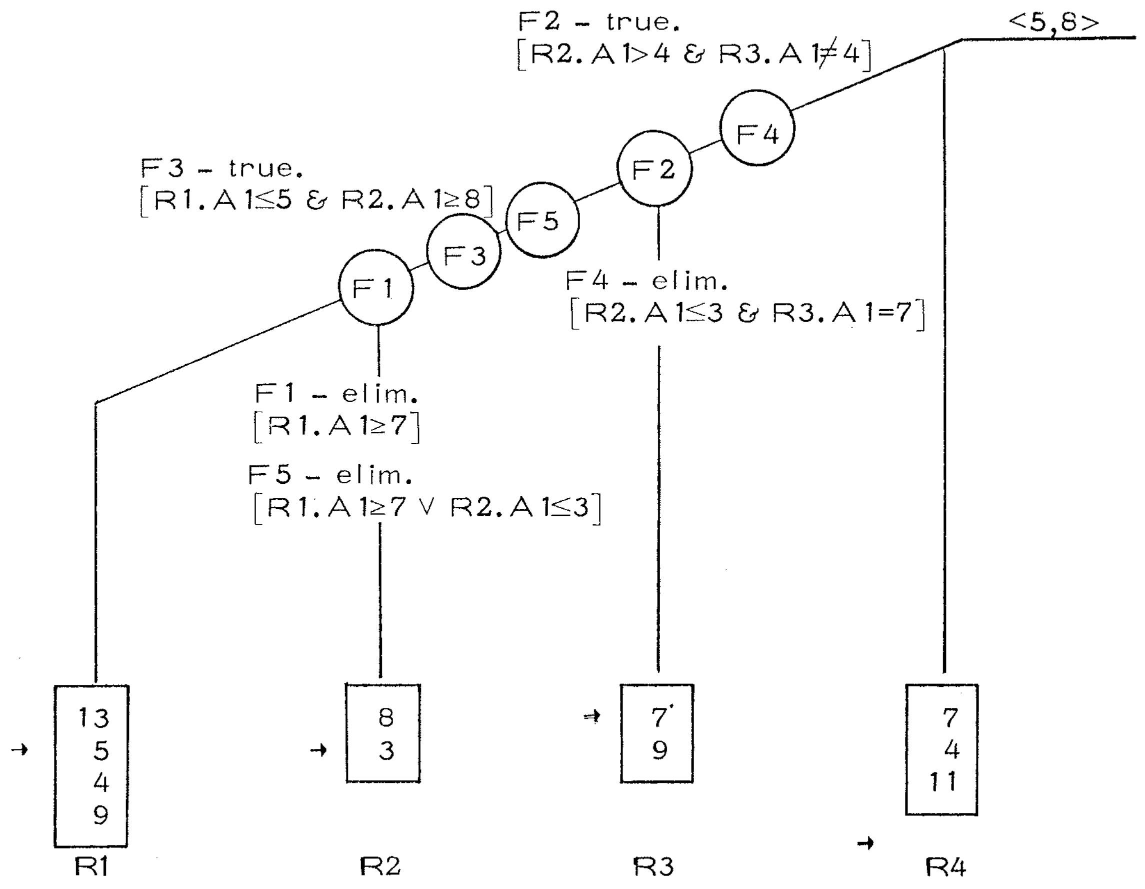


Figure 4.6

In Fig. 4.6 the filter F1 is redundant because it is contained in F5. Situations like this should be avoided although they are not always as simple as in Fig. 4.6.

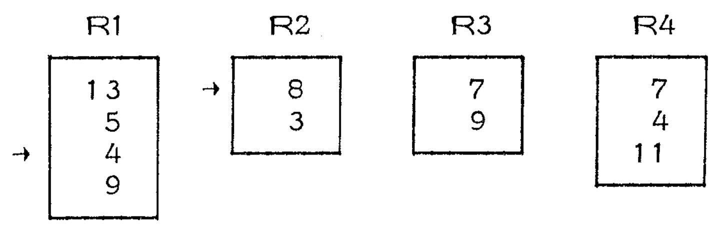


Figure 4.7

The execution of the query is continued and in the situation in Fig. 4.7 the true-filter F3 is used to find that the qualification is true for the tuple combination (<4>, <8>).

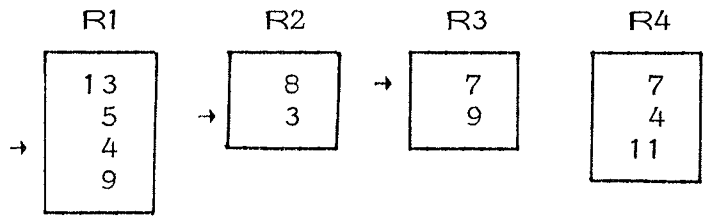


Figure 4.8

In Fig. 4.8 the elimination-filter F4 is used to finish the search of R3 thereby concluding the examination of the qualification for (<4>, <3>).

Both (<9>, <8>) and (<9>, <3>) satisfy the elimination-filter F1 and these combinations are therefore eliminated, too.

Example 4.1 should give a (very simplified) picture of where and how the feedback-optimization is used in TGR. In the rest of this section a more precise description is given of the various options and the structure and use of filters.

4.A Filters

In example 4.1 filters were constructed from (parts of) the qualification, perhaps using some of the current tuples. If R1, R2, R3 and R4 were large relations a great number of filters would be created which would mean a very big overhead in the algorithm. To avoid this, a strategy for the creation of filters is used differing slightly from the straightforward one used in example 4.1.

For instance, take a look at Fig. 4.4 and the filter $F3 = [T1.A1 \leq 5 \wedge T2.A1 \geq 8]$. If the relation R1 contained a tuple <6> then the filter $F = [T1.A1 \leq 6 \wedge T2.A1 \geq 8]$ is created at some stage of the algorithm. F has a structure similar to that of F3. The only difference is that the tuple <6> from R1 is current when F is constructed while <5> from R1 is current at the time F3 is constructed.

This is used in the following manner by the algorithm for execution of a particular query:

1) For each relation it is decided which options to set.

2) For each of these options a filter is created where the condition is only a template with holes to be filled with values from the current tuples.

3) Every time a filter is going to be created at execution time the only thing to be done is to write attribute values of some of the current tuples into a data area for the filter. This gives a new value group in the data area of the filter.

4) To find out if a combination T of tuples satisfies a filter the template is filled with values from one value group at a time to establish a condition C for T. This is continued until either C is true for the combination T i.e. the filter is satisfied, or until every value group has been inserted in which case the filter is not satisfied for T.

The precise definition of a filter can now be given.

A filter consists of the following parts:

a) Relation-part
The filter is checked for combinations of tuples from these relations.

b) Type
Elimination - or True-filter.

c) Condition
The template to be filled with values from d).

d) Data-part
A data area containing attribute values of tuples which are current tuples at some specific points of time (shown in section 4.D, tables 4.1 and 4.2).

From this it is seen that the time spent to construct filters is negligible since the construction of templates is done only once and the insertion in the data-part of values from current tuples is an extremely simple operation. Therefore the overhead introduced by the use of filters is concerned only with the checking of filters.

4. B Conjunctive Normal Form (CNF)

The query is required to be in CNF for two purposes:

1) To be able to use a simple algorithm [PECH76] to optimize the order of the relations in the nested loops.

2) To make it possible to create fairly simple rules for the construction of filters.

Further, CNF is used to have a simple means of finding the condition G_i to be checked when searching a relation R_i.

A precise definition of G_i can be given in the following way [PECH76]:

Let $Q = g_1 \wedge g_2 \wedge \dots \wedge g_k$, $k \geq 1$ where the g_j's are disjunctive expressions:

$$g_j = Y_1^j \vee Y_2^j \vee \dots \vee Y_{s_j}^j \text{ , } s_j \geq 1, \; 1 \leq j \leq k$$

where the Y_i^j's are simple expressions, i.e. either

$$R_l.A_c \text{ op } R_q.A_d \text{ or } R_l.A_c \text{ op } K \text{ ,}$$

op $\in \{=, \neq, <, >, \leq, \geq\}$, $1 \leq l, q \leq m$ and K is a constant.

Define

$$H_{g_j} = \{ i \mid \text{one or more domains of } R_i \text{ are referenced by } g_j \}$$

and

$$P_i = \{ g_j \mid i = \max(H_{g_j}) \}, \qquad 1 \leq i \leq n$$

where n is the number of different relations in Q. (The situation where one relation is searched more than once is not considered here to avoid the treatment of tuple-variables.)

Then

$$G_i = g_{p_1} \wedge g_{p_2} \wedge \ldots \wedge g_{p_m}$$

where

$$P_i = \{g_{p_1}, g_{p_2}, \ldots, g_{p_m}\}$$

(In the above definitions the order of the relations is assumed to be 1, 2, ..., n).

The use of CNF causes no loss in the descriptive power of the language since every query can be converted to CNF. The conversion is straightforward [Chan73] but may introduce several copies of parts of the query. It is a trivial matter to keep track of these copies thereby avoiding more than one evaluation of the same condition.

Example 4.2

Again, let us look at Example 4.1:

$$\begin{aligned} Q1 = \text{GET } (R1.A1,\ R2.A2) :\\ \forall R3\ \exists R4\ (R1.A1 < R3.A1 \wedge R2.A1 > R4.A1 \wedge R3.A1 \neq R4.A1)\end{aligned}$$

Here no condition is used searching R1 and R2, i.e. G_1 = true and G_2 = true. By the time of the search of R3 the expression $R1.A1 < R3.A1$ can be calculated and therefore $G_3 = R1.A1 < R3.A1$. Equivalently, $G_4 = R2.A1 > R4.A1 \wedge R3.A1 \neq R4.A1$.

4.C Rules for the construction of filters

Until now no query involving the logical operator $\vee$ has been treated. This is because it often complicates the construction of templates considerably. Option Asome and option Aall are especially complicated and the precise rules for these options will not be given here. Instead we will look at the rules for option B1some. (The precise rules for all options are given in [TGR78].)

Option B1some

Option B1some is used where G_i is not satisfied for any tuples in the relation R_i. The filter F is in this case constructed only from G_i since the knowledge achieved from the search of R_i concerns only R_i and G_i.

If

$$G_i = g_{i_1} \wedge g_{i_2} \wedge \dots \wedge g_{i_m}$$

where

$$g_j = Y_1^j \vee Y_2^j \vee \dots \vee Y_{s_j}^j$$

then

$$F = g'_{i_1} \wedge g'_{i_2} \wedge \dots \wedge g'_{i_m}$$

where

$$g'_j = Z_1^j \wedge Z_2^j \wedge \dots \wedge Z_{s_j}^j$$

where Z_r^j is formed from Y_r^j using the rules:

1) $Y_r^j = (R_k.A_p) \text{ op } (R_i.A_q) \quad k < i$ gives

$Z_r^j = (R_k.A_p) \text{ op}' (E_k.A_p)$ where

E_k is the current tuple of R_k; that is a hole is left in the template for the element $E_k.A_q$ and at execution time the value $E_k.A_q$ is put into the data-part of the filter when appropriate (i.e. G_i is false for every tuple in R_i).

op	op'
$=$	$=$
$\neq$	$=$
$<$	$\geq$
$>$	$\leq$
$\leq$	$\geq$
$\geq$	$\leq$

2) $Y^j_r = (R_k.A_p) \text{ op } (R_l.A_q)$ $k < i$ and $l < i$ gives

$Z^j_r = (R_k.A_p) \neg\text{op } (R_l.A_q)$

3) $Y^j_r = (R_k.A_p) \text{ op } K$ $k < i$, K is an arbitrary constant, gives

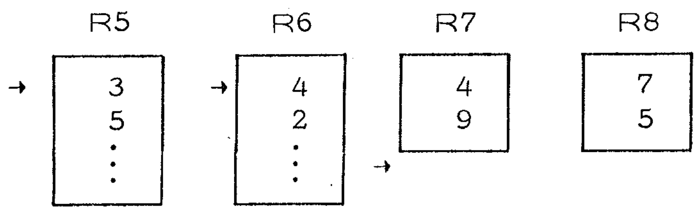
$Z^j_r = (R_k.A_p) \neg\text{op } K$

4) For other forms of Y^j_r the following applies.

If $G_i = Y^j_r$ then Z^j_r = false.
No filter can be constructed from this G_i, i.e. the filter is false for every tuple.

Otherwise Z^j_r = true.
The Y^j_r part of G_i does not contribute to the filter.

Example 4.3
Let us look at relations R5, R6, R7 and R8 and the situation



from the execution of the query

$$Q = \text{GET (R5.A1, R6.A1)}: \exists R7\ \exists R8\ (\\ \ldots\ldots\ldots \\ \wedge (R5.A1 = R6.A1 \vee R5.A1 \geq R7.A1) \wedge \\ \ldots\ldots\ldots \\ )$$

Here we have

$G_6 = \ldots.$
$G_7 = R5.A1 = R6.A1 \vee R5.A1 \geq R7.A1$
$G_8 = \ldots.$

The use of the rules for option B1some gives the template $[R5.A1 = R6.A1 \wedge R5.A1 \leq X]$ where the X indicates the "hole" for R5.A1, an attribute value of the current tuple for R5. In the above situation from the execution the current tuple of R5 is $<3>$ and the value 3 is therefore inserted in the data-part of the filter. Afterwards the filter is checked following the algorithm on p. 18 for every new combination of tuples from R5, R6.

In the use of option B1some the or-signs do not give very complicated rules. Instead the filter in many cases will be very weak, as is seen from example 4.3 where the $\vee$ in the query is converted to an $\wedge$ in the filter.

4.D Options

If a certain option is set, data is inserted into the data-part of the filter at situations shown by the following table 4.1 and table 4.2. (The tables deal with quantified relations only.)

The third column shows when the data area of the filter is filled, after searching the relation R_i. The order of relations is $1, 2, \ldots, n$, i.e. the filter is on the combination $R_1, R_2, \ldots, R_{i-1}$.

In table 4.1 R_i is quantified by an existential quantifier.

option	filter type	situation
Asome	true	G_i and the "rest" of the qualification (concerning only $R_{i+1}, \ldots, R_n$) are true for some tuple in R_i.
B1some	elimination	G_i is false for every tuple in R_i.
B2some	elimination	G_i is true for some tuples in R_i, but for all such tuples the "rest" of the qualification is not.

Table 4.1

In table 4.2 R_i is quantified by a universal quantifier.

option	filter type	situation
Aall	true	G_i and the "rest" of the qualification (concerning only $R_{i+1}, \ldots, R_n$) are true for every tuple in R_i.
B1all	elimination	G_i is false for a tuple in R_i.
B2all	elimination	There exists a tuple in R_i for which G_i is true but the rest of the qualification is not.

Table 4.2

The only situation where useful information is achieved about a relation R_i from the target list is when no tuple from R_i satisfies G_i. This situation is equivalent to the situation from option B1some for an existentially quantified relation and therefore this option is used for relations from the target list. No other options can be used because the qualification is evaluated only for a combination of tuples from all relations in the target list.

In section 4 the implementation in TGR of a feedback optimization method for arbitrary database queries was presented. A proposal is given in [ROTH72] for an extension of DAMAS to implement the feedback mechanism for queries with any number of unquantified and existentially quantified relations. As mentioned earlier the feedback optimization in DAMAS and TGR for two-variable queries are equivalent except for the use in TGR of filters; this postpones the time for the use of the feedback information. It means fewer searches but requires more administration and primary storage space.

In the implementations of the more general database queries mentioned there are some big differences, namely:

- In DAMAS feedback information is achieved for a single relation R and the appropriate actions are done for R immediately. This is completely equivalent to having a temporary filter on that relation in TGR. The filter lasts until a new search is started on R. Therefore all information is then lost in DAMAS. In TGR another line of action is chosen, namely to have filters on combinations of relations; the filter survives throughout the whole execution of the query.

- The use of the universal quantifier gives additional filters in TGR and also causes differences in the filtering conditions concerning the existential quantifier. The reason for this is that a universal quantifier in a query changes the deductions that can be made from a result of an existentially quantified relation.

- The conjunctive normal form of queries in TGR gives a systematic way of creating templates. Many special situations have to be handled in Rothnie's proposal, especially in connection with the $\vee$ operator.

Consequently, Rothnie's proposal for DAMAS does not use all the information found by searching relations. To reach this goal in TGR the use of filters instead of immediate actions as in DAMAS is necessary. Therefore more overhead and a greater need for storage space is introduced in TGR but we believe that this is more than counterbalanced by the additional number of saved searches.

A more detailed comparison between TGR and DAMAS is beyond scope of this paper but some more information about the performance of TGR is given in section 6.

5. STATUS OF IMPLEMENTATION

In this section we describe what has so far been implemented of TGR, and what we intend to implement in the near future.

Combining Fig. 2.1, Fig. 2.2 and Fig. 2.3 from section 2 gives a picture like Fig. 5.1, which shows the structure of a full database system using TGR.

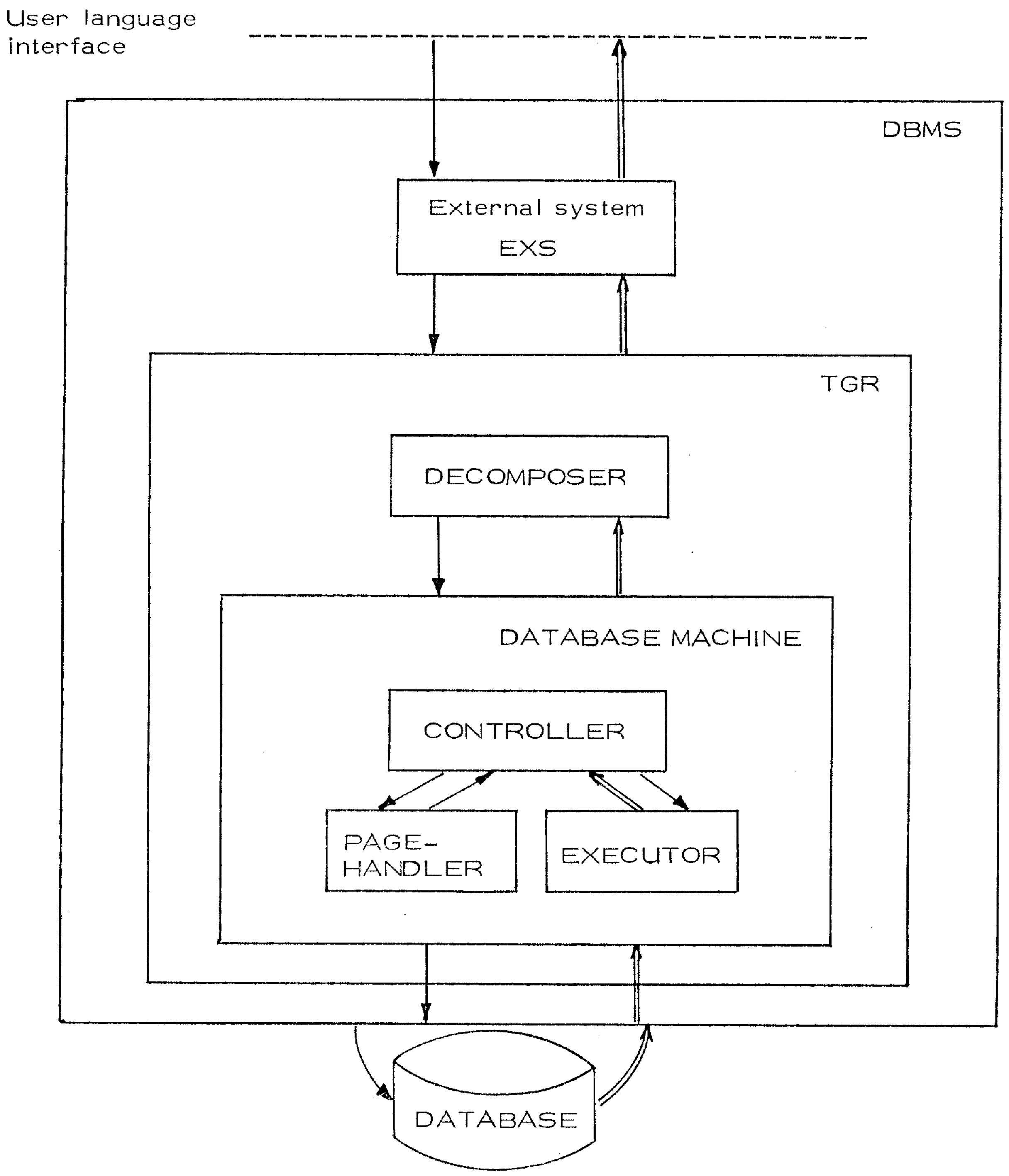


Fig. 5.1

To get a useful test vehicle, we have already implemented a small part of the external system, e.g. data definition facilities.

All the relational calculus interface ILT between EXS and TGR concerning retrieval operations is implemented, but only simple update commands are available.

The DECOMPOSER is implemented on RIKKE in BCPL as described in sections 2-4. The only options implemented at this point of time are option B1some and option C. However, most of the subroutines used for option B1some can be used for the other options too. Therefore the implementation of these options is straightforward and should soon be finished. No access paths are yet supported by TGR, i.e. tuples within a relation are stored randomly and no indices or other logical structures are supported. This is a drawback of the current implementation of TGR, using conventional disks for secondary storage, since this method gives poor performance for large relations. Anyhow, the implementation is very useful to test the feedback-optimization.

The CONTROLLER and PAGEHANDLER are implemented on RIKKE in BCPL as described in section 2.

Some parts of the EXECUTOR are microprogrammed on MATHILDA and the communication modules between RIKKE and MATHILDA are finished. In the present version of TGR a BCPL version implemented on RIKKE is being used until the microprogramming on MATHILDA is finished. The performance of the BCPL version of the EXECUTOR gives an idea of the performance of a final microprogrammed version as the factor of improvement in execution time usually lies between 10 and 100.

The present version of TGR is very useful as a test vehicle for the various optimization techniques to find the relative improvement of performance. It cannot be used to search in large relations because this would involve complete scans of these relations. To get better performance in this case too, one of the following lines of action can be taken:

1) Finishing the microprogramming on MATHILDA and further developing the system to implement various access paths for the relations. To support these access paths additional microprogrammed primitives should be implemented and the CONTROLLER should then translate one-variable expressions to a sequence of instructions utilizing the access paths on the relation. If no access paths exist for a certain relation, the usual approach is chosen. The DECOMPOSER also uses knowledge of the existence of access paths to find the optimal order of the relations.

2) To replace the DATABASE MACHINE by a special-purpose device like the "search processor" mentioned on p. 7. This storage device searches the entire database throughout a revolution and therefore no access paths are necessary.

Proposal 1) is chosen for the following reasons:

- The optimization done in TGR is very well suited for queries involving many relations whereas nearly nothing is done for queries concerning one relation. The performance of a "search processor" still is not good enough for queries involving many relations since the number of revolutions would be some fraction of the product of the sizes of the involved relations. This number would in most cases be too big to get acceptable performance but here for instance the feedback optimization would help considerably. Anyway, we believe that the optimized number of revolutions would in some cases still be too big for today's equipment thereby ruining demands on maximum response time. When an extremely fast bubble-memory with large capacity appears maybe the problem will be solved and proposal 2) above can be chosen.

- A characteristic of most existing database applications is that most of the queries are for a single tuple of a single relation. It therefore seems a waste to search the entire database unless you have an extremely fast secondary memory [SENK78].

- The price of a fixed head disk as used in many database machine projects [OZKA75], [SU75], [LIN76], is still too high to be really competitive with the price of one-head disks and the price seems to be remaining high.

  Bubble memories or CCD's could be used instead and are used in some projects [CHAN78] but they are still too expensive and of too small capacity and speed. Therefore 2) above cannot yet be chosen. However, the development of cheaper and cheaper bubble memories with growing capacity shows that there is a future in this area for bubble memories. In a few years database machines with a large capacity bubble memory should be commercially available.

- In some database machine projects (the search processor [LEIL78] and the Data Base Computer (DBC) [BANE78]) another strategy is used, namely to read all tracks of a cylinder of a normal moving head disk in parallel. Furthermore the disk controller is extended with logic for each track to provide content-addressable search. In the DBC proposal this is used together with clustering in cylinders of related tuples to achieve a performance enhancement over existing database systems; this looks very promising.

  The current RIKKE-MATHILDA configuration does not support this strategy. However, if hardware were available these ideas could easily be incorporated in TGR.

An appropriate access path to support in TGR is an indexed organization because this can give the wanted direct access to the tuples. It is not yet decided if other access paths should be supported too. Therefore the future plans for the further development of TGR are:

- To design and implement subroutines to support an inverted file organization and incorporate these new ideas into the DECOMPOSER, the CONTROLLER and the EXECUTOR.

- To finish the micro-programming of the EXECUTOR primitives including the new primitives for the handling of indexes.

The use of an inverted file organization does not affect the feedback optimization. After the search of a relation the feedback optimization needs to know only if a tuple exists satisfying a certain condition. The way this information is achieved does not matter.

Anyhow the use of indexes affects the selection of options because the execution time for a query is decreased. The overhead for an option in the feedback optimization method is the same whether or not indexes are used. Therefore the cost/benefit calculation for the selection of options (see section 6) has to take into account the possible use of an index.

6. PERFORMANCE ANALYSIS, EXAMPLES

TGR is an experimental system and is only used as such, i.e. the database is of an experimental nature too. Therefore no statistical information about the performance of TGR used for practical purposes is available. In this section statistical information about the performance of TGR for queries on a small experimental database is presented. As mentioned in section 5, TGR in its current implementation performs poorly for queries on large relations so to test the feedback optimization only small relations are used. In example 6.1 program generated relations are used and in example 6.2 very small relations concerning Date's supplier-part-project example [DATE77] are used. Throughout the section only option B1some and option C are considered because these options are the only ones currently implemented.

Example 6.1

Three program generated relations are used. Each relation consists of two attributes which contain integer values generated randomly in different intervals. The intervals are shown in table 6.1.

Relation	Interval of attribute 1	Interval of attribute 2	Number of tuples
R1	1 - 128	51 - 178	1400
R2	1 - 64	1 - 1048	1000
R3	1001 - 1512	1 - 128	1000

Table 6.1

The query executed on this database is

$$Q1 = \text{GET } R1.A1,\ R1.A2,\ R2.A2:$$
$$\exists R3\ (R1.A1 > 100 \wedge R1.A2 < R2.A1 \wedge R3.A1 < R2.A2)$$

The possible options to set for Q1 are:

- option C (elimination of duplicates in target list)
- option B1 for R2 (creates a filter F1 on R1)
- option B1 for R3 (creates a filter F2 on R1, R2)

The filters F1 and F2 have the following structures (see rules in section 4):

F1 = [R1.1 ≥ X] where X is the first attribute of a current tuple of R1.
F2 = [R2.2 ≤ Y] where Y is the second attribute of a current tuple of R2.

The results of using the various options are shown in table 6.2.

Relative time measurements are used in the tables because these give the easiest way of comparing results from the use of different options.

The execution time for Q1 without any options is set to 1000.
(The figures in the tables are experimental results and therefore only approximate values.)

Options	none	C B1 on R2 B1 on R3	B1 on R2 B1 on R3	C B1 on R3	C B1 on R2
Entire algorithm	1000	18	18	105	916
Computation of condition for tuples	193	5	5	36	208
Checking of filters	0	1.5	1.5	1.4	0.1
Searching R1	0.4	0.4	0.4	0.4	0.4
Searching R2	95	9	9	95	11
Searching R3	902	6	6	7	901

Table 6.2

(The remaining three possibilities of combination of options do not give additional information and are therefore omitted.)

Explanation of the rows in the table:

- Entire algorithm: The execution time for the query.

- Computation of condition for tuples: The total time spent to find out if conditions (G_i's) are true or false for the tuples examined in the query. The time used to compute filter-conditions is not included.

- Checking of filters: The total time spent to check filters.

- Searching "relation": The time spent to search "relation". Included are: time used for disk access, finding tuples on a page and computation of conditions and checking of filters for the tuples in "relation".

Table 6.2 shows a very big improvement of execution-time by the use of the feedback-optimization. Option B1 on R3 gives a very big improvement whereas option B1 on R2 gives a somewhat smaller, but in any case both options ought to be set. Option C does not give any visible additional effect, neither positive nor negative.

The cost of checking the filters is very small compared with the decrease in time spent on other parts of the algorithm. In this example the storage requirement for the data-part of the filter is negligible; why this is not always true is discussed later in this section.

In example 6.1 both options B1 and B2 could be used with advantage but this is <u>not</u> always the case.

In example 6.2 we will look at a database query for which some options should be set and others should not.

The database is slightly more realistic than in example 6.1, namely the supplier-part-project example from [DATE77]. The query considered is from exercise 5.16 in the same book. Only the relations P and SPJ are used and the definitions of these are:

P (P#, Pname, Color, Weight), i.e. a tuple (x, name, zzz, y) indicates that there exists a part with name "name", part number X, color zzz and with weight Y .

SPJ (S#, P#, J#, QTY), i.e. a tuple (x, y, z, v) indicates that the supplier with number x supplies part number y to project number z in the quantity v.

The purpose of the query Q2 is to find S# values for suppliers supplying at least one part supplied by at least one supplier who supplies at least one red part, i.e.

```
Q2 =     range SPJY    SPJ
         range SPJZ    SPJ
         GET  SPJ.S# :
            ∃ SPJY  ∃ SPJZ  ∃ P (
                                 SPJY.P# = SPJ.P# ∧
                                 SPJZ.S# = SPJY.S# ∧
                                 P.P# = SPJZ.P# ∧
                                 P.Color = 'red'
                                 )
```

The possible options to set for this query are:

- option C
- option B1 for SPJY (creates a filter F1 on SPJ)
- option B1 for SPJZ (creates a filter F2 on SPJ, SPJY)
- option B1 for P (creates a filter F3 on SPJ, SPJY, SPJZ)

The filters F1, F2 and F3 have the structures:

F1 = [SPJ.P# = A]	where A is the P# attribute of a current tuple of SPJ.
F2 = [SPJY.S# = B]	where B is the S# attribute of a current tuple of SPJY.
F3 = [SPJZ.P# = C]	where C is the P# attribute of a current tuple of SPJZ.

The results of using the various options for Q2 are shown in table 6.3.

	1	2	3	4	5	6	7	8	9
Options	No	C B 1 on SPJY B 1 on SPJZ B 1 on P	C B 1 on P	B 1 on P	B 1 on SPJY	B 1 on SPJZ	B 1 on SPJY B 1 on SPJZ	B 1 on SPJY B 1 on SPJZ B 1 on P	C
Entire algorithm	1000	129	126	249	1007	1008	1022	252	350
Computation of condition for tuples	428	31	31	61	432	440	431	60	143
Checking of filters	0	22	21	51	1	2	2	55	0
Searching SPJ	1	1	1	1	1	1	1	1	1
Searching SPJY	26	18	17	27	27	27	28	27	16
Searching SPJZ	125	59	59	129	126	126	127	134	53
Searching P	800	19	18	24	806	806	812	25	262

Table 6.3

In contrast to example 6.1 option C here gives a quite good improvement and option B 1 on P performs especially well. The fastest execution is reached using these two options (column 3), whereas the use of option B 1 on SPJY and option B 1 on SPJZ slows down the execution (columns 5, 6 and 7). This is because no tuples at all are eliminated using these options and therefore only the overhead from the checking of the filters is introduced.

From examples 6.1 and 6.2 it can be deduced that a reasonable improvement of performance can be achieved if the right options are selected. The cost of checking the filters has been very small compared with the benefit. We will now look in a little more detail at what is actually the cost of using a filter both in terms of time and space. Let us use an example involving relations R1, R2, R3 and R4 in the indicated order where a filter F is used on the combination R1,R2.

- Time: F is checked for every combination of tuples from R1,R2. A check of F requires a calculation of the condition of F for each value group in the data area. The number of value groups increases thorughout the execution therefore no precise expression can be given for the cost of checking a filter.

  For every combination of tuples from R1,R2 eliminated by the use of F a search of R3 and perhaps some searches of R4 are saved. (What is saved is dependent on the option. E.g. for option B1some only a search of R3 is avoided.) Therefore an approximation of what is saved for every elimination can be calculated from information about the type and structure of the filter, sizes of relations and the distribution of values in the relations. Closed expressions for this calculation is given in [ROTH72] for two-variable queries and also approximations for the cost are given. We have not done the equivalent work for arbitrary queries in TGR. To compare cost and benefit in TGR a very straightforward procedure can be used:

  Both the cost and benefit can be measured in terms of the number of tuples for which a condition is calculated. The cost is calculated using probability theory on the G_i's. The calculation of the benefit is done in a similar way. First making an approximation of the expected number of value groups again through the use of G_i's. Then by examination of the filter-condition in order to estimate the probability for it to be satisfied. The complexity of the condition (filter or G_i) is taken into account too, by modifying the number of tuples with a factor describing the complexity. To make the procedures feasible some assumptions have to be made, e.g. that values in relations are randomly distributed (which is certainly not always true).

- Space: The use of storage space is measured by the number of value groups times the size of a value group. The number of value groups is some fraction of the product of the sizes of the relations which the filter covers. If this number is big a lot of space is used and also a lot of time is used for the checking of the filter. It is seen that the analyses of time and space are heavily interconnected and if the use of time is reasonable then the use of space would be too, relative to the sizes of the involved relations.

A proposal for a strategy for automatic selection of options

1. Compile-time selection.
   Using the above mentioned method a cost/benefit calculation is made for every possible option and the appropriate options are set.

2. Runtime selection.
   Throughout the execution, the performance of the filters is watched by comparing the number of eliminated (or true) tuples with the number of checked tuples. If too few tuples are eliminated the option is cleared.

Since the calculation from 1) is quite inaccurate the results could be tested during the execution for some of the options. This would be done by setting the options for a short while and then look at the results as 2) above. A more precise selection of options would be achieved then but also more overhead is introduced.

The automatic selection of options has not yet been implemented in TGR but it is recognized that this aspect of the feedback optimization is about the most important to get good performance. This is seen from examples 6.1 and 6.2 where some options had to be selected to get acceptable performance (option B1 on R3 in example 6.1 and option B1 on P in example 6.2), the selection of others were more or less unimportant and some options should not be selected.

In this section two examples have been used to give the reader an idea of the performance of TGR using the feedback optimization. Tables 6.2 and 6.3 gave an impression of which options to select for particular queries. Besides, the relative time measures shown in the tables can be used to see what is the bottle-neck of the system. A lot of time is spent evaluating conditions (from the qualification or a filter) for tuples. A great part of this time will be saved as soon as the microprogramming of the database primitives is finished. Further, experiments have shown that TGR is not at all disk bound. Therefore, microprogramming would be a step towards a balanced system. Hopefully, a completely balanced system will be reached when the improvements of TGR, proposed in section 5, are carried out.

7. CONCLUSIONS, FURTHER RESEARCH

The TGR project was started in 1977 for three purposes as mentioned in the introduction, namely design and implementation of

1) An intermediate language.
2) A microprogrammed database machine.
3) Optimization features.

All three aspects are discussed in this paper but most emphasis is placed on 3) where only the feedback-optimization is presented.

It is shown that the feedback-optimization gives a great improvement of performance _if_ the right options are selected. How to select the right options is only outlined since this subject is not yet fully investigated. Some research should be made in this area in the future.

The feedback-optimization is very well suited to complex queries involving many relations but it is _not_ any help for single relation queries. Therefore feedback-optimization cannot be used as a stand-alone optimization method. It should be used in a database environment where the information need is changing a lot and is of a very complex nature. Other optimization methods should be available especially for single relation queries which appear frequently in database systems.

Another way to improve performance of database systems is by the use of microprogramming or construction of specialized hardware. In TGR a number of database primitives are designed forming a database machine. The database primitives are intended for microprogramming but the implementation is not yet fully finished. The current implementation of TGR therefore uses an implementation in a high level language (BCPL). This implementation does not perform very well for large relations and a number of improvements are proposed in this paper:

- Design and microcoding of primitives to support an inverted file organization. (Relations are stored randomly and no logical storage structures are supported yet.)

- Conclusion of the microprogramming of the existing database primitives.

It is our hope that the investigation and implementation of these primitives will eventually be used as a guideline for the construction of specialized hardware which can be used as a backend processor in a database system with a structure like that of TGR (replacing the EXECUTOR of the Database Machine, see e.g. Fig. 5.1).

As described in section 5 another step could be taken to incorporate specialized hardware into the system, namely to replace the Database Machine (Fig. 5.1) by for example the "search processor" [LEIL78] or equivalently supporting a one-relation interface. In this kind of system the feedback optimization would be a very big help, in fact nearly indispensable. However, we believe that the performance of this kind of system is still not good enough to be competitive to available <u>commercial</u> systems. The reason for this is that e.g. the fixed head disk as used in the "search processor" is of too low capacity and speed and too expensive to be used evaluating multivariable queries for very large databases. But as soon as the price gets reasonable for bubble memories or CCD's with large capacity and speed then this line of action will be very suitable.

Acknowledgements

I very much appreciate the help I have received from my supervisors John Owlett and Brian Mayoh concerning the formulation and discussion of contents. Also thanks to the Madsen twins, Birgitte and Erling, together with whom I made the initial implementation of TGR, and to Ib Holm Sørensen for many valuable discussions.

References

[ASTR76] : Astrahan, M.M. et al.
System R - Relational Approach to Data Base Management.
ACM TODS vol. 1, 1976.

[BANE78] : Banerjee, J. & Hsiao, D.K.
Performance Study of a Database Machine in Supporting Relational Batabases.
Proc. 4th Int. Conf. VLDB, Berlin 1978.

[CHAN73] Chang, C.L., Lee, R.C.T.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, 1973, s. 37-39.

[CHAN78] : Chang, H.
On Bubble Memories and Relational Data Base.
Proc. 4th Int. Conf VLDB, Berlin 1978.

[CODD70] : Codd, E.F.
A Relational Model for Large Shared Data Banks.
CACM, vol. 13.

[CODD71] : Codd, E.F.
A Data Base Sublanguage Based on the Relational Calculus.
Proc. of the 71 ACM SIGFIDET Workshop.

[DATE77] : Date, C.J.
An Introduction to Database Systems.
2nd edition, Addison-Wesley 1977.

[KORN75] : Kornerup, P. & Shriver, B.D.
A Description of the MATHILDA Processor.
DAIMI PB-52, 1975. Computer Science Department, Aarhus University, Denmark.

[KRES75] : Kressel, E. & Sørensen, I.H.
The First BCPL-System on RIKKE-1.
DAIMI MD-17, 1975. Computer Science Department, Aarhus University, Denmark.

[LEIL78] : Leilich, H.O. & Zeidler, H.C.
A Search Processor for Data Base Management Systems.
Proc. 4th Int. Conf. VLDB, Berlin 1978.

[LIN76] : Lin, C.S. et al.
The Design of a Rotating Associative Memory for Relational Database Applications.
ACM TODS vol. 1 no. 1, 1976.

[OZKA75] : Ozkarahan, et al.
RAP - An Associative Processor for Data Base Management.
AFIPS vol. 44, 1975.

[PECH76] : Pecherer, R.M.
Efficient Exploration of Product Spaces.
ACM SIGMOD 1976.

[ROTH72] : Rothnie, J.B.
The Design of Generalized Data Management Systems.
Ph.D. Dissertation. Dept. of Civil Engineering, MIT 1972.

[ROTH74] : Rothnie, J.B.
An Approach to Implementing a Relational Data Management System.
ACM SIGMOD 1974.

[SENK78] : Senko, M.E.
Private communication, 1978.

[STAU74] : Staunstrup, J.
A Description of the RIKKE-1 System.
DAIMI PB-29, 1974. Computer Science Department, Aarhus University, Denmark.

[STON76] : Stonebraker, M. et al.
The Design and Implementation of INGRES.
ACM TODS vol 1 no. 3, 1976.

[SU75] : Su, S.Y.W. & Lipovski, G.J.
CASSM - A Cellular System for Very Large Databases.
Proc. Conf. VLDB 1975.

[TGR78] : Madsen, B., Madsen, E. & Clausen, S.E.
Design og implementation af et relationelt databasesystem
specielt med henblik på effektiv udtrækning af data.
Speciale-opgave 1978. Computer Science Department,
Aarhus University, Denmark.

Appendix A

A description of the RIKKE-MATHILDA system

This appendix gives a very superficial description of the RIKKE-MATHILDA system. More detailed information can be found in [STAU74], [KRES75] and [KORN75]. The system was designed and built at the Computer Science Department, Aarhus University.

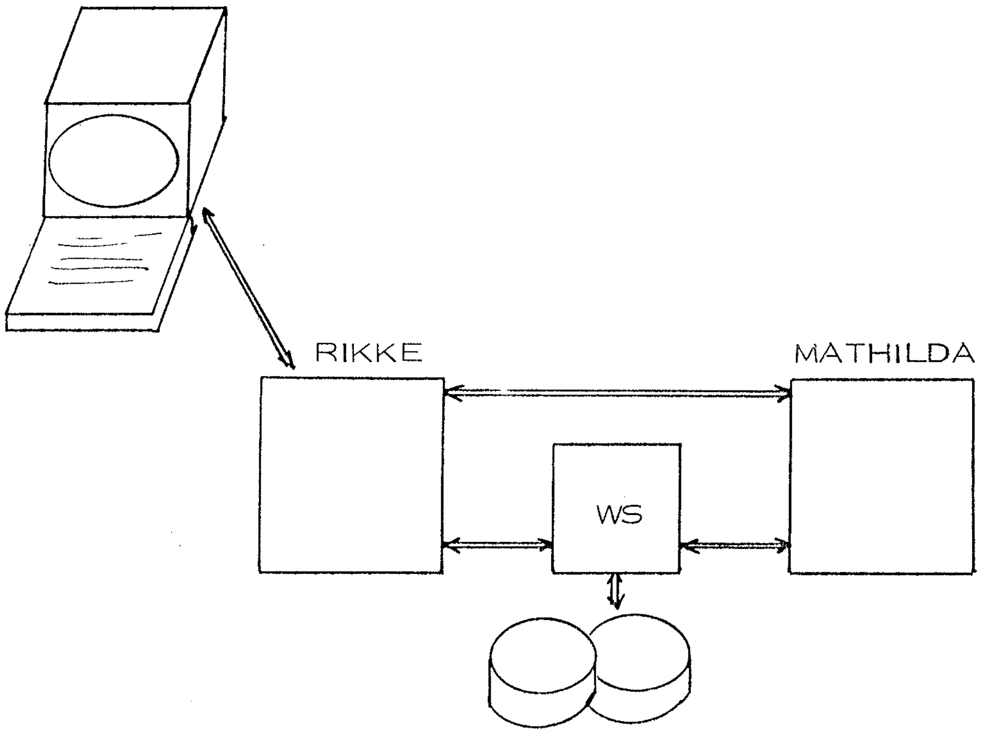


Fig. A. 1

RIKKE is a microprogrammable processor [STAU74] on which a virtual stack machine is realized. A single-user operating system, the RIKKE BCPL-system [KRES75], runs on this machine.

Wide Store (WS) is considered to be the main memory of the computer system. It is a sharable 256K byte memory which can be accessed by as much as 4 independent physical processors. In the configuration shown in Fig. A. 1 only 3 processors are connected to WS, namely RIKKE, MATHILDA and the disk controller.

MATHILDA is a microprogrammable processor [KORN75] which serves as an external functional unit for the BCPL-system. Microcoded routines can be called from and executed concurrently with BCPL programs on RIKKE.

The secondary storage used in the RIKKE-MATHILDA system consists of two 9.6 Mbyte disk units.

	1	2	3	4	5	6	7	8	9
Options	No	C B1 on SPJY B1 on SPJZ B1 on P	C B1 on P	B1 on P	B1 on SPJY	B1 on SPJZ	B1 on SPJY B1 on SPJZ	B1 on SPJY B1 on SPJZ B1 on P	C
Entire algorithm	1000	129	126	249	1007	1008	1022	252	350
Computation of condition for tuples	428	31	31	61	432	440	431	60	143
Checking of filters	0	22	21	51	1	2	2	55	0
Searching SPJ	1	1	1	1	1	1	1	1	1
Searching SPJY	26	18	17	27	27	27	28	27	16
Searching SPJZ	125	59	59	129	126	126	127	134	53
Searching P	800	19	18	24	806	806	812	25	262