Avdelning, Institution, fakultet
Division, department, faculty

Department of Computer and Information Science

Institutionen för datavetenskap

ISBN:	
ISSN:	
Rapportnr: Report no:	LiTH-IDA-Ex9325
Upplagans storlek: Number of copies:	
Datum: Date:	930803

Projekt:
Project:

Titel:
Title:

Randomized Optimization of Object Oriented Queries in a Main Memory Database Management System

Författare:
Author:
Joakim Näs

Uppdragsgivare:
Commissioned by:

Dnr:
Call no:

Rapporttyp:
Kind of report:
[✓ Examensarbete/Final report]
Delrapport/Process report
Reserapport/Travel report
Slutrapport/Final report
Övrig rapport/Other kind of report

Rapportspråk:
Language:
Svenska/Swedish
[✓ Engelska/English]
____

Sammanfattning (högst 150 ord):
Abstract (150 words)

This thesis investigates the behavior of different algorithms when optimizing Object SQL-queries in an object oriented database management system called WS-IRIS (WorkStation-IRIS). Query optimization is a combinatorial optimization problem which makes exhaustive search impossible when the query size grows. This has led to a new approach to query optimization, namely the use of randomized algorithms. In this thesis a number of randomized algorithms are applied to optimize queries in WS-IRIS. A comparison of the algorithms has been done and the combination of the Iterative Improvement algorithm and the Sequence Heuristics algorithm showed the best performance. This report also shows the importance of good cost estimates for query optimization and some improvements to the WS-IRIS cost model are presented.

Nyckelord (högst 8):
Keywords (8):
Query optimization, randomized algorithms, Object Oriented database, Object SQL

Bibliotekets anteckningar

Randomized Optimization of Object Oriented Queries in a Main Memory Database Management System

Joakim Näs

Tel. +46 910 51832
E-mail: Joakim.Nas@sa.erisoft.se
Supervisor and examinant : Tore Risch

Abstract

This thesis investigates the behavior of different algorithms when optimizing Object SQL-queries in an object oriented database management system called WS-IRIS (WorkStation-IRIS). Query optimization is a combinatorial optimization problem which makes exhaustive search impossible when the query size grows. This has led to a new approach to query optimization, namely the use of randomized algorithms. In this thesis a number of randomized algorithms are applied to optimize queries in WS-IRIS. A comparison of the algorithms has been done and the combination of the Iterative Improvement algorithm and the Sequence Heuristics algorithm showed the best performance. This report also shows the importance of good cost estimates for query optimization and some improvements to the WS-IRIS cost model are presented.

Keywords:
Query optimization, Randomized algorithms, Object oriented database, Object SQL.

Contents

1 Introduction

Query optimization is a combinatorial optimization problem. It is an expensive process, mostly because the number of alternative execution plans grow exponentially with the size of the query. The task of a query optimizer is to minimize the execution time for a given query in a database management system (DBMS). A query consists of a number of conditions that the result of the query has to satisfy. The optimization is done by making permutations of the conditions in the query. For each permutation of the query the execution time can be estimated according to a cost function. The permutation with the lowest cost is assumed to be optimal. Facts that effects the execution time (the cost) are the size of the query, the size of the addressed relations used in the query and the occurrence of indexes. The execution time of a query is highly dependent on the execution plan used. One plan may be thousands of times faster than a less optimal plan. A query passes through different phases in a DBMS, figure 1 shows the normal course for a query.

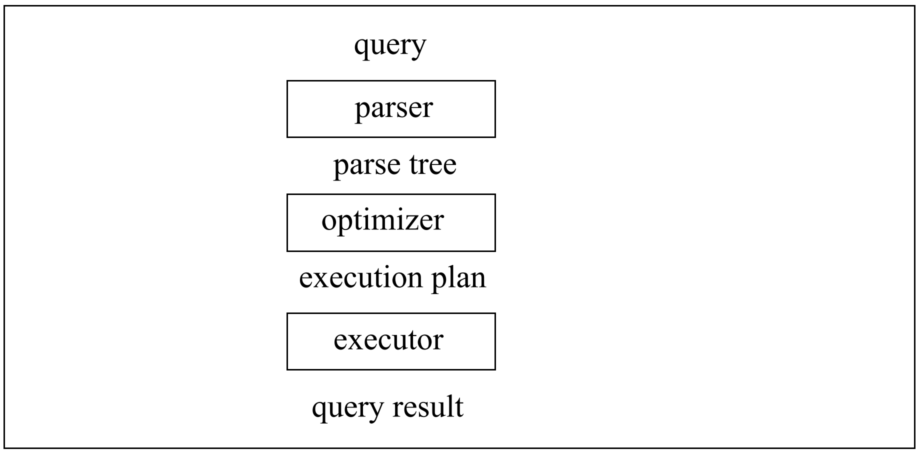


Figure 1. Translation steps in a DBMS.

The ordinary approach to query optimization has earlier been exhaustive search, a method that always returns the best solution. Today, when the need of optimizing large join queries have become high, the exhaustive search is no longer an useful method because of the time requirement of the algorithm. Examples of applications that creates large queries are knowledge based systems and object oriented database systems like WS-IRIS.

The difficulties in optimizing large database queries have led to a new approach, namely the use of randomized algorithms. This kind of algorithms have earlier successfully been used to other combinatorial optimization problems. This thesis examines the use of randomized algorithms in a main memory based DBMS called WS-IRIS. A number of randomized algorithms were compared to each other and to the existing optimizers in WS-IRIS. The outcome of the test shows that the combination of the Iterative Improvement algorithm and the Sequence Heuristics algorithm is superior to other optimization algorithms.

1.1 The work

The work that have done can be divided into four parts.

1. Several randomized algorithms have been implemented for query optimization in WS-IRIS.
2. A test have been done, comparing the randomized algorithms to each other finding out which one that has the best performance. The best algorithm has been implemented to be an optional optimization method in WS-IRIS, capable of optimizing every type of query allowed in WS-IRIS.
3. The best of the randomized algorithms have been compared with the two existing algorithms Ranksort and Exhaustive.
4. Some improvements to the cost model has also been made. This was not part of the originally definition of the work, but during the test of the algorithms it was proved that the current cost estimates in WS-IRIS did not have the precision needed for query optimization.

1.2 Background

This work was done at CAELAB(the laboratory for Computer Assistance in Engineering), a laboratory that is part of IDA(Department of Computer and Information Science), a department at Linköpings University. CAELAB has two directions of research:

- **Computer Support for Automation**
  Incorporating task level programming, realtime architectures for supervisory control, programming of autonomous manufacturing environments and realtime systems.
- **Engineering Information Management**
  Incorporates research on basic database technologies for engineering applications. Important concepts are distribution, heterogeneity, active databases and databases in realtime systems. Within this area of research the WS-IRIS architecture is used in several subprojects.

The work was done with the database group. At present CAELAB consists of nine persons and six of them are involved in projects with WS-IRIS. The work was initiated by professor Tore Risch, the head of the database group and the task was to implement a new optimization method in WS-IRIS. When the work started, two optimization methods were already present in WS-IRIS called *Ranksort* and *Exhaustive*. The motivation of a new method is that Ranksort sometimes comes up with poor solutions and Exhaustive is too slow when the query size grows. The task was to implement and test a new randomized algorithm called *Two-phase optimization* [Yann90].

1.3 Outline of the thesis

In this chapter an introduction is given. What have been done and the background of the work are described.

Chapter two describes the subject of query optimization in depth.

Chapter three describes the WS-IRIS DBMS, the query language WS-OSQL and the intermediate language ObjectLog, which is the language that the query optimizer works with.

In chapter four the existing algorithms in WS-IRIS are described.

In chapter five the randomized optimization algorithms are presented.

In chapter six the randomized algorithms are adopted for query optimization. The cost model is described and some improvements to the origin cost model are also presented.

Chapter seven presents the test of the algorithms. At first a description is given of how the test database and the test queries are generated. Finally the results are presented in a number of tables and graphs.

In chapter eight some implementation details of the algorithms are described.

Chapter nine contains a summary of the work and suggestions of future work are discussed.

Finally a reference list, an index and some appendixes conclude the thesis.

2 Query optimization

A query optimizer has two tasks. One is to produce a permutation of the query that has a short execution time. The other task is to make sure that the solution is a permutation that can be executed by the DBMS. The savings in execution time made by the optimizer is often substansial and appendix C shows an example of the need of a well performing query optimizer in a DBMS. Since a random query optimizer cannot guarantee the solution to be optimal some other criteria must be used to determinate the quality of the optimizer. The common known criteria in query optimization is based on the principle that a query optimization algorithm is considered to be good in practice if *it performs well on the average and very rarely performs poorly.*

2.1 Join methods

The join operator, denoted by $\bowtie$, is used to combine related tuples from two relations into single tuples. This operation is very important because it allows us to produce relationships among relations. The general form of a join operator on two relations $R(A_1,A_2,..,A_n)$ and $S(B_1,B_2,..,B_m)$ is:

$$\boxed{R \bowtie_{\text{<join condition>}} S}$$

The result of the join is a relation $Q(A_1,..,A_n,B_1,..B_m)$. Q has one tuple for each combination of tuples, one from R and one from S, whenever the combination satisfies the join condition.

The join operator is one of the most time consuming operations in query processing. The problem of optimizing a query is to select the join order in which the query is to be executed. Most of the join operators in queries are equi-join (the join condition is in the form A=B) and it is equi-join that has been used in the test of the algorithms presented in this thesis.

A	B
45	2
3	5
23	10
12	13

$\bowtie_{A=C}$

C	D	E
12	1	2
4	69	1
45	2	6

$=$

A	B	C	D	E
45	2	45	2	6
12	13	12	1	2

Example 1: A two way equi-join

The type of join shown in example 1 is a two way equi-join (an equi-join with two operands). The two way join is the most common used join in database management systems including WS-IRIS.

A join can be done in a number of different ways, some of them are presented here. When the different join methods are described the following are given, two relations R and S are to be joined with the join columns A and B. The join formula will be:

$$\boxed{R \bowtie_{A=B} S}$$

- **Nested-loop join**
  For each tupel r in R retrieve every tuple s from S and test whether or not the two tuples satisfy the join condition r[A]=s[B].
- **Access structure**
  With the use of an access structure to retrieve the matching tuples a more efficient join can be done. If an index or a hash key exists for one of the two join attributes, say B of relation S, each tuple r in R can be retrieved one at a time and then use the access structure to retrieve directly all the matching tuples s from S that satisfy r[A]=s[B].
- **Sort-merge join**
  If the tuples of R and S are physically sorted by value of the join attributes A and B, the join can be implemented in the most efficient way possible. The two relations are scanned in order of the join attribute, matching the tuples that have the same values for A and B. When using this method, the tuples of each relation are scanned only once, each for matching with the other relation.
- **Hash join**
  The tuples of relation R and relation S are hashed to the same hashtable using the same hashfunction on the join attributes A of R and B of S as hash keys. A single pass through each relation hashes the tuples to the hash table buckets. Each bucket is then examined for tuples from R and S with matching join attribute values to produce the result of the join operator.

In WS-IRIS the nested-loop join is used, unless there is an index on the join column in the relation then the access structure method can be used.

2.2 Join processing tree

In any database system the query optimizer has to come up with a processing strategy in some form. A strategy can be viewed as a *Join processing Tree* (JT). This is a tree in which each internal node is a join node and each leaf node is a base relation. Each join node represents the operation of joining the operand relations and it also represents the result of the join. The query optimization problem is to find the JT with the lowest cost. There exist many different kinds of JTs. In query processing the join operator is often thought of as a binary operator. This give us the *Binary Join processing Tree*(BJT), which is a JT in which each join operator has exact two operands. A special kind of BJT is the *Linear Join processing Tree* (LJT). In a LJT at most one of the operands can be an intermediate relation. An intermediate relation is the result of a join and is represented by an internal node in the tree.

Most join methods distinguish the two operands from each other, one being the outer relation and the other being the inner relation. The outer relation is the left child of a node and the inner relation is the right child. An *Outer Linear Join processing Tree* (OLJT) or left-deep tree is a LJT which inner relations always are base relations. Another type of LJT is the *Pipelined Join processing Tree* (PJT). A PJT can be thought of as an OLJT where the result is directly generated from the base relations without creating any intermediate relations. Figure 2 shows the structure of different JTs.

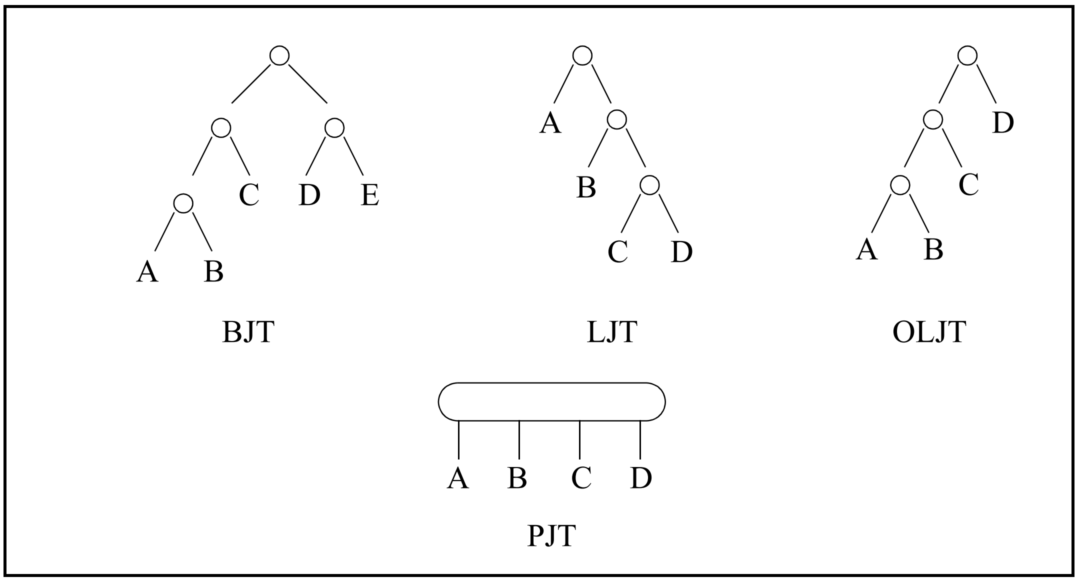


Figure 2. Join processing Trees

WS-IRIS has only one kind of JT and that is PJT. The PJT is especially suited for the nested-loop join method, which is the join method used in WS-IRIS.

3 WS-IRIS

WS-IRIS(WorkStation-IRIS) is an object oriented database management system. WS-IRIS is used as an experimental DBMS by the database group at CAELAB. One feature in WS-IRIS is that it keeps the entire database in main memory, time consuming disc accesses can thereby be limited. The query language used in WS-IRIS is WS-OSQL, an object oriented variant of SQL(Structured Query Language). WS-IRIS distinguish three types of functions, stored functions that are tables in the database, derived functions that are functions compound by other functions and foreign functions that are functions written in some foreign language like C or Lisp. The query optimizer works with a language called ObjectLog. In WS-IRIS WS-OSQL functions compile into ObjectLog programs as described in section 3.2.

3.1 WS-OSQL

WS-OSQL(WorkStation-Object Structured Query Language) is the query language in WS-IRIS and it is a superset of OSQL in Iris [Fish89]. WS-OSQL is in detail described in *WS-IRIS User's Guide* [Risc93a] and for the interfaces to C and Lisp see *WS-IRIS Advanced Programmer's Manual* [Risc93b]. In the following sections some features in WS-OSQL will be described. Some of them will effect the ObjectLog program and thereby also the query optimizer.

3.1.1 Features in WS-OSQL

- **Foreign functions**
  A foreign function is a function defined in some other language than WS-OSQL and can be called by WS-OSQL functions. In WS-IRIS a foreign function can be written in either C or Lisp.
- **Late binding**
  WS-IRIS supports late binding of overloaded functions where the overload resolution is done at run time instead of compile time.
- **Nested queries**
  A select statement can be part of another select statement. The inner select statement is called a subquery and it returns a bag of tuples as the result.
- **Recursive functions**
  WS-IRIS supports a limited class of recursive functions, only recursive functions that call themselves recursivly with all arguments to the call bound are handled.
- **Aggregation operators**
  An aggregation operator is a function that treats some of its arguments as a bag of tuples. The operator sum is an example of an aggregation operator.
- **Second order functions**
  Second order functions are allowed in WS-OSQL. This means that a function symbol can be used as a parameter to other functions.

- **Procedures**
  A procedure is a WS-OSQL function defined as a sequence of WS-OSQL statements that may have side effects. Procedures may also return a result.
- **Overloaded functions**
  An overloaded function is a function that has the same name and result type as some other function but they have different argument types. This allows generic functions to apply to several different object types. In WS-IRIS each specific implementation of an overloaded function is called a resolvent.

3.1.2 WS-OSQL Example

This section shows some WS-OSQL functions that will show some of the constructs available in WS-OSQL. In the example two types of objects are defined, persons and students, the `student` type is a subtype of type `person`. A number of stored functions are defined, `age`, `name`, `address`, `car`, `made_in`, `scollarship`, `income`, `father` and `mother`. The functions `income`, `ancestor` and `parent` are examples of derived functions. The `income` function has two definitions and is thereby overloaded. An example of a recursive function is the function `ancestor` and the foreign function `plus` is used in one of the `income` functions. The function `Number_of_persons` have a nested query and an aggregation function `count`. In the end there is an example of late binding, the function `all_income` returns the correct income for students as well as for persons.

```
create type person;
create type student subtype of person;

create function age(person p) -> integer a as stored;
create function name(person p) -> charstring c as stored;
create function address(person p) -> charstring i as stored;
create function car(person p) -> charstring c as stored;
create function made_in(charstring c) -> charstring d as stored;
create function scollarship(student p) -> integer i as stored;
create function income(person p) -> integer i as stored;
create function income(student s) -> integer i as
   select plus(person.income(s),scollarship(s));
create function father(person p) -> person q as stored;
create function mother(person p) -> person q as stored;
create function parent(person p) -> person q
   as select q where q=father(p) or q=mother(p);
create function ancestor(person p) -> person a
   as select a for each person q
   where (a=ancestor(q) and q=parent(p))
           or a= parent(p);
create function Number_of_persons()->integer i as
   select count(b) for each bag of person b
        where b=(select p for each person p);
create function all_income()-> integer i as
   select i for each person p where i = late(income(p));
```

3.2 ObjectLog

Queries in WS-OSQL compile into an intermediate language called ObjectLog. ObjectLog is the language that the optimizer works with. From the creation of the query to the execution and result, the query passes through several stages that transforms the query. Figure 3 shows the different steps a function passes through in WS-IRIS.

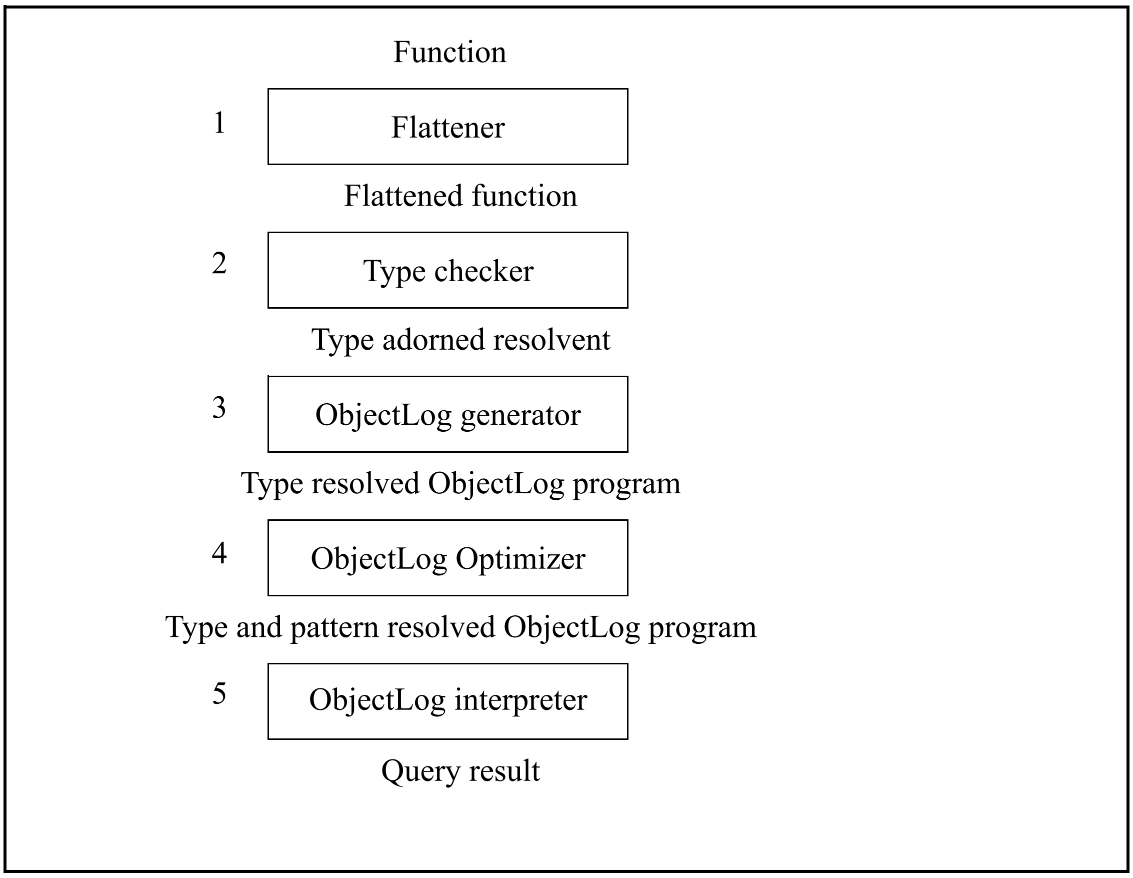


Figure 3. The translation steps of a WS-OSQL function

1. **Flattener**.
   ObjectLog does not allow function symbols to appear as arguments to a function. In this phase select statements are flatten by introducing a new intermediate variable for each nested function call. For example the function call g(f(x)) will be exchanged to _G1=f(x), g(_G1) where f, g are functions and x, _G1 are variables. The flattener also detects and marks recursive functions.
2. **Type checker**.
   This stage is divided into three phases, first the type adornment phase when function calls are marked with argument and result types. The result is called a Typed-Adorned(TA) resolvent. If there are calls to overloaded functions the matching TA resolvents are chosen. Finally, dynamic type checks are added to the function definition whenever the type of a variable cannot be guaranteed to be the required one.

3. **ObjectLog generator**.
   The ObjectLog generator transforms TA resolvents into a Type Resolved(TR) ObjectLog program. Stored functions become TR-facts, derived functions become TR-rules and foreign functions become TR-foreign predicates.
4. **ObjectLog optimizer**.
   The optimizer changes the execution order of the ObjectLog program. Changing the execution order can result in a much faster execution of the program. The output of the optimizer is an optimized ObjectLog program consisting of Type and Binding pattern Resolved(TBR) predicates.
5. **ObjectLog interpreter**.
   The ObjectLog interpreter executes the optimized ObjectLog program and produces a result.

Below an example of an ObjectLog program is shown. Functions from the WS-OSQL example are used and a derived function `country` is created. The function is called with a name of a person and returns the name of the country where the person’s car has been made.
WS-OSQL function

```
create function country(charstring c)-> charstring d as
    select made_in(j) for each charstring j where
         ch=name(p) and
         j=car(p);
```

The corresponding ObjectLog program looks like:

```
((OID[P_CHARSTRING.MADE_IN:158]  J _G2)
 (OID[P_PERSON.NAME:151]  P C)
 (OID[P_PERSON.CAR:153]  P J))
(C)
```

The ObjectLog program consists of two parts. The first part is a list containing the predicates of the program. The order of the predicates specifies the execution order of the program. The second part is a list of variables that are bound when the program is called. The optimizer rearranges the ObjectLog program and produces an optimized ObjectLog program.
The optimized ObjectLog program:

```
((OID[P_PERSON.CAR:153]  P J)
 (OID[P_PERSON.NAME:151]  P C))
 (OID[P_CHARSTRING.MADE_IN:158]  J _G2))
```

The optimizer returns the permutation of the ObjectLog program with the lowest execution cost the optimizer has found.

4 Existing optimization algorithms

Two optimization algorithms were already implemented in WS-IRIS when this work started. They had both some disadvantages leading to the implementation of a new optimizer. In this chapter the existing optimization algorithms are described.

4.1 Exhaustive

The Exhaustive algorithm [Seli79] is the classic algorithm for query optimization used by many database management systems. The algorithm searches through the complete state space. This means that the algorithm always can produce an optimal solution i.e. the state with the lowest cost. An algorithm that produces optimal solutions can be thought of as the obvious choice in a DBMS. But the algorithm has a worst time complexity of $O(2^N)$ where N is the number of predicates in the query. The algorithm is also implemented in a dynamic programming fashion leading to an exponential memory requirement. The time complexity and the memory requirements makes it impossible to use the Exhaustive algorithm with queries that have ten or more predicates. The algorithm is implemented in WS-IRIS and can be chosen as an option.

4.2 Ranksort

The Ranksort algorithm [Kris86, Litw92] is an example of a heuristic query optimization method. The algorithm is already implemented in WS-IRIS and it is the default optimization algorithm. For each predicate in the query a rank is computed and the predicate with the lowest rank is chosen to be the first predicate in the optimized query. The same procedure is repeated again with the remaining predicates and the predicate with the lowest rank is chosen. This continues until all predicates in the originally state has been chosen. The algorithm has a time complexity of $O(N^2)$ where N is the number of predicate in the query.

$$\boxed{R_i = \frac{F_i - 1}{C_i}}$$

Figure 4. The rank calculation formula.

The rank calculation formula is showed in figure 4. The formula shows how to compute the rank of the predicate at position i in the ObjectLog program. R_i is the computed rank, F_i is the fanout of the predicate and C_i is the execution cost of the predicate. Fanout and execution cost are described in detail in section 6.5. There is one problem with this algorithm, it sometimes performs poorly. One example when the algorithm performs poor is when two relations P(X,Y) and Q(R,S) with some properties are to be joined.

$$\boxed{P \bowtie_{Y=R} Q}$$

The relations P and Q have the following properties.

- Relation P is large, and has an index on column Y.
- Relation Q is small.
- X is given when the query is called.

In this example the ObjectLog program has two predicates P and Q. The rank is computed for the predicates. P will be chosen as the first predicate since the relation is large and thereby has a high cost. This is not a very smart move since the entire relation P has to be scanned. It would be smarter to scan the small relation Q, this would give us a bound variable to the Y column in P and the index on P can be used. This example contains only one join, so the Exhaustive algorithm could cope without any time problems. But the problem can be part of a much larger query when an exhaustive search is impossible. In appendix C a more detailed example with cost calculation is given. The *bound-is-easier* heuristics [Ullm89] and *selection pushing* [Elma89] in traditional query optimization have the same problem.

5 Randomized optimization algorithms

Recently it has been proved that the use of a randomized optimization algorithm is the correct choice for optimizing database queries[Ioan90, Swam88]. This chapter presents the terminology used by the randomized algorithms and presents the algorithms that were considered to be the new optimization method in WS-IRIS.

Each solution to a combinatorial optimization problem can be thought of as a state in a state space. In query optimization a state is a join processing tree as described in chapter 2. Each state has a cost associated with it and the cost is given by a cost function. The purpose of an optimization algorithm is to find the state with the lowest cost. The cost is an estimate of the execution time and with a well performing cost model the state with the lowest cost is also the state with the shortest execution time.

The randomized algorithms described in this report performs random walks in the state space by making a series of moves. A move is a permutation of the state where you have your current position in the state space. The states that can be reached in one move from a state is called the *neighbours* of the state. The neighbour function is described in detail in section 6.2. A move which takes you to a state with a lower cost is called a *downhill* move and a move to a state with higher cost is called an *uphill* move.

A state is a *local minimum* if all of the neighbours to the state has higher cost. A state is a *global minimum* if it has the lowest cost among all states. A state is on a *plateau* if it has no lower cost neighbour but the optimizer can still reach lower cost states without any uphill moves. Using the above terminology several randomized algorithms are presented in this chapter.

5.1 Iterative Improvement

The Iterative Improvement(II) algorithm is based on local optimization and has the following behavior. The algorithm starts with a randomly chosen state as the initial state. From the initial state the algorithm moves to neighbour states with lower cost than the current state, this continues until a local minimum has been reached. After a local minimum has been reached a new start state is generated at random. This procedure is repeated until a stop criteria is fulfilled, then the algorithm returns the local minimum with the lowest cost that has been found. The II algorithm is in figure 5 described with pseudo code.

```
II()
 minCost=Maxreal
 while not(stop_criteria) do
     S=Random_state()
     while not(Local_minimum(S)) do
         S’=Neighbour(S)
         if Cost(S’) < Cost(S) then S=S’
     if Cost(S) < minCost then
        minS=S
        minCost=Cost(S)
 return(minS)
```

Figure 5. The II algorithm.

Algorithm explanation
The stop criteria is fulfilled when a certain amount of local minimum has been computed. A randomly chosen state is computed according to the algorithm in section 6.4. In the implementation of the algorithm an approximation is used to identify a local minimum. A state is considered to be a local minimum if none of the neighbours of the state has a lower cost. To be able to do this the neighbour function returns the neighbours in a sequential order. Note that a plateau can be mistaken as a local minimum. The savings in execution time obtained with this method are substantial and motivates the approximation. A neighbour is achieved according to the algorithm in section 6.3. The cost of a state is computed according to section 6.5.

5.2 Simulated Annealing

The Simulated Annealing(SA) algorithm was originally derived by analogy to the process of annealing of crystals, which explains the terminology in the algorithm. The SA algorithm is a more complex algorithm than II and has the possibility to accept moves to a neighbour with a higher cost than the current state, uphill moves. This to avoid being trapped in a high cost local minimum. The initial state is randomly chosen. A move to a neighbour with lower cost, a downhill move, is always accepted just like II. But with some probability it can also accept an uphill move. This probability decreases with time and after a while it becomes harder to accept a move that largely would increase the cost. The algorithm stops when a stop criteria called *frozen* has been reached. Figure 6 shows pseudo code for the SA algorithm. The algorithm consists of two loops. The inner loop of SA is called a stage. Each stage is performed under a fixed value of the *temperature*, which controls the probability of accepting uphill moves. Each stage ends when the algorithm is considered to have reached *equilibrium.*

```
SA()
S= Random_State()
T= Initial_Temperature()
MinS=S
While not(frozen) do
    While not (equilibrium) do
        S' = Random_Neighbour(S)
        $\Delta C$ = Cost(S') - Cost(S)
        if ( $\Delta C \leq 0$ ) then S = S'
        if ( $\Delta C > 0$ ) then S = S with probability $e^{-\Delta C/T}$
        if Cost(S) < Cost(minS) then minS = S
    T = T × Tempfactor
return(minS)
```

Figure 6. The SA algorithm.

Algorithm explanation

The algorithm starts with a random state and the initial temperature is calculated. The stop criteria, frozen, is fulfilled if the best solution has not changed for 5 stages and the percentage of accepted moves do not exceeds a limit *minpercent*. The algorithm is also considered frozen if a *timelimit* has been exceeded. The stage ends when the algorithm has reached an equilibrium. The condition of equilibrium is fulfilled when a certain number of neighbours have been visited. The definition of a neighbour is stated in section 6.2 so is also the method to chose a random neighbour. The cost of a state is calculated according to the cost model given in section 6.5.

5.3 Two phase optimization

The Two phase optimization algorithm(2PO) [Ioan90] was the origin to this work and was to be implemented in WS-IRIS. The idea of this algorithm is that all the low cost states are gathered in a small area. The 2PO algorithm is a combination of II and SA. The algorithm can be divided into two phases. In phase one, II is run for a small period of time and the outcome is the initial state to phase two. In phase two SA is run with a low initial temperature. Since the algorithm consists of two already described algorithms the pseudo code in figure 7 is very short.

```
Two-phase()
    S=II()
    S’=SA(S)
    return(S’)
```

Figure 7. The 2PO algorithm

Algorithm Explanation

The algorithm chooses a good local minimum and searches the area around it. The low initial temperature in phase two prohibits the algorithm from climbing up very high hills.

5.4 Sequence Heuristics

The Sequence Heuristics(SH) algorithm is a variation of local optimization just like II. The initial state is chosen at random and the algorithm moves to a local minimum in the same way as II does. After the first local minimum has been reached a new start state is obtained by making a number of random moves away from the local minimum. When the stop criteria is fulfilled, the algorithm returns the local minimum with the lowest cost that has been found.

This algorithm will not be used on its own, instead the algorithm is used in a two phase manner where it is proceeded by the II algorithm. This to get a good initial state for the SH algorithm. The idea of this algorithm came up when the second phase in 2PO did not make the number of moves one could expect. With this method a number of moves away from the local minimum is guaranteed. This method is intended to give a good result if the states with low cost are gathered in a rather small area. Figure 8 shows the pseudo code of the SH algorithm.

```
SH()
minCost=Maxreal
while not(stop_criteria) do
    S=random_moves(minS)
    while not(local_minimum(S)) do
        S'= Neighbour(S)
        if Cost(S') < Cost(S) then S=S'
    if Cost(S) < minCost then
        minS=S
        minCost=Cost(S)
return(minS)
```

Figure 8. The SH algorithm

Algorithm explanation

The SH algorithm do not differ much from the II algorithm. The stop criteria, local minimum and the cost are computed in the same way as for the II algorithm. The only thing that differ is the way the algorithm choose new start states. The new start state is obtained by making random moves to neighbours. In the implementation the algorithm makes the same number of moves as the number of predicates in the query.

6 Implemented query optimization algorithms

The algorithms in chapter 5 are described in a general way and in this chapter some details are filled in for query optimization. Six different algorithms are presented in this chapter. There are also some details that are common to all algorithms that needs to be described, namely the neighbour function and the cost model.

6.1 Implementation specific parameters

In this section the six candidate algorithms for WS-IRIS are presented. The algorithms are defined by setting parameters to the general algorithms. There are these algorithms that are compared to each other in chapter 7.

- **II5**

parameter	value
general algorithm	Iterative Improvement
stop criteria	5 local minimum calculated

- **II10**

parameter	value
general algorithm	Iterative Improvement
stop criteria	10 local minimum calculated

- **SA**

parameter	value
general algorithm	Simulated Annealing
initial temperature	2×Cost(initial state)
equilibrium	1×(number of predicates in query)
timelimit	0.3×(number of predicates in query)^2
minpercent	2
tempfactor	0.9

- **SH37**

parameter	value
general algorithm (phase one)	Iterative Improvement
general algorithm (phase two)	Sequence Heuristics
stop criteria (phase one)	3 local minimum
stop criteria (phase two)	7 local minimum
random_moves	Number of predicates in query

- **SH55**

parameter	value
general algorithm (phase one)	Iterative Improvement
general algorithm (phase two)	Sequence Heuristics
stop criteria (phase one)	5 local minimum
stop criteria (phase two)	5 local minimum
random_moves	Number of predicates in query

- **TWO**

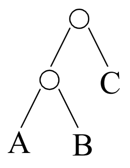
parameter	value
general algorithm (phase one)	Iterative Improvement
general algorithm (phase two)	Simulated Annealing
stop criteria (phase one)	5 local minimum
initial temperature	2×Cost(initial state)
equilibrium	1×(number of predicates in query)
timelimit	0.3×(number of predicates in query)^2
minpercent	2
tempfactor	0.9

6.2 The neighbour function

The randomized algorithms described in chapter 5 improve their solutions by making moves in the state space trying to find the global minimum. The algorithms make two different kind of moves. One is a move to a random state and the other is a move to a neighbour state. The neighbours of a state S are defined as the states that can be reached from S in one step. The neighbours of a state is determined by a set of transformation rules. The same five rules as presented in *Randomized Algorithms for Optimizing large join queries* [Ioan90] are used. The five rules are:

1. Join method choice	$A \bowtie_{*} B \longrightarrow A \bowtie_{+} B$
2. Join commutativity	$A \bowtie B \longrightarrow B \bowtie A$
3. Join associativity	$(A \bowtie B) \bowtie C \longleftrightarrow A \bowtie (B \bowtie C)$
4. Left join exchange	$(A \bowtie B) \bowtie C \longrightarrow (A \bowtie C) \bowtie B$
5. Right join exchange	$A \bowtie (B \bowtie C) \longrightarrow B \bowtie (A \bowtie C)$

Each state can be represented as a join processing tree, see chapter 2. A JT is a tree which leaves are relations and nodes are join operators.



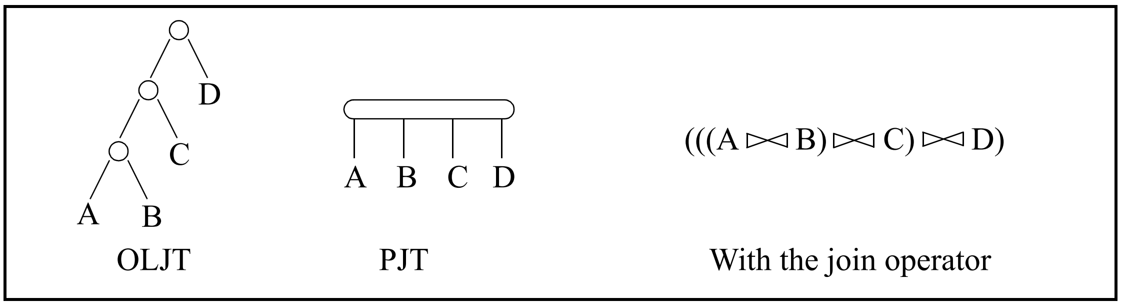
This tree represents the state $(A \bowtie B) \bowtie C$

Example 2: A join processing tree.

WS-IRIS has some simplifications that causes some of the above presented rules to be not applicable or that the rules can only be used in some special cases.

6.2.1 The simplicity of WS-IRIS

WS-IRIS has several simplifications when it comes to the above rules. As stated in chapter 2, a JT has a certain shape in WS-IRIS. They are all pipelined join processing trees which can be thought of as OLJTs that do not create any intermediate relations. Example 3 shows the connection between OLJT, PJT and the join operator.



$(((A \bowtie B) \bowtie C) \bowtie D)$

Example 3: Join processing Trees

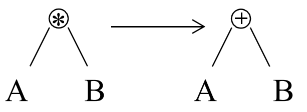
There is one more restriction in WS-IRIS that affect the use of the rules. That is the possibility to choose join method in WS-IRIS. The one and only join method used in WS-IRIS is the nested-loop join. In the following sections it will be shown how the rules are applicable to states in WS-IRIS.

6.2.2 Join method choice

This rule changes the join method of a join operator. A node in the JT will be replaced with a new node representing another join method.

Formal: $A \bowtie_{*} B \longrightarrow A \bowtie_{+} B$

Graphical:



Where ⊛ and ⊕ are two different join methods.

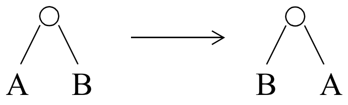
This rule is not applicable in WS-IRIS. There is only one join method available so a change is impossible. The join method in WS-IRIS is called nested-loop join.

6.2.3 Join commutativity

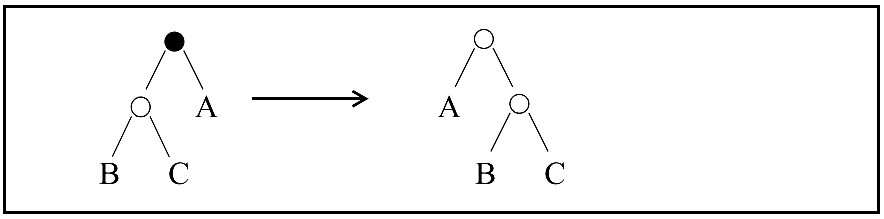
This rule changes the order in which the operators are applied to a node. It makes the left and the right subtrees of the node to switch places.

Formal: $A \bowtie B \longrightarrow B \bowtie A$

Graphical:



This rule can only be used in WS-IRIS with some restrictions. All JTs in WS-IRIS are OLJTs and if the rule is applied to a node which children are not leaves the new tree will not be a OLJT, as shown in example 4.



Example 4: The join commutativity rule applied on the marked node.

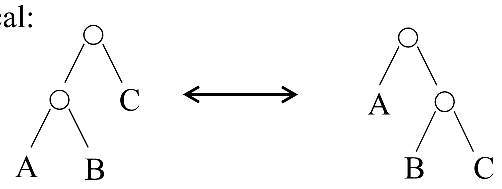
The conclusion is that there exists only one case when this rule can be used. That is when both of the children to the node are leaves. There is only one such node in a left-deep tree, a join between the first and second relations in the tree.

6.2.4 Join associativity

This rule changes the order in which two join operators are applied to their operators.

Formal: $(A \bowtie B) \bowtie C \longleftrightarrow A \bowtie (B \bowtie C)$

Graphical:



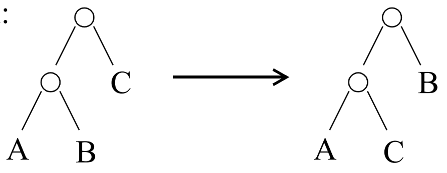
The use of this rule on a node makes a left-deep subtree of the node to become right-deep. Right-deep subtrees are not allowed in WS-IRIS so this rule cannot be used.

6.2.5 Left join exchange

This rule is a more complex rule and the result can be obtained using the above rules in a compound fashion. This rule works as a bypass trying to avoid plateaus in the state space. The rule swap places of the inner relation of a join with an intermediate relation and the inner operand of the intermediate relation i.e. the switch of two right operands on different but adjacent levels in a JT.

Formal: $(A \bowtie B) \bowtie C \longrightarrow (A \bowtie C) \bowtie B$

Graphical:



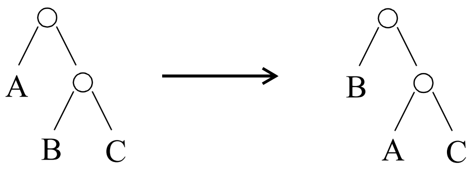
This rule will not change the structure of the tree and can thereby be used in WS-IRIS.

6.2.6 Right join exchange

This rule works in the same manner as left join exchange, with the difference that there are two outer operands that switch places. With this rule two left operands on different but adjacent levels in a JT can switch places.

Formal: $A \bowtie (B \bowtie C) \longrightarrow B \bowtie (A \bowtie C)$

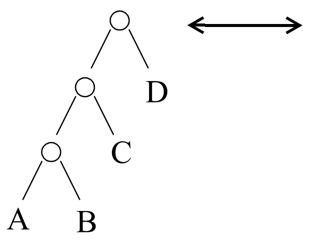
Graphical:



This rule cannot be used because there are no subtrees with such a structure in WS-IRIS.

6.2.7 Conclusions

It has been shown that only two of the five rules can be applied to states in WS-IRIS, join commutativity that swap places of two leaves of a node and left-join exchange to swap places of two leaves on different but adjacent levels in a tree. A join processing tree in WS-IRIS is simply represented by a list of ObjectLog predicates. Therefore a neighbour to a state is calculated simply by swapping places of two adjacent elements in the list. Example 5 shows a JT and its corresponding list. The example also shows the neighbours of the state.



(A B C D)

Neighbours: (B A C D)
(A C B D)
(A B D C)

Example 5: The neighbours of a state

6.2.8 Invalid neighbours

It is not always the fact that a calculated neighbour is a valid state. This happens when a predicate is swapped to a place where it do not have the number of bound variables it needs to be executed. It is often foreign functions that requires certain variables to be bound.

6.3 Random neighbour

With the above reasoning the calculation of a random neighbour is rather simple. The pseudo code for the algorithm is shows in figure 9. The only thing that require special handling is to avoid invalid neighbours. Functions for recognizing invalid states have already been implemented in WS-IRIS.

```
Random_neighbour(S)
  y=Size_of(S)
  repeat
    x=Random(1,y-1)
    S'=swap(S,x,x+1)
  until valid_state(S')
  return(S')
```

Figure 9. The random neighbour algorithm

Algorithm explanation

The algorithm is called with the source state S. The number of predicates in S is calculated. The Random function returns a random number in the range [1..y-1], this chooses a random neighbour. Then the neighbour is calculated by swapping places of two predicates in the list. This is repeated until a valid neighbour has been found.

6.4 Random state

A random state can not be computed just by choosing a random permutation of the source state. Some of the predicates that the state consists of can be depended on that certain variables are bound when the predicates are to be executed. This is the same problem as before when choosing a random neighbour. The algorithm must take this under consideration. Figure 10 shows the pseudo code of the algorithm for computing a random state.

```
Random_state(S)
  x=Size_of(S)
  for i=1 to x do
      repeat
          y=Random(i,x)
      until Valid(i,y)
      S=Swap(S,i,y)
  return(S)
```

Figure 10. The random state algorithm

Algorithm explanation

The algorithm is called with the source state S. The number of predicates S consists of is calculated. For each position i in S a random predicate that has not yet been chosen is selected. If the chosen predicate can be executed at position i it is swapped to position i.

6.5 The cost function

The task of a query optimizer is to minimize the execution time of an ObjectLog program. As stated in chapter 5 the optimizer searches through the state space trying to find the state with the lowest cost. The cost is an estimate of the execution time. The reason to use an estimate instead of the real value of time is that the optimizer visits a large number of states and it would be far too time consuming to execute each state to get the execution time. Functions for cost estimates were already implemented in WS-IRIS and are used by Ranksort and Exhaustive. The existing cost estimates proved to be too rough, resulting in that the query optimizer came up with a state with lower cost than some other state but with longer execution time.

Hence a better cost estimate function was implemented.

Most of the moves made by the randomized optimization algorithms are moves to neighbour states. With the definition of a neighbour in the previous section the cost of a neighbour can be incrementally computed. This gives us a more efficient implementation.

6.5.1 Cost model

The structure of the cost function is primary effected by two things. First the join method which in WS-IRIS is the nested-loop join. Secondly it is effected by the fact that the database is entirely in main memory. This means that expensive disc accesses do not have to be estimated in the cost function. The cost model in WS-IRIS is the following:

Let P be an ObjectLog program with a corresponding tuple containing the bound variables. For each TBR-predicate P_i in the ObjectLog program two estimates are calculated.

1. The execution cost of P_i, called C_i, defined as the number of visited tuples in the database, given that the variables of the input tuple are bound.
2. The fanout of P_i, called F_i, which is the estimated number of output tuples produced by P_i for a given input tuple.

The optimizer minimize the total cost of an ObjectLog program with the predicates $P_1, P_2, ..., P_n$. The total cost of a program is calculated according to the formula in figure 11.

$$\boxed{C = \sum_{i=1}^{n} C_i \prod_{j=1}^{i-1} F_i}$$

Figure 11. The cost function.

Default values

In a populated database the system estimates C_i and F_i from the cardinality of stored predicates, join selectivities and the presence of indexes. To get a reasonably optimization even before the database is populated, the system uses the default parameters below.

- $F_i = 1$ if a bound variable has a unique index.
- $F_i = 2$ if the variable has a non unique index.
- $F_i = 4$ otherwise.
- The default size of a stored predicate is assumed to be 100 tuples.
- $C_i = F_i$ if the bound variable has an index.

- $C_i = 100$ if the bound variable is unindexed since the system has to scan the entire table.
- Foreign predicates have default $F_i = 1$ and $C_i = 1$, assuming that they are cheap to execute and return a single tuple.

6.5.2 Calculation of the cost of a neighbour

A neighbour is computed with the simple operation of swapping places of two adjacent predicates in the ObjectLog program. C_i, F_i and the input tuple with bound variables are saved for each position in the ObjectLog program. The fact that a move to a neighbour only locally effects the cost function can be used. The information is saved when a new initial state is created. The algorithm in figure 12 is used to test whether or not a neighbour has lower cost than the source state. The algorithm is assuming that it is the i:th neighbour that are to be checked.

```
$C[i]_{new}$=Predicatecost(pred(i+1) , B[i])
$F[i]_{new}$=Fanout(pred(i+1) , B[i])
$B[i+1]_{new}$=Pred_binds(pred(i+1) , B[i])
$C[i+1]_{new}$=Predicatecost(pred(i) , $B[i+1]_{new}$)
$F[i+1]_{new}$= Fanout(Pred(i),$B[i+1]_{new}$)
$\Delta C' = (C[i]_{new} + F[i]_{new} \times C[i+1]_{new}) - (C[i]_{old} + F[i]_{old} \times C[i+1]_{old})$
**if** $\Delta C' < 0$ **then**
    **return**(better state)
**else**
    **return**(worse state)
```

Figure 12. The neighbour cost function

Algorithm explanation

C, F and B are arrays where the values of cost, fanout and bound variables for each predicate are saved. Pred(i) is the predicate at position i in the ObjectLog program.

1. The new cost for the predicate at position i, $C[i]_{new}$, is calculated using the predicate at position i+1 and the input tuple for position i.
2. The same goes for $F[i]_{new}$.
3. The new input tuple for position i+1, $B[i+1]_{new}$, are calculated. This is done by using the input tuple for position i and add the variables that the new predicate at position i will bound.
4. $C[i+1]_{new}$ is calculated using the predicate at position i and the new input tuple for position i+1.
5. The same goes for $F[i+1]_{new}$.

6. According to the cost model the old and the new cost are compared.
7. And last the function return whether or not a move to the neighbour is an improvement of the cost.

Of the above seven point only the sixth point needs to be motivated a little deeper. The cost function has the following form.

$$C_1 + F_1C_2 +...+ F_1\times...\times F_{i-1}\times C_i + F_1\times...\times F_i\times C_{i+1} +...+ F_1\times...\times F_{n-1}\times C_n$$

A swap of places between the predicates at position i and i+1 will lead to a new cost as described below.

$$C_1 + F_1C_2 +...+ F_1\times...\times F_{i-1}\times C_{inew} + F_1\times...\times F_{inew}\times C_{i+1new} +... \\ ...+ F_1\times...\times F_{inew}\times F_{i+1new}\times...\times F_{n-1}\times C_n$$

The first i-1 terms are not effected of the swap. The terms i and i+1 have changed with the new values that the algorithm has assigned to C_i, C_{i+1}, F_i and F_{i+1}. The rest of the terms have not changed in spite of the new definition of F_i and F_{i+1}. This is true because the number of tuples produced by two predicates is the same no matter the order of the predicates. Also the number of bound variables will be the same. If the move to the neighbour are chosen the ΔC can be calculated according to: $\Delta C=F_1\times...\times F_{i-1}\times\Delta C'$.

6.5.3 Improvements of the cost model

The estimate of the fanout made by WS-IRIS appeared to be not good enough for query optimization. The bad estimates could force the optimizer to choose a wrong solution. In the old version of WS-IRIS the fanout was calculated according to the algorithm in figure 13.

```
Fanout=Relation Cardinality
For each column in the relation that has a bound variable do
  If there is an unique index then
    F=1
  If there is a nonunique index and the relation is empty then
    F=2
  If there is a nonunique index and the relation is not empty then
    F=Relation Cardinality / Number of distinct values in column
  If index is missing then
    F=4
  Fanout=min(Fanout , F)
return(Fanout)
```

Figure 13. The old fanout estimation algorithm.

The returned fanout is the minimum fanout of all columns with bound variables in the relation. In Appendix C an example is presented that shows the result of a bad fanout estimation. The reason of the bad behaviour are two things. First the default value 4, when an index is missing on the current column, can be very wrong. Second the way of choosing the minimum fanout of the columns as the fanout should instead be calculated as the product of all selectivities in the columns.

In the new implementation the fanout is calculated by multiplying the individual selectivities for each column to each other, this assumes independence between columns. To get a reasonable good estimate when an index is missing on a column, a part of the relation is scanned and a selectivity is calculated. Figure 14 shows the new fanout estimation.

```
Fanout=1
For each column in the relation that has a bound variable do
   If there is an unique index then
      F = 1 / Relation Cardinality
   If there is a nonunique index and the relation is empty then
      F = 2 / Relation Cardinality
   If there is a nonunique index and the relation is not empty then
      F = 1 / Number of distinct values in column
   If index is missing and the relation is empty then
      F=0.4
   If index is missing and the relation is not empty then
      F = 1 / Scanned number of distinct values in column
   Fanout=Fanout × F
return(Fanout × Relation Cardinality)
```

Figure 14. The new fanout estimation algorithm.

The estimated selectivity that is used, when there is no index on the current column, is a sample of 100 tuples in the relation. A sample of a relation can be done because the database is in main memory, in a disc based database on the other hand a sampling would take too long time. The method of sample tuples in a relation to estimate the selectivity of a column assumes that the sampled column contains of an uniform distribution of values. If the relation is empty a default value of 0.4 is used as the selectivity.

7 Testing the algorithms

The test of the algorithms can be divided in three phases, first the database and the test queries are generated according to sections 7.1 and 7.2. Then the randomized algorithms that have been implemented are tested against each other. The algorithm with the best performance is chosen to be an optional optimization algorithm in WS-IRIS. Finally the chosen algorithm is tested against the existing algorithms in the system, Ranksort and Exhaustive.

• **Experimental method**

The test has the same structure with some modifications as the method Arun Swami used when he tested a number of randomized algorithms for query optimization [Swam88]. The choice of Swami's method was mainly based on the fact that he used a main memory database, just like WS-IRIS, in his test. Yannis Ioannidis [Ioan90] on the other hand used an disc based database which led to a different cost estimate model.

• **Differences with Swami's test**

There are some major differences between this test and Swami's. Foreign functions are used as part of the queries in this test. WS-IRIS uses another join method namely the nested-loop join instead of the hash join method. It has been argued that the nested-loop join method combined with dynamic index creation is the most important query processing method for a main memory DBMS [Whan90]. Swami did not test the two-phase algorithm nor the combination of Iterative Improvement and Sequence Heuristics. Swami also used another method to calculate the neighbours of a state. WS-IRIS have no sorted indexes this means that comparison functions are treated as foreign functions.

• **A standard method**

A more standardized test method for the experiment than Swami's method was desirable. *The benchmark handbook for database and transaction processing systems* [Gray91] describes a number of methods, but there were not any directly applicable methods for a test on optimizing large join queries in an object oriented main memory based DBMS.

• **Equipment**

The test was performed on a SUNsparcstation 1 workstation. The optimization algorithms were implemented in Lisp and the query language was WS-OSQL.

7.1 Building a database

The database used during the test of the algorithms is generated by a Lisp program. The database is generated at random with the restriction of a number of parameters. The random number generator is initialized at the beginning of the program. The same database can thereby be generated again if needed. Below follows a description of the generated database and in appendix A the contents of the generated database are presented.

- **Number of relations**
  The database has 25 relations.
- **Relation Cardinality**
  The cardinality of a relation is the number of tuples that the relation contains of. The database was generated with the following cardinalities of the relations.
  20% of the relations contain 10 - 100 tuples.
  64% of the relations contain 100 - 1000 tuples.
  16% of the relations contain 1000 - 10000 tuples.
  The exact number of tuples is randomly chosen between the upper and lower limit for each group.
- **Number of columns**
  The number of columns in each relation was randomly chosen with probabilities taken from the table below.

Number of columns	1	2	3	4	5	10
Probability	1%	35%	30%	25%	5%	4%

- **Distinct values in join column**
  To have the possibility for making a join between two arbitrary columns the relations contain nothing but integers. The first column in the relation is separated from the other, this because it is often the first column that contains the key elements in the relation. This leads to a larger amount of distinct values in the first column. The values that are used for the first column is presented in the following table.

1	50%
0.85-0.95	50%

An explanation of the table might be appropriate here. The first column in the above table contains the percentage of distinct values for a column in a relation. The second column specifies the probability of chosen the value in the first column. What these values mean are that with a probability of 50% the column is chosen to have
1 × ”the cardinality of the relation”
number of distinct values.The second row has the same probability to be chosen namely 50%. If the second row is chosen the number of distinct values are randomly chosen to be in 0.85 to 0.95 times the cardinality of the column. The values that are used for the other columns are the following.

1	25%
0.2-1	5%
0.001-0.2	75%

This table is read in the same way as the table for the first column. The values are spread with a uniform distribution in the range [0..”the cardinality of the largest table”]. This makes the values in each column in each relation to be distributed in the same range.

- **Index**
  There is always an index on the first column in the relation and with a probability of 10% on the other columns. In an Object Oriented DBMS like WS-IRIS relations are treated as functions. Functions are normally accessed in a forward direction, this means that an index on the first argument(column) of the function is needed. Functions are normally single value functions leading to the choice of an unique index on the first argument(column).

7.2 Query generation

The OSQL-queries that have been used during the testing of the algorithms were generated by a Lisp program. With specifications of the number of queries and the number of predicates in the query, the program generates and writes the queries to an external file. The predicates in the query are joined in three different ways. The first way is the standard join method. This method is a join between two columns in two relations (stored functions). Only equi-join is used in the test. Second there is selection, a method that works as a filter. In this test the selection predicate is always a foreign function that checks if a bound variable fulfils some condition. Finally ordinary foreign functions like plus and times are used in the queries. The type of predicate used is chosen at random with probabilities taken from the table below.

Standard Join	60%
Selection	20%
Foreign function	20%

The program adds joins to the query until the correct number of predicates have been received. The random number generator is initialized at the beginning of the program. The same queries can thereby be regenerated if the test is to be repeated.

7.2.1 Standard join

A standard join means a join where a column in a relation is compared to a bound variable. The type of join that is used in this test is equi-join. The two columns of the two relations are compared by using the same variable in the two columns.
For example a join is to be done after the first predicate has been chosen.
The first predicate: R5(I0 , I1)=I2
A relation, a column and a variable are chosen at random: 3, 0, 1
The new predicate in the query gets the following looks: R3(I1 , I3 , I4)=I5
A join between relation R5 and R3 has been done, the join columns are the second in R5 and the first in R3.

7.2.2 Selection

Selection means that a subset of the values that are assigned to a variable are chosen. The table below shows the selectivities that have been used in the test. For each selection a random selectivity is fetched from the table.

0.001	0.01	0.1	0.2	0.34
0.34	0.34	0.34	0.34	0.5
0.5	0.5	0.67	0.8	1.0

For example a selectivity of 0.34 means that 34% of the values a variable is assigned to are chosen. The selectivities 0.34 and 0.5 are dominating the table. This is an estimate made by many query optimizers including system/R[Seli79]. To be able to chose the correct number of values when a selection is done, information about the amount of unique values in a column is used. The values in a column have an uniform distribution starting with zero as the lower limit and have the maximum cardinality as the upper limit. The obvious way to choose a subset of values will be to choose all values higher than some constant
For example, assume that a selectivity of 0.34 has been chosen, the largest relation has 10000 tuples and the selection is to be done with variable I1. The variable has an uniform distribution in [0..10000]. To select 34% of the values is done by selecting all values larger than (1-0.34)×10000=6600. I1 > 6600

7.2.3 Foreign functions

The third way to import predicates to the query is to add foreign functions. In the test queries five different foreign functions have been used. The actual function is randomly chosen from the table below.

Function name	Description
Plus	Addition
Times	Multiplication
Inc	Add with one
Sum3	The sum of 3 integers
Avg	The average of 2 integers

Variables to be used as parameters to the chosen function are randomly chosen from the set of already defined variables i.e. variables that have been used in earlier predicates. A new variable is created for the result.
Example:

PLUS(I1 , I5)=I10

7.2.4 WS-OSQL queries

This section shows two examples of how the generated queries looks like. The queries have five and ten predicates.

```
create function test7()-> integer I1 as
 select I1 for each integer I0,
 integer I1, integer I2, integer I3, integer I4, integer I5
 where
        R24(I0)=I1 and
        R13(I2,I0)=I3 and
        I1 > 1999 and
        R10(I4)=I2 and
        PLUS(I0,I3)=I5;
```

Example 6: A query with five predicates.

```
create function test2()-> integer I9 as
 select I9 for each integer I0,
 integer I1, integer I2, integer I3, integer I4, integer I5,
 integer I6, integer I7, integer I8, integer I9, integer I10,
 integer I11, integer I12, integer I13, integer I14 where
         R5(I0,I1)=I2 and
         R13(I3,I4)=I0 and
         R14(I2,I5)=I6 and
         R2(I7,I2)=I8 and
         I0 > 5000 and
         R0(I5)=I9 and
         I4 > 1999 and
         PLUS(I3,I2)=I10 and
         R6(I11,I12,I13)=I6 and
         PLUS(I8,I1)=I14;
```

Example 7: A query with ten predicates.

7.3 Experiment procedure

To be able to distinguish the performance of the different algorithms from each other a number of queries were optimized with the different algorithms. Each query was optimized with each one of the algorithms. In the test 10 different queries were used for each N=3, 5, 10, 15, 20, 25, 30, 40, 50, 75, 100, where N is the number of predicates in the query. Each query was generated in the manner described in the previous section. This give us a total of 110 queries. Ranksort was used once on each query, so was also the Exhaustive algorithm as long as the optimization time allowed that. The randomized algorithms were used five times on each query. This because the randomized algorithms do not necessarily generate the same solution each time the algorithm is run. At each optimization information about the optimization time and the execution cost were gathered.

7.3.1 Optimization time

Optimization time is the CPU-time that the optimization algorithm needs to produce an optimized query. The CPU-time is read with the UNIX system function clock. This function returns the amount of CPU-time in microseconds used since the first call to clock. The time reported is the sum of the user and system times of the calling process.

7.3.2 Execution cost

The cost of executing a query is an estimate direct associated with the execution time of the query. The cost model is described in section 6.5. The importance of minimizing the cost is increasing with the number of executions the query will do. This means that the cost is the most important factor when optimizing a query that will be executed more than just a few times.

7.4 Results

The measured results of the algorithms are presented in this section. The execution cost of different queries need to be expressed on a common scale before they can be compared to each other. This is done by dividing each cost of a query by the lowest cost that the optimizers has found for the query. The new cost is called the scaled cost and the best scaled cost of each query has the value 1. The results are shown in graphs. Two tables with results are presented in appendix B. In the test eight different algorithms have been compared to each other. To get readable graphs the result of the randomized algorithms are presented first. The randomized algorithm with the best performance is then compared to the results of Ranksort and Exhaustive.

The results shows that the SH55 algorithm is the best suited optimization algorithm for WS-IRIS. First the median scaled cost of the output strategies for each query size are shown for the randomized algorithms. The median cost is used instead of the average cost because otherwise one very large cost affects the average too much. Two of the randomized algorithms have better median scaled cost than the other, SH55 and II10. The SA algorithm proved to be the worst algorithm.

To be able to check if the algorithms fulfils the criteria of a good query optimizer the median cost of the 10% of the worst optimization results are presented. Again SH55 and II10 are superior of the other algorithms. The average optimization time shows that the SH55 algorithm is faster than II10. SH55 and II10 cannot be separated from each other by their optimization result. The optimization time has to be the determinative factor. SH55 has shorter optimization time and is therefore chosen to be the third optimization algorithm in WS-IRIS.

The SH55 algorithm is then compared with the existing algorithms. The optimization time of the Exhaustive algorithm makes it impossible to optimize queries of size ten and larger. Ranksort is a very fast algorithm, twice as fast as SH55. The optimization result for Ranksort on the other hand is not very good. The median scaled cost of the output strategies for Ranksort is never better than for SH55 not for any query size. The scaled cost is often (24% of the cases) more than 1000 times worse for Ranksort compared to SH55. Of all optimized queries Ranksort produces a worse result in 72% of the queries compared to SH55 and the algorithms performed equal in 16% of the queries. The SH55 algorithm is superior to Exhaustive because of the optimization time and superior to Ranksort because of the optimization result.

7.4.1 The median scaled cost

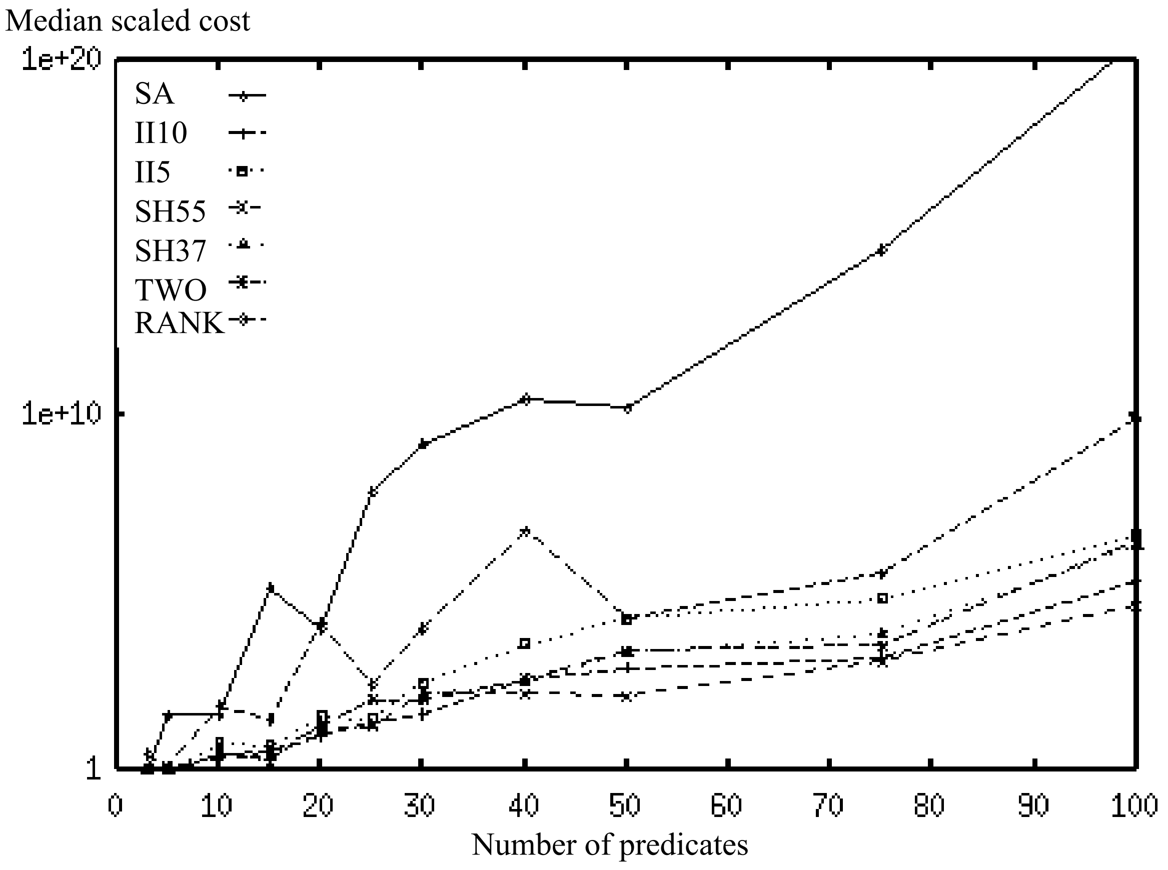


Figure 15. The median cost of the output strategies of the randomized algorithms

The graph in figure 15 shows the median scaled cost of the output strategies of the randomized algorithms and the Ranksort algorithm. The SA algorithm is not very effective in query optimization and has a very high output cost compared to the other algorithms, note that the y-axis in the graph has a logarithm scale. The Ranksort algorithm has better output than SA but is worse than the rest. To get a closer look at the rest of the algorithms the SA and the Ranksort algorithms are omitted and the graph is generated again in figure 16 with the same data.

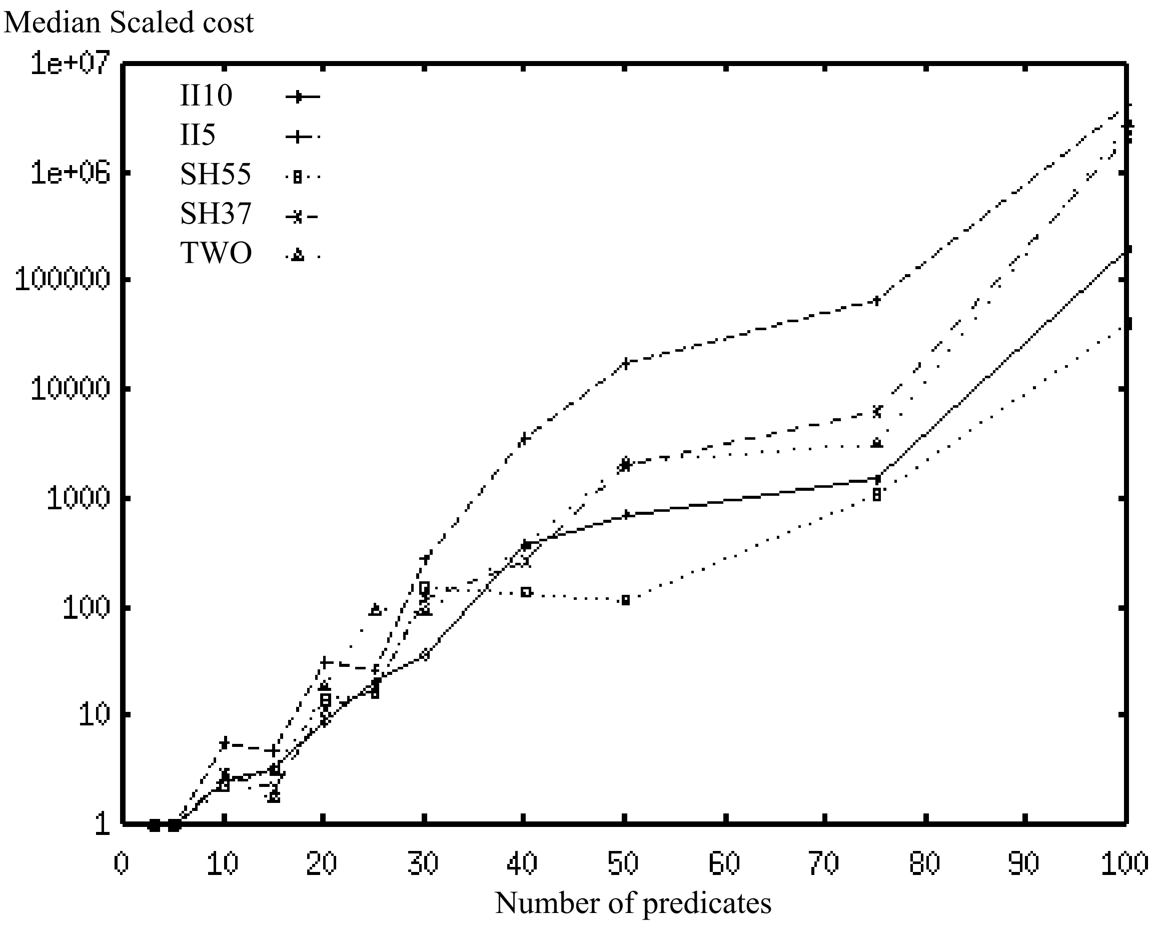


Figure 16. Median cost of the output strategies of the randomized algorithms.

In figure 16 the median scaled cost of the output strategies of the randomized algorithms at each size of the query are shown. Two algorithms, II10 and SH55 have better performance than the rest when the query increases in size. The figure also shows that 5 local minimum is too few to get a good optimization. The TWO and the SH37 algorithms have their performance somewhere in the middle.

7.4.2 The worst cost

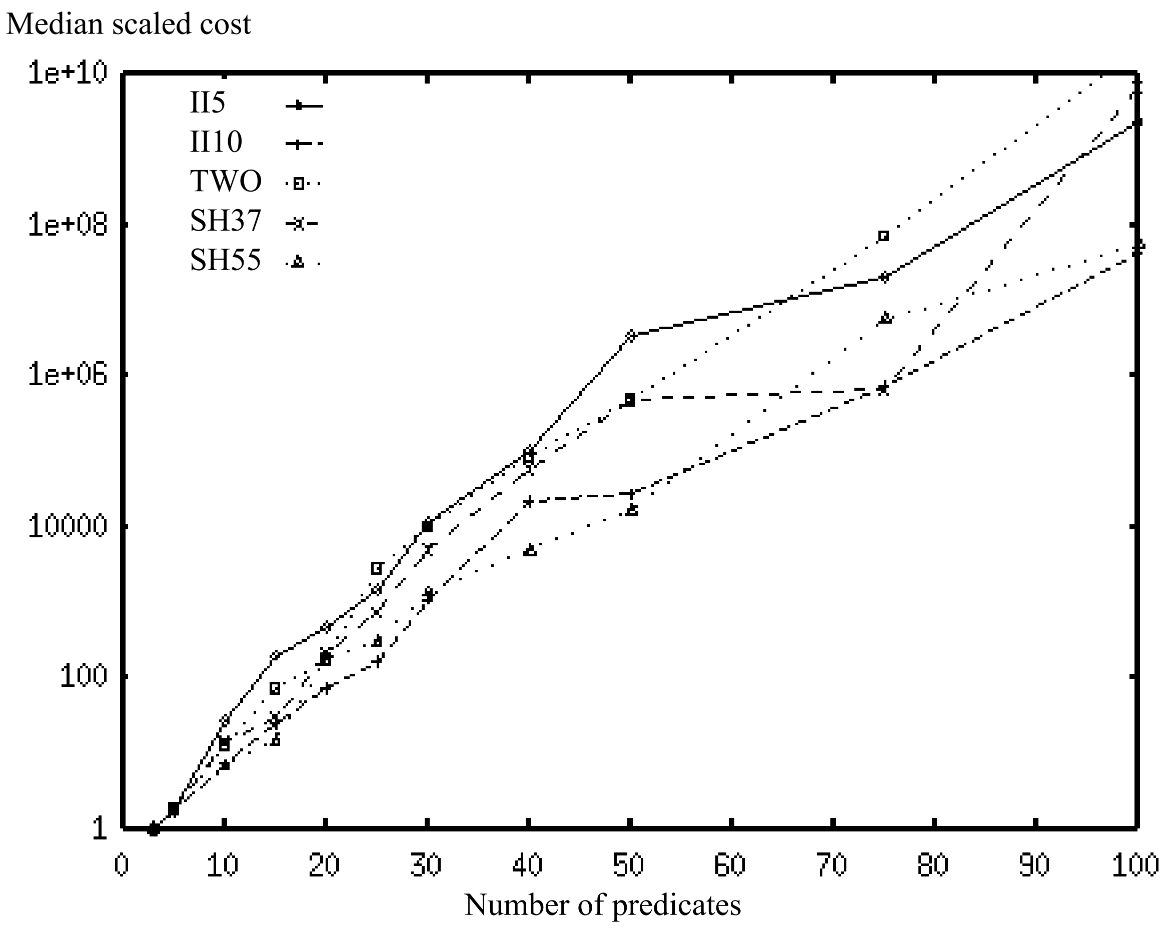


Figure 17. Worst 10% of the output strategies

As stated before it is more important for a query optimization algorithm to avoid bad solutions than it is to find the very best. Figure 17 shows the median of the 10% of the worst optimization results made by the randomized algorithms. Again the II10 and SH55 algorithms shows the best performance. This means that the result of II5 and SH37 can be improved with longer optimization time. The TWO algorithm shows no better optimization results than II5, which means that the second phase do not improve the optimization result very much.

7.4.3 The average optimization time

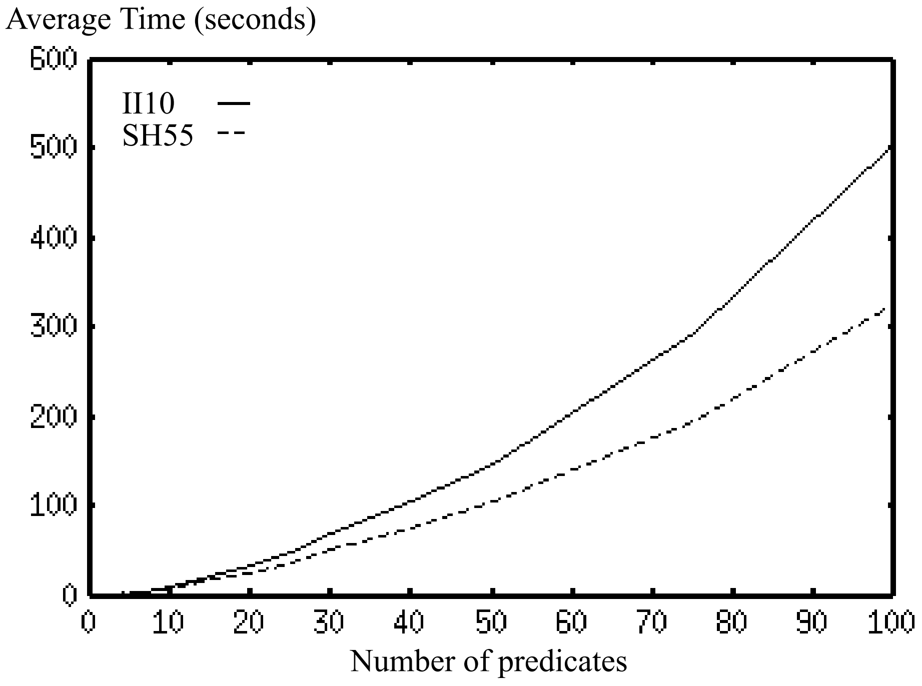


Figure 18. Average optimization time II10 and SH55

Since the II10 and SH55 algorithms have almost the same performance, the optimization time has to be the determinative factor. Figure 18 shows the average optimization time and the figure shows that the SH55 algorithm is faster than II10 and will therefore be chosen as the new optimization method in WS-IRIS.

7.4.4 Improvements over time

The behaviour of the randomized algorithms over time are shown in figure 19. The example contains only results of queries with 50 predicates, but it can give us a clue of how the algorithms work. In the figure it is again shown that the SA algorithm is not suited for query optimization even if the algorithm is getting more time. The SA phase in the TWO algorithm do not improve the optimization result very much.

One important thing to notice is that the SH algorithm improves the solution more rapidly than the II algorithm when it starts the second phase. After a while when the SH algorithm has searched through the states nearby the best solution no more improvements are made. II improves the solution during all the optimization time and passes the SH algorithm after some time. The above means that with a limited optimization time the combination of Iterative Improvement and Sequence Heuristics gives the best performance. II improves rapidly at the beginning and then after a while the Sequence Heuristics algorithm can still improve the solution by searching the states near the best solution. This gives us a good optimization with short execution time.

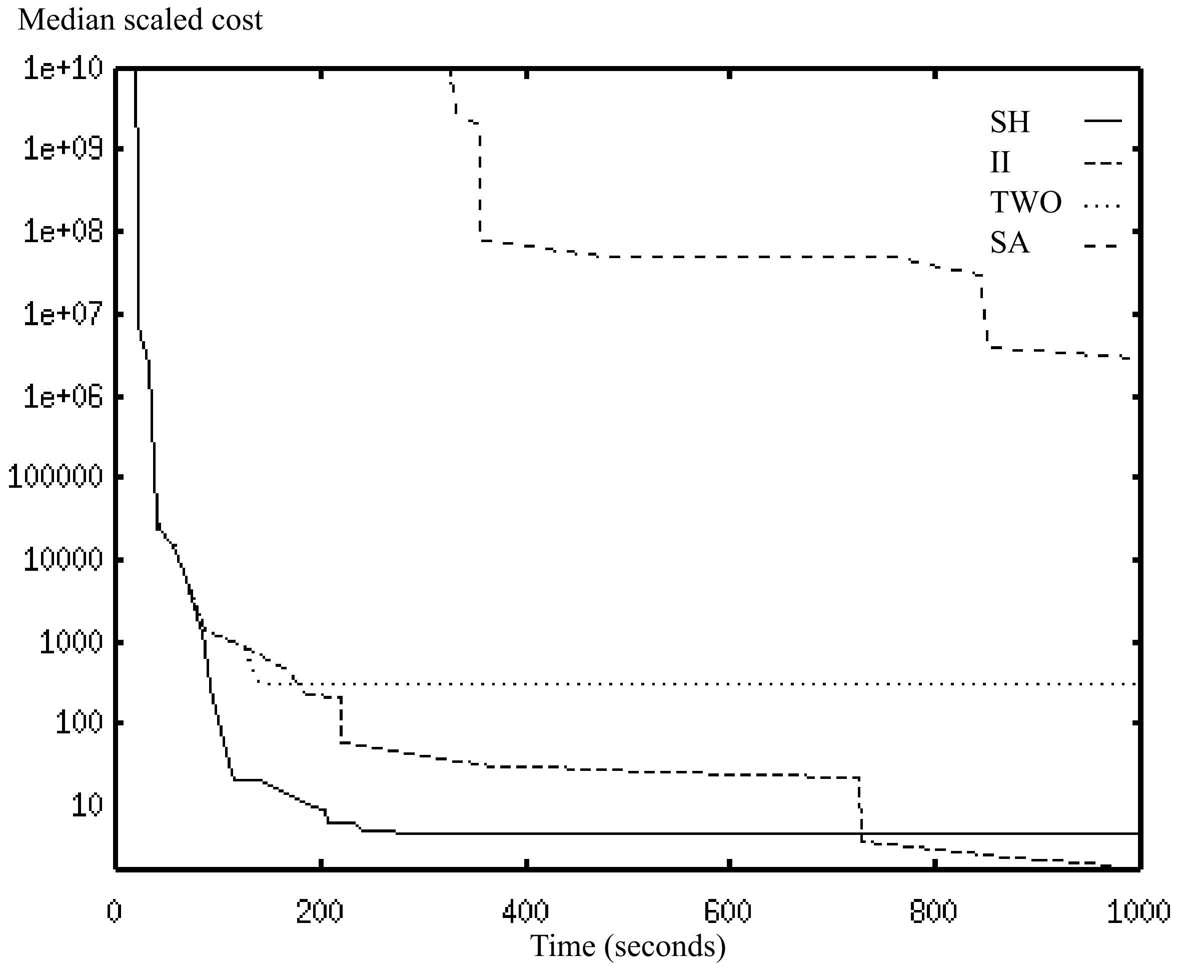


Figure 19. Improvements over time N=50

7.4.5 Randomized versus Existing algorithms

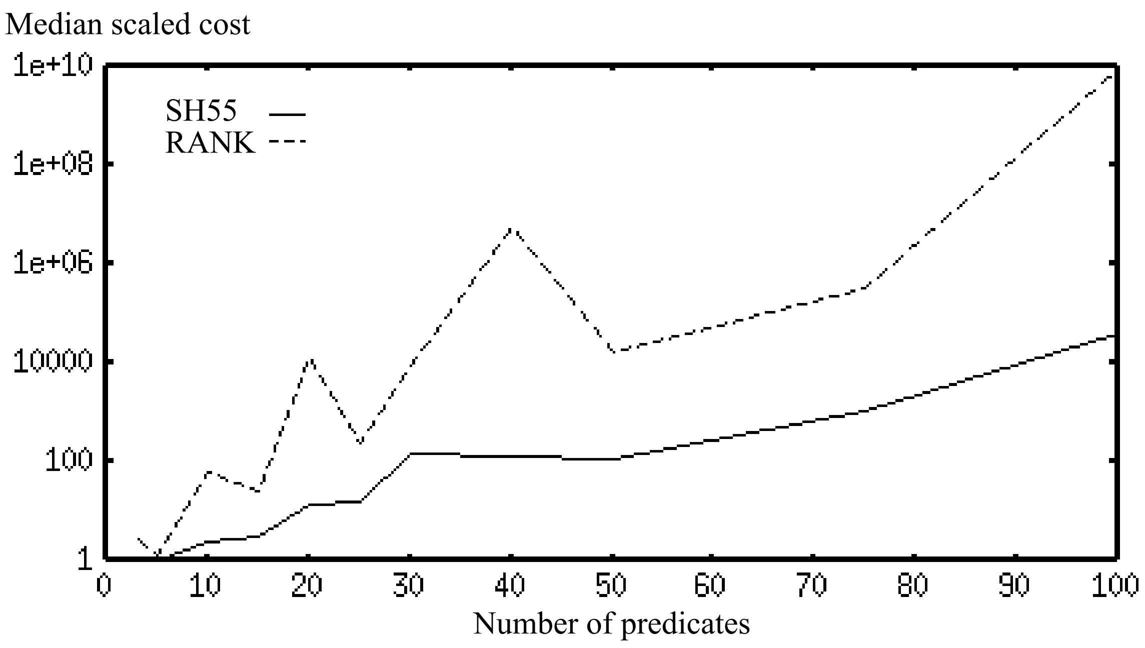


Figure 20. Median cost of the output strategies of Ranksort and SH55.

The SH55 algorithm was chosen to be the third optimization algorithm in WS-IRIS. Figure 20 shows the median scaled cost of the output strategies of Ranksort and SH55. Ranksort performs much worse than SH55 and has a much more unforeseeable behaviour. The difference between the two algorithms is so large that the Ranksort algorithm often gives a result more than 100 times worse than SH55.

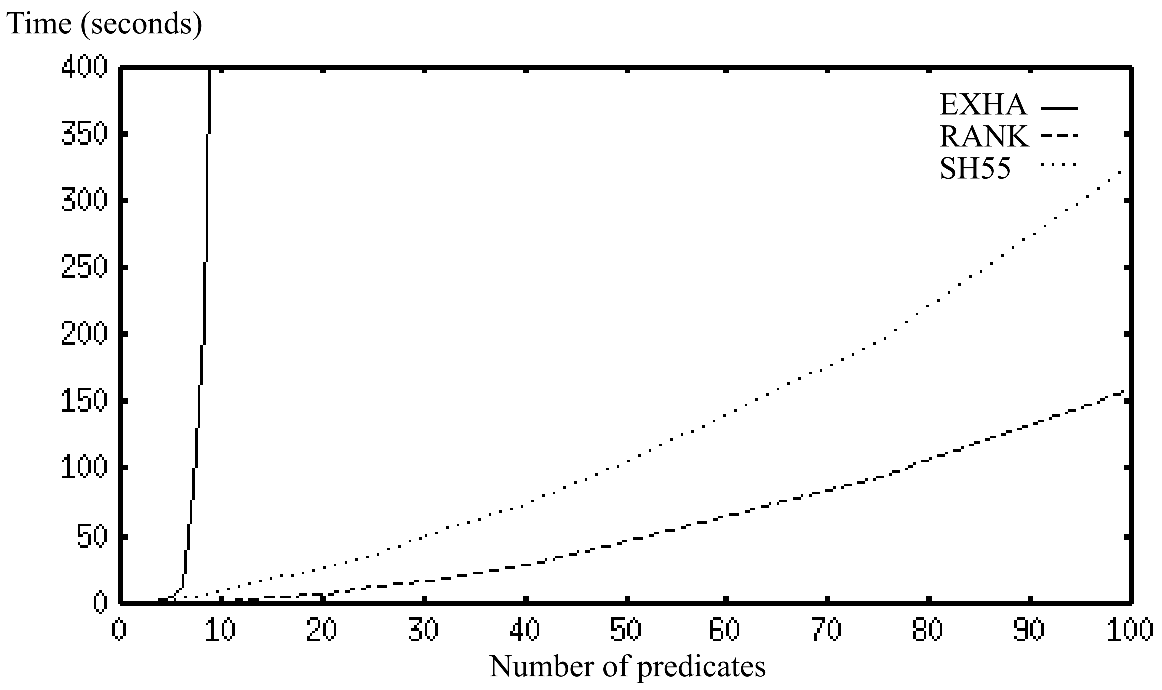


Figure 21. Average optimization time

Ranksort has one advantage, figure 21 shows the average optimization time, the Ranksort algorithm is twice as fast as SH55. The figure also illustrates why it is impossible to use the Exhaustive algorithm on large queries.

8 Implementation

The query optimizer is an independent module in WS-IRIS. The integration of the new optimization algorithm with WS-IRIS went smooth without any problems.

Some special implementation details that improved the optimization time of the algorithms were:

- A hashtable was used for the cost values, i.e. the cost and fanout for a predicate with a list of bound variables were cached in a hashtable. This means that the values only need to be calculated once.
- A state is represented as an array. Many moves made by the randomized algorithms are made by swapping places of two predicates, this is much faster with an array than with a list.

With the above specified methods the optimization time of the Ranksort algorithm can also be reduced.

WS-IRIS is extended so that the new randomized optimization algorithm can be globally chosen with the command:
`optmethod(”randomopt”);`
in the WS-OSQL interpreter.

The number of iterations the algorithm does is default set to five local minimum with Iterative Improvement and five local minimum with Sequence Heuristics. The number of iterations can by the user be chosen with the command:
`optlevel(II-number,SH-number);`
where `II-number` is the number of iterations with Iterative Improvement and `SH-number` is the number of iterations with Sequence Heuristics.

9 Summary and future work

The results in this report shows that the combination of Iterative Improvement and Sequence Heuristics is the best suited optimization algorithm for the WS-IRIS DBMS. The existing algorithms Ranksort and Exhaustive have both problems that lead to a bad behaviour. Ranksort gives far too often very bad solutions and Exhaustive with its exponentially optimization time and memory requirement is often unusable. The randomized algorithms have other advantages, in a realtime database the algorithms can be given a time limit or be interrupted and still have a solution.

In the test the Simulated Annealing algorithm performed badly. The bad result for SA and for the Two-phase algorithm can be explained with a simple neighbour function, defining few neighbours. It is simple because WS-IRIS uses the pipelined nested-loop join method as the only join method. Few neighbours leads to a high probability of choosing a move back to a lower cost state after a move to a higher cost state, instead of making another uphill move. This will also affect the second phase of the Two-phase algorithm. It gets the rather good performance from phase one, the Iterative Improvement phase.

The importance of a good cost model has also been shown. In order for the query optimization to be of any use the optimization algorithm must be able to distinguish bad solutions from good ones.

Future work

- The median scaled cost of the output strategies showed not to be an increasing function with the number of predicates which was expected. This can be explained that the number of tested queries are too few so a larger amount of queries must be optimized before any certain conclusions can be made.
- The result could have been different with another neighbour function, mainly because the amount of neighbours to a state became so limited. This could perhaps effect the SA algorithm positively, making it move around a little bit more.
- The test of the algorithms was done without any use of heuristics. The use of heuristics can improve the performance of the optimization[Swam 89].
- The cost model can be further improved.
- One approach to a better optimization could be an optimization process in the background unknown by the user. This would not affect the user much and the optimizer could be allowed much longer optimization time.
- The Iterative Improvement and Sequence Heuristics algorithms are very suited for parallel optimization and one way of increasing the speed of the optimization could be a parallel implementation of the algorithm.

10 References

[Elma89] Elmasri R. and Navathe S. B., (1989), *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California.

[Fish89] Fishman D. H., Annevelink J., Chow E., Connors T., Davis J. W., Hasan W., Hoch C. G., Kent W., Leichner S., Lyngbaek P., Mahbod B., Neimat M. A., Risch T., Shan M. C. and Wilkinson W. K., (1989), *Overview of the Iris DBMS, Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley.

[Gray91] Gray J., (1991), *The benchmark handbook for database and transaction processing systems*, Morgan Kaufmann Publishers Inc., San Mateo, California.

[Ioan90] Ioannidis Y. E. and Kang Y. C., (1990), *Randomized Algorithms for Optimizing large join queries*, *Proceedings of the 1990 ACM-SIGMOD Conference*, Atlantic City, pp 312-321.

[Kris86] Krishnamurthy R., Boral H. and Zaniolo C., (1986), *Optimization of Nonrecursive Queries, Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, pp 128-137.

[Litw92] Litwin W. and Risch T.,(1992), *Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates*, Linköping University, Lith-IDA-R-92-24.

[Risc93a] Risch T., (1993), *WS-IRIS User's Guide*, Internal report, IDA, Linköping University.

[Risc93b] Risch T., (1993), *WS-IRIS Advanced Programmer's Manual*, Internal report, IDA, Linköping University.

[Seli79] Selinger P. G., Astrahan M. M., Chamberlin D. D., Lorie R. A. and Prince T. G., (1979), *Access Path Selection in a Relational Database Management System*, *Proceedings of the 1979 ACM-SIGMOD Conference*, Boston, pp 23-34.

[Swam88] Swami A. and Gupta A., (1988), *Optimization of Large Join Queries, Proceedings of the 1988 ACM-SIGMOD Conference*, Chicago, pp 8-17.

[Swam89] Swami A., (1989), *Optimization of Large Join Queries*, Department of computer Science, Stanford University, STAN-CS-89-1262

[Ullm89] Ullman J. D., (1989), *Principles of Database and Knowledge-Base Systems volume 2*, Computer Science Press, Inc., Rockville, USA

[Whan90] Whang K. Y. and Krishnamurthy R., (1990), *Query Optimization in a Memory-Resident Domain Relational Calculus Database System, acm Transactions on Database Systems*, volume 15, number 1, ACM press, New York.

Index

Appendix A: The database

The generated database described in section 7.1 received the following contents.

Table	Cardinality	Number of columns	Distinct values in each column	Index on column
0	86	2	1, 0.02289	0
1	94	2	0.929, 1	0
2	64	3	0.928, 0.808, 1	0
3	59	2	0.929, 0.18607	0
4	15	2	0.859, 0.896	0
5	33	4	0.921, 0.011, 0.031, 0.043	0
6	299	2	0.857, 0.04279	0
7	575	3	0.939, 1, 0.02488	0
8	440	4	1, 1, 0.1204, 1	0, 2
9	766	5	0.881, 0.170, 0.063, 0.816, 0.182	0
10	869	4	1, 0.00697, 0.08259, 0.832	0
11	643	4	1, 0.0408, 0.09254, 1	0
12	149	2	1, 0.1403	0, 1
13	353	3	0.923, 0.17413, 0.14627	0
14	262	2	1, 0.044788	0
15	122	4	1, 1, 0.19005,0.15423	0
16	832	2	1, 0.02687	0
17	500	2	1, 0.01493	0
18	864	4	1, 0.07463, 1, 1	0
19	938	4	0.863, 0.03085, 0.13831, 0.07065	0
20	120	3	0.931, 0.07065, 1	0
21	977	2	1, 1	0
22	7653	10	1, 0.19, 0.14, 1, 0.099, 0.16, 0.083,1, 0.97, 1	0, 6
23	3377	2	0.87, 0.08458	0
24	4045	3	1, 0.05274, 0.05672	0

Appendix B:Test results

This appendix contains a closer look at the test results from the test presented in chapter 7. The first table shows the result when comparing the randomized algorithms and the Ranksort algorithm to each other and the second is a comparison between Ranksort and SH55.

In the table below the scaled cost of the output strategy of a query for two algorithms have been compared. In the test 110 queries have been optimized and every combination of the results produced by the algorithms have been compared. A cell in the table contains three values. The first value is the number of times the vertical algorithm is superior, the middle value is the number of times the two algorithms is considered equal and the last value is the number of times the horizontal algorithm is superior.

For example, the comparison between the II5 and the SH37 algorithms shows that II5 is superior 24 times, SH37 is superior 58 times and they have equal results 28 times. The results are considered to be equal if the costs of the result of the two algorithms do not differ more than 10%.

	TWO	II5	II10	SA	SH37	SH55
II5	32-28-50					
II10	49-35-26	63-32-15				
SA	0-09-101	01-10-99	0-09-101			
SH37	47-29-34	58-28-24	31-35-44	100-09-1		
SH55	53-31-26	61-30-19	39-35-36	100-10-0	39-37-34	
RANK	25-14-71	28-14-68	14-14-82	86-06-18	19-15-76	13-18-79

The randomized algorithms can with the values in the table be ordered by their results. The order will be SH55, II10, SH37, TWO, II5, RANK and SA with the best performing algorithm first.

In the table below the scaled cost of the Ranksort and the SH55 algorithms have been compared to each other. The table shows the result for each size of query and how much the results differ. The values in the table are the number of times the quotient (cost Ranksort)/(cost SH55) is in the interval for the column. For example the values in the >10 column are the number of times the quotient are in the interval]100..10]. The results are considered to be equal if the results do not differ more than 10%.

N	>1000	>100	>10	>1.1	1.1-0.9	< 0.9	< 0.1	< 0.01	< 0.001
3	0	0	3	2	5	0	0	0	0
5	0	1	0	2	6	1	0	0	0
10	0	0	5	4	1	0	0	0	0
15	1	1	1	6	1	0	0	0	0
20	4	0	0	6	0	0	0	0	0
25	1	2	0	4	2	1	0	0	0
30	1	4	0	1	2	1	1	0	0
40	6	2	1	0	0	0	1	0	0
50	3	2	0	2	1	2	0	0	0
75	4	1	2	0	0	0	1	1	1
100	6	0	0	1	0	2	1	0	0
Total	26	13	12	28	18	7	4	1	1

The Ranksort algorithm is very rarely superior of the SH55 algorithm. The SH55 algorithm are superior in 72% of the queries, Ranksort in 12%. It is notable that in 24% of the cases the Ranksort algorithm produces a result more than 1000 times worse than the SH55 algorithm. The Ranksort algorithm can thereby not be said to fulfil the condition of a well performing query optimization algorithm.

Appendix C:Examples

In this appendix three examples are presented. All of them use the same OSQL functions and the database is populated in the same way in all examples. It is recommended for better understanding that chapter 3, in this report, has been read before you read this appendix.

```
create type person;
create type student subtype of person;
create type employee subtype of person;
```

The database consists of 1100 persons, 1000 employees and 100 students in the examples with the `test1` function, when the `test2` function is used the database is populated with ten times more employees and students. Each employee has a unique name and an income in the range [0..200000]. Five of the employees are bosses of some department. Each student has a unique name and is attended to a group. All students attend the same group, the C-group. All student has a father chosen from the employees. The student type also contains the name and grade of the courses a student has taken. Each student can have up to ten courses attached to him.

```
create function name(person p) -> charstring c as stored;
create function clases(student s nonkey) ->
   <charstring c1, charstring c2> as stored;
create function line(student s) -> charstring c as stored;
create function income(employee p)->integer i as stored;
create function father(person p) -> charstring c as stored;
create function boss(employee e) -> charstring c as stored;
create function who_earn(integer i) -> employee e as select
   e where income(e)=i;
```

Function `test1` returns the names of the employees that earns more than 50000 dollars and that is father of a student in C-group that has taken the class databases with grade A.

```
create function test1() -> charstring as select
   n for each employee p, student s,charstring n where
       group(s)=”C” and
       courses(s)=<”Databases”,”A”> and
       n=father(s) and
       n=name(p) and
       income(p)>50000;
```

Function `test2` returns the name of the department of a boss that has a specific income.

```
create function test2(integer i) -> charstring c as select
   c for each employee e where
       who_earn(i)=e and
       boss(e)=c
```

The need of query optimization

Without a query optimizer the query will be executed in the order the user has written the query. In this example two versions of the `test1` function has been executed. The query has not been optimized so the execution order will be different in the two cases. In this example the database contains 1000 employees and 100 students.

```
create function test1() -> charstring as select
    n for each employee p, student s,charstring n where
         n=father(s) and
         income(p)> 50000;
         courses(s)=<”Databases”,”A”> and
         group(s)=”C” and
         n=name(p);
```

The CPU time of executing the function was 91.25 seconds.

The reason of the long execution time is that many tuples matches `father(s)` and `income(p)`. 100 tuples matches `father(s)` and 1000 tuples matches `income(p)`. This has the effect that the rest of the conditions have to be tried many times. For example the condition directly following the two first has to be checked 100*1000=100000 times. The same functions defined with a new order of the conditions.

```
create function test1() -> charstring as select
    n for each employee p, student s,charstring n where
         courses(s)=<”Databases”,”A”> and
         group(s)=”C” and
         n=father(s) and
         n=name(p) and
         income(p)> 50000;
```

The execution time in this case was 0.38 seconds, an improvement of 240 times in speed. In this case the number of matching tuples of the first predicates are much lower, leading to savings in execution time. The result of the two functions are the same. The function used in this example is a small function with only five conditions, the differences in execution time between different permutations of a larger query can be much larger.

When Ranksort performs poorly

There is one problem with the Ranksort algorithm, it sometimes performs poorly. This example shows when the algorithm is not to our satisfaction. The example contains only one join, so the optimal alternative, the Exhaustive algorithm, would cope without any time problems. But this problem can occur as part of a much larger query when an exhaustive search is impossible. In this example function `test2` is used and the database is populated with 10000 employees and 5 of them are bosses. The function contains two conditions `who_earn(i)=e` and `boss(e)=c`. the first condition is the reverse of function income. This has the effect that the index is on the variable e instead of the bound variable i.

- An integer i is given to the function.
- The condition `who_earn(i)=e` can be substituted for `income(e)=i` that has an index on variable e. The relation contains 10000 tuples.
- The relation `boss` contains 5 tuples.

```
create function test2(integer i) -> charstring c as select
    c for each employee e where
         who_earn(i)=e and
         boss(e)=c
```

The ObjectLog program will be:

((OID[P_EMPLOYEE.INCOME:200] E I)
(OID[P_EMPLOYEE.BOSS:212] E C))

The Ranksort algorithm works in the following way: For each predicate in the ObjectLog program calculate a rank and choose the lowest rank.

P_1=**(OID[P_EMPLOYEE.INCOME:200] E I)**
$F_1=99.0594$, $C_1=20010$
$R_1=(F_1-1)/C_1=0.00490052$

P_2=**(OID[P_EMPLOYEE.BOSS:212] E C)**
$F_2=5$, $C_2=10$
$R_2=0.4$

The algorithm chooses the first predicate. The variable E will be bound and the second predicate are calculated.

P_1=**(OID[P_EMPLOYEE.BOSS:212] E C)**
$F_1=1.$ $C_1=2$
$R_1=0$

The optimized ObjectLog program will be

```
((OID[P_EMPLOYEE.INCOME:200] E I)
 (OID[P_EMPLOYEE.BOSS:212] E C))
```

The total cost can now be calculated according to the cost function in section 6.5.

$C=C_1+F_1*C_2=20010+99.0594*2=20208.1$

The execution time of this ObjectLog program are 0.22 seconds

The `test2` function optimized with the Exhaustive algorithm came up with the following ObjectLog program

```
((OID[P_EMPLOYEE.BOSS:212] E C)
 (OID[P_EMPLOYEE.INCOME:200] E I))
```

$F_1=5$ $C_1=10$
$C_2=2$

The total cost will be 10+5*2=20.

The execution time was 0.02 seconds, more than 10 times faster than the ObjectLog program Ranksort came up with. This is a small example the same situation can occur as part of a larger query leading to a much worse result than 10 times slower. It should be noted that the naive heuristics of the selection-pushing optimization method [Elma89] and the bound-is-easier method [Ullm89] will generate the same suboptimal execution strategy.

The need of a good cost model

This example shows that even the best query optimizer cannot do a good optimization if the cost function is wrong. With the old cost function the `test1` function was optimized with the Exhaustive algorithm.
The optimal execution plan:

```
((OID[”P_STUDENT.GROUP”:160] S ”C”)
 (OID[”P_PERSON.FATHER”:166] S N)
 (OID[”P_PERSON.NAME”:154] P N)
 (OID[”P_EMPLOYEE.INCOME”:163] P _G3)
 (OID[”OBJECT.>”:64] _G3 50000)
 (OID[”P_STUDENT.COURSES”:157] S ”Databases” ”A”))
```

cost=9233.79
The ObjectLog program is below shown with one predicate at the time.

(OID[”P_STUDENT.GROUP”:160] S ”C”)
Bound variables=NIL, Cost= 200, Fanout=4 (100)
(OID[”P_PERSON.FATHER”:166] S N)
Bound variables=(S), Cost=2, Fanout=1
(OID[”P_PERSON.NAME”:154] P N)
Bound variables=(N S), Cost=2210, Fanout=4 (1)
(OID[”P_EMPLOYEE.INCOME”:163] P _G3)
Bound variables=(P N S), Cost=2, Fanout=1
(OID[”OBJECT.>”:64] _G3 50000)
Bound variable=(_G3 P N S), Cost=1, Fanout=1
(OID[”P_STUDENT.COURSES”:157] S ”Databases” ”A”))
Bound variables=(_G3 P N S), Cost=8.61176, Fanout=4 (0.16)

The execution time with this permutation was 2.75 seconds.

A random state was computed with a much higher cost than the optimal solution.
The execution plan.

```
((OID[”P_PERSON.FATHER”:166] S N)
 (OID[”P_STUDENT.COURSES”:157] S ”Databases” ”A”)
 (OID[”P_STUDENT.GROUP”:160] S ”C”)
 (OID[”P_PERSON.NAME”:154] P N)
 (OID[”P_EMPLOYEE.INCOME”:163] P _G3)
 (OID[”OBJECT.>”:64] _G3 50000))
```

cost=890661.

A closer look at each predicate.

((OID[”P_PERSON.FATHER”:166] S N)
Bound variable=NIL, Cost=200, Fanout=100
(OID[”P_STUDENT.COURSES”:157] S ”Databases” ”A”)

Bound variable=(N S), Cost=8.61176, Fanout=4 (0.16)

```
(OID[”P_STUDENT.GROUP”:160] S ”C”)
```

Bound variable= (N S), Cost=2, Fanout=1

```
(OID[”P_PERSON.NAME”:154] P N)
```

Bound variable=(N S), Cost=2210, Fanout=4 (1)

```
(OID[”P_EMPLOYEE.INCOME”:163] P _G3)
```

Bound variable=(P N S), Cost=2, Fanout=1

```
(OID[”OBJECT.>”:64] _G3 50000))
```

Bound variable=(_G3 P N S), Cost=1, Fanout=1

With this permutation the execution time was 0.62 seconds. The random state has a cost almost 100 times higher than the optimal solution, but with an execution time more than 4 times faster than the state with the lowest cost.

The reason of the bad behaviour of the Exhaustive algorithm is the default value 4 of the fanout when an index is missing. The values 100, 1, 0.16 are the real values that should have been used instead of a default value. The correct value of the total cost would then be 222561.

With the new cost function the same function, `test1`, is optimized again with the Exhaustive algorithm. The optimal ObjectLog program will be:

```
((OID[P_STUDENT.COURSES:157] S ”Databases” ”A”)
 (OID[P_STUDENT.GROUP:160] S ”C”)
 (OID[P_PERSON.FATHER:166] S N)
 (OID[P_PERSON.NAME:154] P N)
 (OID[P_EMPLOYEE.INCOME:163] P _G3)
 (CALL OBJECT.>-- OID[OBJECT.>:64] _G3 50000))
```

The total cost is now 28143.2 and the execution time is 0.60 seconds

The total cost of the random state is calculated again with the new cost function and the new cost is 36501.