BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M15

"EREQ Query Representation and Cost Model"

by

Andrew Clement Thornton MacKeith

EREQ Query Representation and Cost Model

Andrew Clement Thornton MacKeith
B. Sc., University of Leeds (England), 1968.

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

May 1993

This research project by Andrew C. T. MacKeith is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science

Date 4/30/93

Stanley B. Zdonik

Stanley B. Zdonik

EREQ Query Representation and Cost Model

by

Andrew C. T. MacKeith
Computer Science Department
Brown University
e-mail: acm@cs.brown.edu
6 May 1993

ABSTRACT

This paper describes the implementation of a Tree Representation of a query in the Equal Query Language. The tree is annotated with data such as local variables, and cost of the sub query (in disk accesses). The cost functions used for these annotations are based on the paper "An Analytical Model of Object-Oriented Query Costs" by E. Bertino and P. Foscoli, March 30, 1992. This has been adapted for used with the above Tree Representation of an Equal Query.

Contents

Tables and Figures

Appendices

1.0 Introduction

This paper describes the implementation of sections of the EREQ project. The three parts dealt with in this paper are:

- A new query representation to be used by the EREQ query optimizer.
- A revised Schema Manager, and
- A cost model for the optimizer to be used in conjunction with that representation.

The EREQ query rep is a C++ model and is an extension of a previous model written in C.

Experiments have been done using the cost model to show how the cost of a query varies with respect to the various parameters input to the cost model.

1.1 The EREQ query optimizer

The Encore Revelation Exodus Query (**EREQ**) project is a current joint project between Brown University, Oregon Graduate Institute, and University of Wisconsin, Madison. The project aims to produce an Object Oriented Query Language and two optimizers, one at Brown and one at Oregon Graduate Institute. The Optimizer being built at Brown is called EPOQ.

2.0 Related Work

2.0.1 Query Representation

This work on the query representation follows work done by Gail Mitchell and Ted Leung. The query representation is described in chapter 7 of [MITCH93].

A previous implementation was made by George Lo, and is described in [LO92]. This implementation is used directly by the new EREQ rep described here.

2.0.2 Cost Model

The cost model described in this paper is an implementation of the model described by Elisa Bertino and Paola Foscoli in [BERT92] and [BERT93].

3.0 Implementation

3.1 Class and Type

Please note that In this paper and in the program, I use the term "`type`" to refer to a type in the database. Bertino uses the term "class" for this. I have used the term "`class`" to refer to c++ classes only.

3.1.1 USLC Class Library.

I have used two C++ classes from the USLC class library.

The first is Map. This class provides an extensible associative array structure which we have used for the query tree annotations.

The second is USLString. This provides a class for strings. At present this is only as the key for the Map, but it would seem to be a useful class to use generally for strings.

3.1.2 Multivalued Attributes

There are several different structures that a database schema may use for a multivalued object. The most common description of on of these is a set. I have tried not to use the description "set" since this can be confused with "set value" as a verb. The description "multivalued attribute" therefore is meant to cover set, list, tree, array or any collections of objects in an attribute.

3.1.3 Primitive Types

Primitive types are defined in [BERT92] as Integer, Float, String and Boolean. (Paragraph 2, Definition 4).

3.2 EREQ Query Rep

3.2.1 Previous Work

Work has been done previously on the parser and tree builder by George C. Lo (**gcl**) in 1992. This formed the basis of the tree which has been used to build the current query rep as described here. In this paper I call this previous tree representation "qTree rep".
This previous work provided an API which was incorporated in a library called libRep.a.

The library had been used to implement the following programs:

- "table". This could be used to build a data dictionary to describe the schema of the database being queried.
- "opt" This program calls an optimizer generated by the Exodus optimizer generator, and displays the query tree in graphical form in an X-window. This is based on the API functions createQueryTree() and printTree(). The displayed query tree is of the qTree type.

3.2.2 Current Work

The current work has included the following:

- Amendments and additions to previous program.

Several changes and additions to the previous work have been made. These have not changed the functionality of the qTree API, but have been done for two reasons: 1. To avoid duplication of functions in different programs, and 2. to provide additional functions to access the data dictionaries created by gcl. The original program "table" has been rebuilt and renamed "OptRepTable". Most of the functionality provided by the functions in table (functions also in the library libOptRep.a) are now provided by the class SchemaMgr.

- Implementation of new Query Rep as proposed by T. W. Leung.

The aim of the new query rep is to provide an Object Oriented version of the query tree (an EAT or Equal Annotated Tree) which can be used in the extensible optimizer now being built at Brown.
The qTree representation of the query tree has been used as a basis for the new Query Rep, to avoid having to rebuild the query parser. The new rep uses the previous model as a basis, so the parsing of the query is done by the functions that build the old rep.

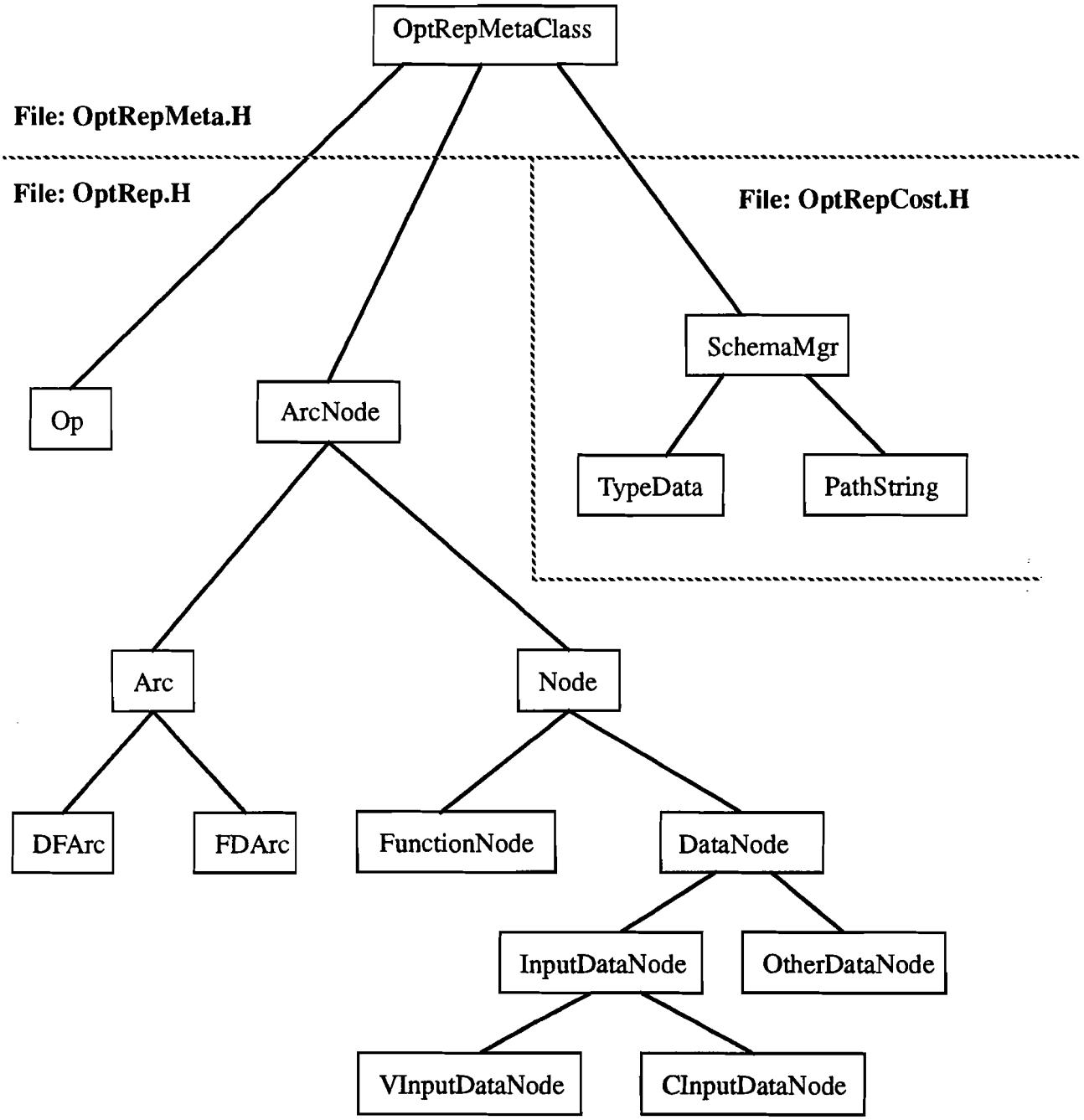


Figure 1: C++ Class diagram for the Rep and SchemaMgr classes

Features of the new Rep are:

1. The Annotations are now fully extensible.
2. Arcs between Nodes are now objects and have Annotations.
3. Type checking of the graph is accomplished through the use of the C++ type checking mechanism.
4. New Annotations have been added: "path" Annotation, and "cost" Annotation.
5. An additional field of the DataNode enables the size of a multivalued variable (ie a SET) to be stored. This information is required for cost calculations.
6. Methods of the new Rep include methods to build Query Paths of the kind used by Bertino. These query paths are described in more detail below under Cost Model, but simply, a query path is of the form VariableName.AttributeName. This can be extended as far as the schema allows such as V.A1.A2.A3.A4, until a primitive type is reached. The function BuildQueryPaths() assumes that a FunctionNode operator that is NOT predefined is an attribute of a type.

3.2.3 Annotations

Annotations may be of any type. Previously the qTree only supported a LambdaVar type in the Annotation list. The annotation list is now implemented as an Associative Array of void*, via a Map structure. This provides a fully extensible means of adding annotations using a string "name" as key.

One of the problems with the void* representation is that the data cannot be copied by a copy constructor, neither can the data be deleted by a destructor.

A special class has been defined to use with Annotations, which is similar to a very simple list node, but including the size of the data object of the list element. This will allow copying and deletion of an Annotation List without knowing anything about the list except the size of the data block in the list, however this has not been implemented.

New Annotations have been added: "path" Annotation, and "cost" Annotation. Functions exist in the Rep to manipulate these annotations, which questions the assumption that the annotations should be of type void*. However, this question is left open.

3.2.4 Build Query Paths

There is a method of this name in the classes FunctionNode, DataNode, FDArc and DFArc. In each one the principle is the same but the method has a different function. The tree is searched from the root for the Query Paths, and the Paths are built as the tree is traversed. If the constructor DataNode::DataNode(USLString) is called with no second argument, no paths are built, and no cost data is calculated. If the constructor is called with a second argument of TRUE then the paths and the costs are annotated to the tree. This allows the tree to be built without having a file of cost data available.

The methods in the **Arcs** just call the BuildQueryPaths() function of the Child and then copy the path and cost annotations to themselves from the child. The methods in the **Nodes** are more complex, but in principle, the DataNode::BuildQueryPaths() builds the paths, and FunctionNode::BuildQueryPaths() calculates the costs that cannot be directly calculated by the PathString::getCost() function.

A Path is only built for a sub-tree below a FunctionNode that is an attribute, or for a leaf node that is a global variable.

3.2.5 C++ Class descriptions

The class definitions are in Appendix A. Function descriptions are in Appendix B.

3.3 Schema Manager.

3.3.1 Function.

The Schema Manager is used to find details of any type in the database. The type details stored in the schema manager for each type T include

- Structural details such as supertype and subtype of T, and attributes of T (and their types).
- Cost parameters used for calculating cost of a Path, as described below. See paragraph 3.4.5 The Query Path. on page 10.

The principle is that the class SchemaMgr should replace the functions of the file "OptRepTable" previously written by gcl. Class TypeData and PathString are both subClasses of class SchemaMgr. The type names, attribute details and variable names are taken from the "gcl" data dictionaries.

3.3.2 Previous work.

The previous query rep and current optimizer use a schema manager written by gcl. The functionality of this formaer schema manager is implemented in the files `OptRepTable.c` and `OptRepCompute.c`. The "gcl" data dictionaries are named `.type_dict`, `.attr_dict` and `.data_dict`. The internal "gcl" data dictionary has to be initialized by calling the function readInBuf() which reads the `*_dict` files in the current directory. This is done implicitly by calling `createQueryTree()`.

3.3.3 Class Descriptions.

The Schema Manager used for the cost functions is composed of three classes: `SchemaMgr`, `TypeData`, and `PathString`.

The Functions of these classes are briefly:

`SchemaMgr`: provide a reference table of type data.

`TypeData`: provides a structure for data relating to each type.

`PathString`: calculates all the functions specified in the cost model of [BERT92].

3.3.4 Sequence of Operations.

The Schema Manager is not activated until the SchemaMgr constructor is called. This is called by the constructor of any subclass of SchemaMgr, ie classes TypeData or PathString. The first call to the SchemaMgr constructor calls the function buildTypeMap(), which builds the associative array typeMap that provides the reference for all information on the types in the schema via the type name. The typeMap is a structure of class Map(USLString), TypeDataPtr).

3.4 Cost Model.

3.4.1 Basis

The basis of the cost model is the paper entitled "An Analytical Model of Object-Oriented Query Costs" by E. Bertino and P.Foscoli dated March 30, 1992 [BERT91].

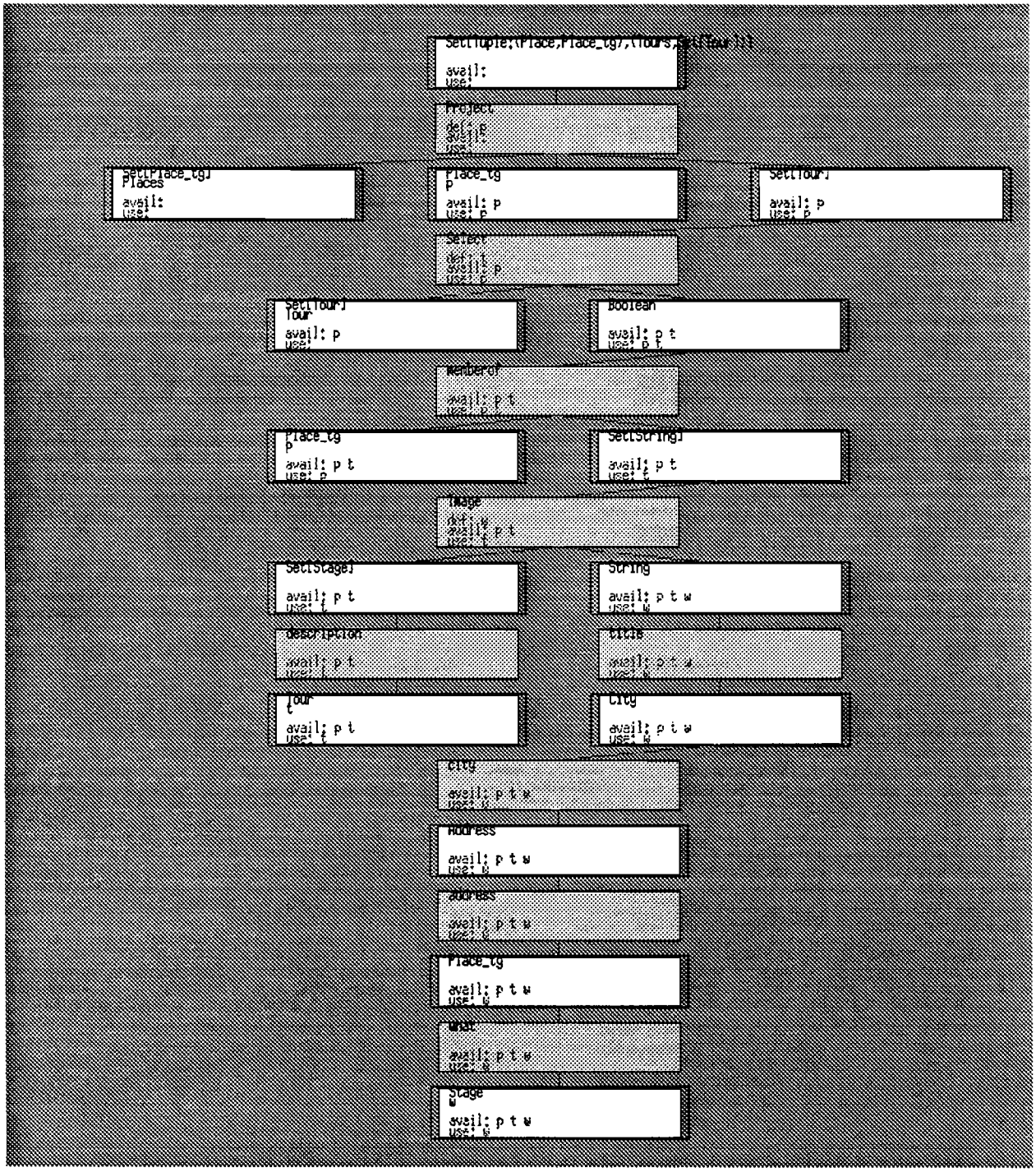


Figure 2: qTree Query Tree as produced for the optimizer

Query text:

Project(Places,$p{(Place,p),

(Tours,Select(Tour,$tp memberof Image(t@description,$w w@what@address@city@title)))})

3.4.2 Use of the model

The cost functions described in [BERT92] are for evaluating a nested predicate of the form:

C1.A1.A2.....An. *op exp*.

I have assumed that the operator is of a form that requires equality, rather than a range to be resolved. Without this constraint the number of objects which resolve the query is not determinesd

3.4.3 Class and Function names

The function names which refer to the parameter fields of the PathString class have been kept as close as possible to the names used by Bertino. The class which holds information about each type is called **`TypeData.`** This is roughly equivalent to the reference **`Cij`** adopted by Bertino. (See 3.1 Class and Type on page 4.)

The reference method to the types of the attributes in the path is by indexing a two dimensional array which is built whenever a **`PathString`** is built. The cost parameters are generally of the form X(i) or X(i,j), where i refers to the type of attribute (i-1) in the path. The path is of length n since are n attributes in the path. If the type of the last attribute is included, we have n+1 types. Bertino does not include the type of the last attribute This is always a primitive type in her queries. I have included it in my **`Path-String`**, since the last attribute of a path taken from the EAT may have a non-primitive type. This data is not used at present in this implementation. The access functions to the cost parameters are methods of the class **`PathString`**, but there are parallel methods in the class **`TypeData`**. Where possible, if a parameter depends on more than one type, the access method calculates the parameter when it is called. For example N(i), which is a function of all the types in the type hierarchy rooted at C(i,1).

A complete class listing is given in appendix A on page 27.

3.4.4 Primitive Types

The end of a path as described in [BERT92] is always a primitive type. The paths that I consider do not always keep to this restriction. A **primitive type** is a Integer, Float, String, or Boolean type.

3.4.5 The Query Path.

A Path is a string describing a *type* plus zero or more *attributes*. The length of the path is the number of attributes in the path. Note that the type of Attribute(i) in the functions and in [BERT91] is C(i+1,1). The definition of a path is shown below. The Type of each element of the path is shown under each element. C(i,1) refers to a Type. Generally, C(i) refers to the type C(i,1).

Type . Attribute(1). Attribute(2) . . . Attribute(i) . . . Attribute(n-1) . Attribute(n).

C(1,1) . C(2,1) . C(3,1) . . . C(i+1,1) . . . C(n,1).

For the purpose of the EREQ query tree I have interpreted the Path as variable plus attribute(1) etc. The variable at the start of the path is a variable named in a leaf DataNode of the tree.

3.4.6 Cost Parameters.

A full list of the Cost Parameters is given in the table on page 17. Some of the parameters have been eliminated by making assumptions about the nature of the database schema. These assumptions are listed below.

Assumption 1. Number of pages containing members of the type C(i) is assumed to be the sum of the number of pages containing instances of each type in the hierarchy. This assumes *the same clustering* of instances in the hierarchy based on a class as the instances of the base class. Therefore $Ph(i) = \Sigma_j P(i,j)$.

Assumption 2. Number of distinct values for attribute A(i) for all instances in the inheritance hierarchy rooted at type C(i,1) is assumed to be sum of those for each type in this hierarchy. Therefore $D(i) = \Sigma_j D(i,j)$.

Assumption 3. The average size of a multivalued attribute over the members of a type is assumed to be the average over the average size for each of the constituent types. $Fan(i) = (\Sigma_j fan(i,j) * N(i,j)) / (\Sigma_j N(i,j))$.

These assumptions reduce the number of statistical parameters supplied to 2 per Type plus 4 x number of attributes. (N(i,j), P(i,j), plus a value of D(i,j), fan(i,j), d(i,j), and r(i) for each attribute of the Type.)

3.4.7 Cost Parameters D, fan, d and r.

D(i,j), fan(i,j) and d(i,j). These parameter can be different for each attribute of a type. For example, with reference to the above path, C(1,1) has an attribute Attribute(1) of type C(2,1). The value of fan(1,1) in this case is the *collection size* of Attribute(1) (See "Multivalued Attributes" on page 5.). If A is *not multivalued,* the value of fan(1,1) is 1. As can be seen, the value of fan(i,j) depends on the attributes in the particular path..

r(i). This parameter is a Boolean, TRUE if there exists a "reverse" reference from type C(i) to type C(i-1). Just as the average Set Size has to be stored for each attribute of a type, there could be a reverse reference from type T to (theoretically) any other type in the schema. Since it would only be interesting for references (connections between types) that might exist in a path, I have stored the reverse reference as a parameter of the type referenced. (ie the type that the reference points to rather than the referencing type.) Type C(i) must therefore ask C(i-1) to find out if there is a reverse reference to C(i-1).

3.4.8 Constraints on Cost Parameters

There are several constraints that must be checked if values for the cost parameters are invented for experiments. These are:

d is the number of instances of C(i,j) with no NULL vales for A(i). This cannot be greater than the number of instances of the type, the cardinality Nij. Therefore:

d(i,j) <= N(i,j).

P is number of disk pages containing instances of the type C(i,j). I have assumed that an instance does not occupy more than a disk page; this allows a check to be made as to whether this parameter is sensible.

P(i,j) <= N(i,j)

D is the number of distinct values for Attribute A(i). There cannot be more distinct values than the cardinality of the domain of the attribute. Therefore:

D(i,j) <= Nh(i+1).

Similarly to the above, the if the attribute A(i) is a set (or more generally a collection), then the collection size (or set size) of attribute A(i) cannot be greater than the cardinality of the domain of the attribute. Therefore:

fan(i,j) <= Nh(i+1).

This is not true if duplicates are allowed in a set. I have assumed that they ar not.

The collection size of attribute A(i) cannot be greater than the number of distinct values for attribute A(i). Therefore:

fan(i,j) <= D(i,j)

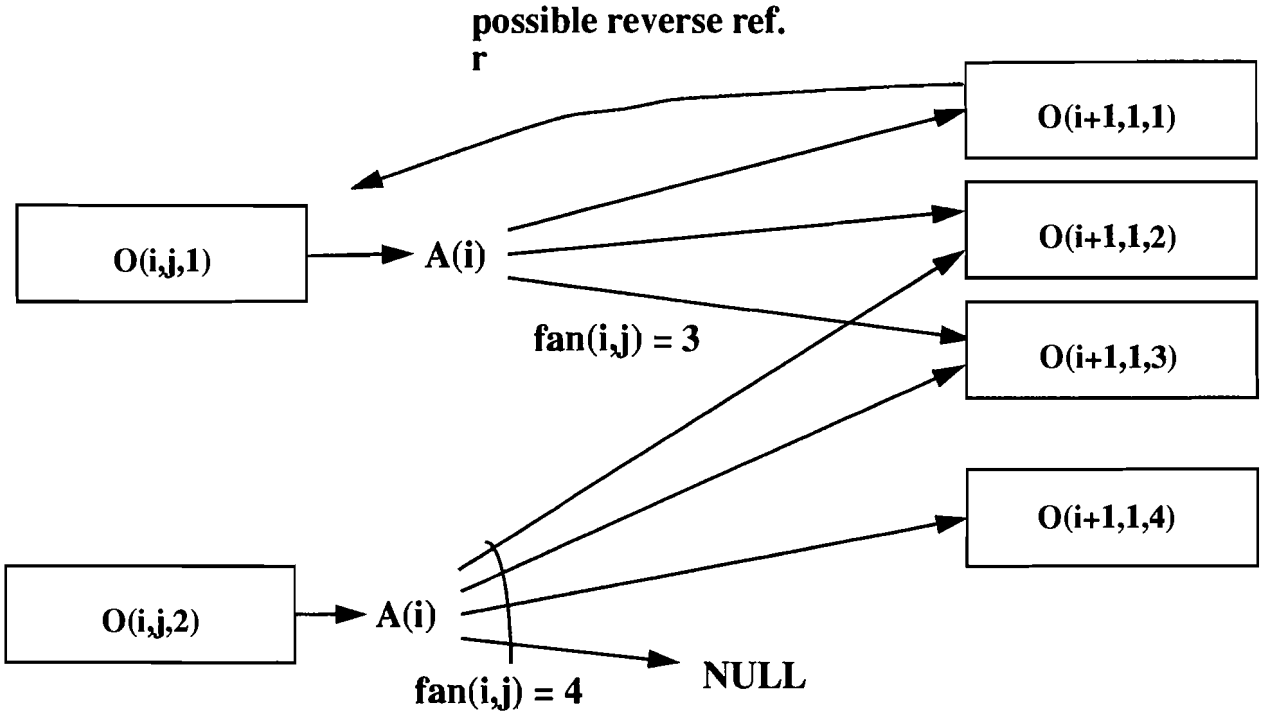
If this is violated, it affects cost functions Pr11() and Pr12), thence RefBy() and RefByh().

These constraints are all checked during the building of a path, with the exception of D() which is checked (together with the others) in function **`PathString::getCost()`**.

The requirement fan<=D, means that another requirement k(i,j)<=d(i,j) is not violated. This last affect the other functions thus:

If k(i,j) > d(i,j), affects cost functions Pr21() and Pr(22) thence Ref() and Refh().

If kh(i) > d(i), affects cost functions Pr31() and Pr32() thence E(), Ref() and kbar().



N(i,j) = 2
D(i,j) = 5
fan(i,j) = (3+4)/2 = 3.5
d(i,j) = N(i,j) = 2

N(i+1,j) = 4

O(i,j,k) indicates an object of type C(i,j).
In this case the RH objects are type O(i+1,1) since they are referenced by A(i).

Figure 3: Graphical Description of Cost Parameters.

3.4.9 Execution Strategy

There are 2 Graph traversal (Forward & Reverse) and 2 Retrieval (Nested Loop and Sort Domain) strategies mentioned in [BERT92] § 3.2, which give 4 basic query execution strategies. The cost functions are different for each one.

Forward Traversal is to compute the cost of accessing the type C(n,1) (as well as its subtypes if applicable) from just the type C(1,1) or from C(1,1) and its subtypes according to the type of the query.

Reverse Traversal is to compute the cost of accessing a type C(1,1) (as well as its subtypes if applicable) from the type C(n,1) and its subtypes.

Nested Loop searches for each attribute in the path in turn using a sequential search loop.

Sort Domain sorts the whole domain of a type and then passes this to the next type in the path to be filtered.

The strategy is given as an argument to the `getCost` function of `PathString`. The 4 strategies are therefore:

NLFT Nested Loop Forward Traversal

NLRT Nested Loop Reverse Traversal

SDFT Sort Domain Forward Traversal

SDRT Sort Domain Reverse Traversal

The SORT() function used in the retrieval method Sort Domain assumes 100 OID's per disk page for the sort. OIDperPage = 100;

In the cost functions the Cardinality of the range of values that resolve the predicate has an effect on the cost. For a predicate of the type "attribute==value", the cardinality is 1, whereas if the relational operator is <, >, <=, >=, the cardinality is unknown. I have therefore assumed a cardinality constant `cardPred` = 1.

3.4.10 Cost Functions.

Details and a brief description of each of the cost functions are given in Appendix B2, page 46.

3.4.11 The Cost Model and the EAT.

On essential difference between the path used Bertino cost model and the path used in the query tree Rep (the EAT), is that in the path built by the EAT, there are no multivalued attributes. For example, if we have a type "Hotel" which has a multivalued attribute "facility", and a variable "TheRitz" of type Hotel. Bertino would allow a path which includes TheRitz.facility. In our Query Tree, the operator Image (or Select) is used to iterate over the values in the set, and the operator Image is represented by a FunctionNode in the Tree. The Path therefore stops before the Image operator.

In the experiments therefore, I have used a Path which is not obtained from the EAT, but built directly from a string representation of a Path.

3.4.12 Cardinality of values that resolve the predicate.

I have assumed a value of 1 for this factor (cardPred=1). This implies Attribute==Value, rather than <,>,<= or >=. Values for the other comparison operators would be larger, but undefined. This factor is used by cost functions NI() and V3(). See [BERT91] paragraph 3.2.

3.4.13 Target Class (targetType)

The meaning of Target Class in [BERT91] is not clear. It would appear that this would be type C(n,1), ie the class at the end of the query path, but on paragraph 3.2 of the paper (category 1) it is clear that the target class can be C(1,1) with the path length greater than 1 (k=2 where 1<k<=n in NI(k) for Execution Strategy SDFT). I first assumed that the Target Class (named targetType in PathString) is C(n,1), however I believe that the definition is in paragraph 2.2 of [BERT91] implies that the target is the type at the *start* of the path. If the attribute specified is an attribute of the base type (or the most specialized type in this specific hierarchy) then targetType is the type C(1). If the attribute is not an attribute of the base type, but of one of the sub types in the hierarchy, then the targetType is "IN" type C(1). This is the same as saying that the targetType is a member of C(1).

4.0 Experiments

4.0.1 Location of Rep/ directory.

The full path name of directory `Rep/` is `/pro/oodb/opt/Rep/`.

4.1 Sample Database: The Altair Travel Agency.

4.1.1 Source of the Schema.

The database schema for the Altair Travel Agency is a sample O2 database schema. The data for this database schema is in the directory `Rep/altair`. The program `OptRepTable` in the directory `Rep/RepLib` (formerly `table` in directory `Rep/rep_lib`) has been amended to be able to import a text file of this type, and the file `.text_dict` has been read into the data dictionary files `.type_dict`, `.data_dict` and `.attr_dict` in this directory.

4.1.2 Source of Cost Parameter values.

The parameter values used to calculate costs are in the disk file `.OptRepTypeCostData`. This file is intended to simulate statistical information obtained from a real database. This file must be in the default directory.

Information is read from this file when the first instance of the class `SchemaMgr` is declared. This happens when an EREQ query tree Rep is built, and the "path" annotations are added. This class data is stored in a Map structure named `typeMap`, which is a static field of the class `SchemaMgr`, the superclass of both `TypeData` and `PathString`.

The values of these parameters is invented, and because of the constraints between the parameters that need to be maintained it is difficult to maintain compatibility between parameters when varying one parameter at a time. See 3.4.8 Constraints on Cost Parameters on page 12.

Figure 4: Cost Parameters

These parameters are described in [BERT92] paragraph 3.1. They refer to a PATH composed of a variable and n attributes, each one an attribute of the previous type in the path. The type of each of the objects in the path is shown below as C(i,1). In our implementation C(i,1) refers to an instance of the class TypeData.

Object . Attribute(1) . Attribute(2) . . . . . Attribute(n-1) . Attribute(n)

C(1,1) . C(2,1) . C(3,1) . . . . . C(n)

Many of the parameters described in [BERT92] are derived from a few basic parameters; in addition some assumptions (given below) reduce the required cost parameters per type to 5 plus one for each multivalued attribute of the type.

All the parameters are integers.

i indexes into the path length. 1<=i<=n.

j indexes into the members of the type hierarchy based at C(i,1). 1<=j<=nc(i).

Note that parameters D(i,j), fan(i,j), d(i,j) and k(i,j) depend on the *type and an attribute* of the type, parameters N(i,j) and P(i,j) depend only on the *type*.

LOGICAL Data Parameters (data)

D(i,j) Number of distinct values for attribute A(i) of type C(i,j), 1<=i<=n, 1<=j<=nc(i).

N(i,j) Cardinality of type C(i,j), 1<=i<=n, 1<=j<=nc(i).

fan(i,j) Average number of references to members of type C(i+1,j), contained in the attribute A(i) for an instance of type C(i,j), 1<=i<=n, 1<=j<=nc(i). Note that for single valued attributes, fan=1.

d(i,j) Average number of instances of type C(i,j), having a value different from NULL for attribute A(i), 1<=i<=n and 1<=j<=nc(i).

PHYSICAL Data Parameters (data)

P(i,j) Number of pages containing instances of the type C(i,j) for 1<=i<=n, 1<=j<=nc(i) .

r(i) A binary variable assuming value equal to 1 if members of type C(i,j) have reverse references to members of type C(i-1,1) in the path, equal to 0 otherwise for 2<=i<=n. This value obviously varies depending on type C(i-1).

The following parameters are all derived from the above parameters and the particular path being considered, either by summing over the sub-classes of the type C(i,1), or by other means.

LOGICAL Data Parameters (derived from Path)

n Path length.

nc(i) Number of classes in the inheritance hierarchy rooted at type C(i,1), 1<=i<=n.

Dh(i) Number of distinct values for attribute A(i) for all instances in the inheritance hierarchy rooted at type C(i,1).

Nh(i) Number of members of type C(i,1), 1<=i<=n.

Figure 4: Cost Parameters (continued)

fanh(i) Average number of references to members of type C(i+1,1) contained in the attribute A(i) of a member of type C(i,1), 1<=i<=n. The difference between this parameter and fan(i,j) is that this parameter is obtained as the average evaluated on all members of a type hierarchy, while in fan(i,j) the average is for each class.

dh(i) Average number of members of type C(i,1), having a value different from NULL for attribute A(i), 1<=i<=n; dh(i) = SUM(1<=j<=nc(i))(d(i,j)).

k(i,j) Average number of instances of type C(i,j) having the same value for attribute A(i), 1<=i<=n and 1<=j<=nc(i); k(i,j)=CEILING((d(i,j)*fan(i,j))/D(i,j)).

kh(i) Average number of members of type C(i,1) having the same value for attribute A(i), 1<=i<=n. kh(i) = SUM[j] k(i,j).

PHYSICAL data parameters (based on Path)

Ph(i) Number of pages containing members of the type C(i,1) for 1<=i<=n. Assumed to be SUM[j] P(i,j); theoretically could be smaller than the sum, if there is some clustering.

QUERY parameters (based on Path)

NI(i) Number of members of type C(i,1) to be searched for, 1<=i<=n.. Depends on previous parameters.

AP(i) Number of accessed pages containing members of the type C(i,1) for 1<=i<=n.

DERIVED parameters. See appendix A of [BERT92].

RefBy(i,s,y,k) Average number of values contained in the nested attribute A(y) for a set of k instances of type C(i) whose position is s in the inheritance hierarchy. 1<=i<=y<=n, 1<=s<=nc(i), 1<=k<=Nh(i).

RefByh(i,y,k) Average number of values contained in the nested attribute A(y) for a set of k members of type C(i,1), 1<=i<=y<=n, 1<=k<=Nh(i).

k_bar(i,j)
Average number of instances of type C(i,j) having the same value for the nested attribute A(n), 1<=i<=n, 1<=j<=n.

kh_bar(i)
Average number of members of type C(i,1) having the same value for the nested attribute A(n), 1<=i<=n.

Ref(i,s,y,k)
Average number of instances of type C(i,s) having as value of the nested attribute A(y) a value in a set of k elements for 1<=i<=y<=n, 1<=s<=nc(i), 1<=k<=Nh(i). Also in this case, as for RefBy(i,s,y,k), s determines the position of the type in the inheritance hierarchy supposing that the classes are sorted in the hierarchy.

Refh(i,y,k)
Average number of members of type C(i,1), having as value of the nested attribute A(y) a value in a set of k elements for 1<=i<=y<=n, 1<=k<=Nh(i).

4.2 A driver program to create the EREQ rep.

4.2.1 Instructions

The files are located in the directory

`/pro/oodb/opt/Rep/.`

The executable which drives the current programs is named **`makeRep`**. This takes 2 arguments. First a file name which should contain a query in the correct format, and second an integer which selects certain outputs.

Use `1` to just output the query paths and the cost of each query path.

Use `2` to output the query tree in the EREQ Rep format. This produces quite a lengthy output, but the fields can be inspected this way.

Use `3` to output the Class Map which is used when calculating the cost.

Use 4 to create a qTreeType tree from the EREQ query tree and display it using TREEprint_qTree().

Use `0` to output all 4 of the above.

For example, to print the costs for the query paths generated by the query contained in the file altair11.input, type:

```
> makeRep altair11.input 1
```

alternatively, use makeRep from the altair subdirectory , type:

```
> ../makeRep altair11.input 1
```

4.2.2 Input Queries.

The queries used for testing were taken from the file **`test-queries`**. In order to get these into the form required by the "opt" parser, I wrote a translator in emacs lisp. This function is named **`o2-equal-conv`** and is in the file **`o2-equal-conv.el`**. A listing of the latter file is in Appendix C.

4.2.3 Typical Output.

The graphical output from **`opt`**, produced using the function **`TREEprint_qTree()`** is shown on page 9 The output from the EREQ Rep has not yet been output in graphical form. A sample of the output from the EREQ rep, indicating the fields of the Nodes, is included on page 9.

There is a paradoxical result in the "cost" annotation that the cost is zero when the strategy is Reverse Traversal and reverse references are provided. See note in [BERT92] section 3.2 after the section on Nested Loop Reverse Traversal. This is because in those cases the path length is 1 and in this case, the reverse pointers, that are given as OID's form the solution of the query.

4.3 Description of Experiments

Experiments have been carried out to see hoe the cost varies as the input parameters are changed. To do this I have built a PathString from scratch, not via the Query Rep, and varied one parameter at a time.

The driver used for these experiments is named **`costExp`**, and is in the directory **`Rep/work`**..

4.3.1 Use of the experiment driver programs, costExp and costTest.

These programs are in the directory **`Rep/work/`**.

The driver program **`costExp`** will produce calculate the results of varying one parameter of N, fan, D, d, or P. All 5 can be done in one run in series, and each execution strategy can be selected. In addition,

both values for the parameter r can be displayed. It is assumed that the user of the program will be able to amend the source code file **costExp.C** if more experiments are to be carried out.

In order to trace the reasons for the relationships described below, I compiled the cost functions with the variable TESTING defined. This is done in the makefile by giving a flag -DTESTING. This causes additional output giving the values of individual functions. The library functions have been compiled with TESTING not set.

The driver **costExp** produces several temporary files, suffixed **.tmp**, and two results data files. The first (named "**plot*nnnn*.xy**") is used to plot the points using xgraph, and the second (named "**plot*nnnn*.txt**") which prints out the data points in a more readable format..

In addition there is a second test file, named costTest, which can be used to test individual PathString functions. At present only H() works, but it is easy to add a function to this program and then test a function by calling **costTest <function name> <arg 1> <arg 2>** etc.

4.3.2 Graphical output.

The above programs call the program xgraph to display their results graphically. These have been included in Appendix D.

The output from the experiments is shown in Appendix D: Experimental Results on page 57.

4.3.3 Method of including graphical results.

To import the plots into this document, the xgraph file was written out to an idraw file. The idraw file was then imported to idraw and written out as an EPSI file (the "save as" option). Since an encapsulated postscript file can be imported into framemaker, this is then imported into framemaker as a graphic file.

To dump the xgraph direct via xwd or the frame capture option would have printed the data lines as colored lines, which are fairly unreadable in monochrome.

4.4 Experiments 1.1.1 to 1.1.5.

4.4.1 Introduction

This is based on the Path "City.places_to_go.name". I have varied each parameter of the type "City" in turn, keeping all others constant. The plots of these are shown as Experiment 1.1.1 to Experiment 1.1.5.

The constant parameters of the PathString are as follows:

City.places_to_go.name

		Parameters given:					
Object:	Class:	Dij	Nij	fanij	dij	Pij	ri
City	:City	100	100	10	80	100	0
Attribute:							
places_to_go	:Place_tg	200	200	1	71	200	0
	:Monument	1	10	1	1	5	
	:Museum	1	10	1	1	10	
	:Theater	1	12	1	1	12	
name	:String	1	10000	1	1	10000	0

		Parameters calculated:						
Object:	Class:	kij	Di	Nh	fani	di	kh	Ph
City	:City	8	100	100	10	80	8	100
Attribute:								
places_to_go	:Place_tg	1	203	232	1	74	1	227
	:Monument	1						
	:Museum	1						
	:Theater	1						
name	:String	1	1	10000	1	1	1	10000

Note that some of these parameters seem a bit dumb. For example, what does it mean that a primitive type such as "string" has a cardinality N ? This should be an exception, but it is not done this way yet.

4.4.2 Results

The first result from 1.1.1.and 1.1.2 shows that NLFT depends directly on Cardinality "N", and the set size of attribute "fan", and not on the other parameters. The functions for NLFT are just this: (see [BERT92] para 3.2 Category 1). The cost is N(1,1) * fan(1,1), then progressively cost = cost * fan(i,1) along the path. If all the parameters (fan==1) then the cost will be the size of N(1,1) only.

Experiments 1.1.3 and 1.1.4 shows that nothing depends on parameter D or d. This seems surprising and may be due to this being the first attribute in the path.

Experiment 1.1.5 shows the effect of P, number of Pages containing instances of the type. As the number of pages increases, the cost increases.

4.5 Experiments 2.1.1 to 2.1.5.

4.5.1 Introduction

The second set of experiments is based on a similar Path to the first. In this case the parameters of the second type C(2,1) are varied.

```
City.places_to_go.address.street
                                    Parameters given:
Object:          Class:             Dij    Nij     fanij    dij     Pij     ri
City             :City              100    100      10      80      100      0
Attribute:
places_to_go     :Place_tg           61    200       1      71      100      0
                 :Monument            1     10       1       1        5
                 :Museum              1     10       1       1       10
                 :Theater             1     12       1       1       12

address          :Address         12671   155       1     134      155      0

street           :String              1   10000      1       1    10000      0


Object:          Class:           Parameters calculated:
City             :City              kij      Di       Nh      fani     di       kh       Ph
Attribute:                           8      100      100       10      80        8      100
places_to_go     :Place_tg
                 :Monument           2       64      232        1      74        2      127
                 :Museum             1
                 :Theater            1
                                     1
address          :Address
                                     1    12671     155        1     134        1      155
street           :String
   1       1      10000      1        1        1      10000
```

4.5.2 Results

• Experiment 2.1. The top graph (NL) shows that N(2,1) does not alter the cost. This is as expected; see note in 4.4.2 above. The particular parameters selected for this case produced an error message, that N(2,1) is smaller than D(1,1) and P(2,1). The bottom graph (SD), shows that there is an inverse relationship between cost and N, for SDFT, values of N>=73. The reason for the out-of-range values for cost is that the parameter Ph>Nh. This causes the function of Yao (PathString::H()) to give a stupid answer. The program gives error messages if it detects such a conflict. The reason for the inverse relationship also lies with the function H(); this is because, as N(2) increases, with P constant, the concentration of records per page is increasing. This increases the hit rate. The number of records to be searched for is constant, since

```
Creating new rep for:

Project(Places,$p
{(Place,p),(
 Tours,Select(Tour,$t
p memberof Image(t@description,$w w@what@address@city@ti-
tle)))))


----------------------------
PRINTING DETAILS OF NEW REP.
----------------------------

DataNode: Ptr=0x419e8.
X=0, Y=0,...ID [0]

Annotations:
Key="avail"     Ptr=0x0
Key="cost"      Ptr=0x6b010
            NLFT cost = 1085400
            NLRT cost = 4461480
            SDFT cost = 123120
            SDRT cost = 145216800
Key="path"      Ptr=0x6a5b8
            Places
            p
            Tour
            p
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x0

ParentFD->ParentFN = 0x0->0x0
ChildDF->ChildFN = 0x49b58->0x49c38
Data =     0x0
DataType = "Set[Tuple:(Place,Place_tg),(Tours,Set[Tour])]"
Name =     ""
 Multivalued, size 300
-------------------------------------------------

FunctionNode: Ptr=0x49c38.
X=0, Y=2,...ID [00]
Operator name: "Project"
Arity=3, Input args=2, Other args=1

Annotations:
Key="avail"     Ptr=0x0
Key="cost"      Ptr=0x69848
            NLFT cost = 1085400
            NLRT cost = 4461480
            SDFT cost = 123120
            SDRT cost = 145216800
Key="def"       Ptr=0x45710
                                p : Place_tg
Key="path"      Ptr=0x5e8f8
            Places
            p
            Tour
            p
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x0

ParentDF()->ParentDN() = 0x49b58->0x495e0
-------------------------------------------------
FunctionNode 0x49c38: INPUT # 0:
NthInputFD()->ChildDN() = 0x430c8->0x431d0

CInputDataNode: Ptr=0x431d0.
X=0, Y=4,...ID [0000]

Annotations:
Key="avail"     Ptr=0x0
Key="cost"      Ptr=0x5e730
            NLFT cost = 0
            NLRT cost = 0
            SDFT cost = 0
            SDRT cost = 0
Key="path"      Ptr=0x5e6f0
            Places
Key="used"      Ptr=0x0

ParentFD->ParentFN = 0x430c8->0x49c38
ChildDF->ChildFN = 0x0->0x0
Data =     0x0
DataType = "Set[Place_tg]"
Name =     "Places"
 Multivalued Global variable, size 300

LEAF NODE.
-------------------------------------------------
FunctionNode 0x49c38: INPUT # 1:
NthInputFD()->ChildDN() = 0x433a0->0x434a8

VInputDataNode: Ptr=0x434a8.
X=1, Y=4,...ID [0001]

Annotations:
Key="avail"     Ptr=0x477b0
                                p : Place_tg
Key="cost"      Ptr=0x5eda0
            NLFT cost = 0
            NLRT cost = 0
            SDFT cost = 0
            SDRT cost = 0
Key="path"      Ptr=0x5ed60
            p
Key="used"      Ptr=0x47890
                                p :

ParentFD->ParentFN = 0x433a0->0x49c38
ChildDF->ChildFN = 0x0->0x0
Data =     0x0
DataType = "Place_tg"
Name =     "p"
 Local variable

LEAF NODE.
-------------------------------------------------
FunctionNode 0x49c38: OTHER ARGUMENT # 0:
NthOtherFD()->ChildDN() = 0x43678->0x43780

OtherDataNode: Ptr=0x43780.
X=2, Y=4,...ID [0002]

Annotations:
Key="avail"     Ptr=0x47820
                                p : Place_tg
Key="cost"      Ptr=0x68750
            NLFT cost = 1085400
            NLRT cost = 4461480
            SDFT cost = 123120
            SDRT cost = 145216800
Key="path"      Ptr=0x68078
            Tour
            p
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x49570
                                p :

ParentFD->ParentFN = 0x43678->0x49c38
ChildDF->ChildFN = 0x43950->0x43a58
Data =     0x0
DataType = "Set[Tour]"
Name =     ""
 Multivalued, size 36
-----------------------------------------

FunctionNode: Ptr=0x43a58.
X=0, Y=6,...ID [000200]
Operator name: "Select"
Arity=2, Input args=1, Other args=1

Annotations:
Key="avail"     Ptr=0x47900
                                p : Place_tg
Key="cost"      Ptr=0x67688
            NLFT cost = 1085400
            NLRT cost = 4461480
            SDFT cost = 123120
            SDRT cost = 145216800
Key="def"       Ptr=0x45b00
                                t : Tour
Key="path"      Ptr=0x5f548
            Tour
            p
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x49500
                                p :

ParentDF()->ParentDN() = 0x43950->0x43780
-------------------------------------------------
FunctionNode 0x43a58: INPUT # 0:
NthInputFD()->ChildDN() = 0x43be0->0x43ce8

CInputDataNode: Ptr=0x43ce8.
X=0, Y=8,...ID [00020000]

Annotations:
Key="avail"     Ptr=0x47970
                                p : Place_tg
Key="cost"      Ptr=0x5f358
            NLFT cost = 0
            NLRT cost = 0
            SDFT cost = 0
            SDRT cost = 0
Key="path"      Ptr=0x5f1a8
            Tour
```

Figure 5: The EREQ Query Tree in text format.

Query text:

Project(Places,$p{(Place,p),

(Tours,Select(Tour,$tp memberof Image(t@description,$w w@what@address@city@title))))})

```
Key="used"      Ptr=0x0

ParentFD->ParentFN = 0x43be0->0x43a58
ChildDF->ChildFN = 0x0->0x0
Data =     0x0
DataType = "Set[Tour]"
Name =     "Tour"
 Multivalued Global variable, size 120

LEAF NODE.
-------------------------------------------------
FunctionNode 0x43a58: OTHER ARGUMENT # 0:
NthOtherFD()->ChildDN() = 0x43eb8->0x43fc0

OtherDataNode: Ptr=0x43fc0.
X=1, Y=8,...ID [00020001]

Annotations:
Key="avail"     Ptr=0x479e0
                            p : Place_tg
                            t : Tour
Key="cost"      Ptr=0x66848
            NLFT cost = 9045
            NLRT cost = 37179
            SDFT cost = 1026
            SDRT cost = 1210140
Key="path"      Ptr=0x662c8
            p
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x49340
                            p :
                            t :

ParentFD->ParentFN = 0x43eb8->0x43a58
ChildDF->ChildFN = 0x44190->0x44298
Data =     0x0
DataType = "Boolean"
Name =     ""

-------------------------------------------------

FunctionNode: Ptr=0x44298.
X=0, Y=10,...ID [0002000100]
Operator name: "memberof"
Arity=2, Input args=1, Other args=1

Annotations:
Key="avail"     Ptr=0x47ac0
                            p : Place_tg
                            t : Tour
Key="cost"      Ptr=0x65628
            NLFT cost = 9045
            NLRT cost = 37179
            SDFT cost = 1026
            SDRT cost = 1210140
Key="def"       Ptr=0x0
Key="path"      Ptr=0x5fa88
            p
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x493b0
                            p :
                            t :

ParentDF()->ParentDN() = 0x44190->0x43fc0
-------------------------------------------------
FunctionNode 0x44298: INPUT # 0:
NthInputFD()->ChildDN() = 0x44420->0x44528

VInputDataNode: Ptr=0x44528.
X=0, Y=12,...ID [000200010000]

Annotations:
Key="avail"     Ptr=0x47ba0
                            p : Place_tg
                            t : Tour
Key="cost"      Ptr=0x5f8c0
            NLFT cost = 0
            NLRT cost = 0
            SDFT cost = 0
            SDRT cost = 0
Key="path"      Ptr=0x5f880
            p
Key="used"      Ptr=0x47d60
                            p :

ParentFD->ParentFN = 0x44420->0x44298
ChildDF->ChildFN = 0x0->0x0
Data =     0x0
DataType = "Place_tg"
Name =     "p"
 Local variable

LEAF NODE.
-------------------------------------------------
FunctionNode 0x44298: OTHER ARGUMENT # 0:
NthOtherFD()->ChildDN() = 0x446f8->0x44800

OtherDataNode: Ptr=0x44800.
X=1, Y=12,...ID [000200010001]

Annotations:
Key="avail"     Ptr=0x47c80
                            p : Place_tg
                            t : Tour
Key="cost"      Ptr=0x64b68
            NLFT cost = 335
            NLRT cost = 1377
            SDFT cost = 38
            SDRT cost = 44820
Key="path"      Ptr=0x647a8
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x49260
                            t :

ParentFD->ParentFN = 0x446f8->0x44298
ChildDF->ChildFN = 0x449d0->0x44ad8
Data =     0x0
DataType = "Set[String]"
Name =     ""
 Multivalued, size 27
-------------------------------------------------

FunctionNode: Ptr=0x44ad8.
X=0, Y=14,...ID [00020001000100]
Operator name: "Image"
Arity=2, Input args=0, Other args=2

Annotations:
Key="avail"     Ptr=0x47dd0
                            p : Place_tg
                            t : Tour
Key="cost"      Ptr=0x640d0
            NLFT cost = 335
            NLRT cost = 1377
            SDFT cost = 38
            SDRT cost = 44820
Key="def"       Ptr=0x46120
                            w : Stage
Key="path"      Ptr=0x60bf0
            t.description
            w.what.address.city.title
Key="used"      Ptr=0x492d0
                            t :

ParentDF()->ParentDN() = 0x449d0->0x44800
-------------------------------------------------
FunctionNode 0x44ad8: OTHER ARGUMENT # 0:
NthOtherFD()->ChildDN() = 0x44c60->0x45160

OtherDataNode: Ptr=0x45160.
X=0, Y=16,...ID [0002000100010000]

Annotations:
Key="avail"     Ptr=0x47eb0
                            p : Place_tg
                            t : Tour
Key="cost"      Ptr=0x609a0
            NLFT cost = 11
            NLRT cost = 0
            SDFT cost = 11
            SDRT cost = 0
Key="path"      Ptr=0x60960
            t.description
Key="used"      Ptr=0x48380
                            t :

ParentFD->ParentFN = 0x44c60->0x44ad8
ChildDF->ChildFN = 0x45330->0x45438
Data =     0x0
DataType = "Set[Stage]"
Name =     ""
 Multivalued, size 27
-------------------------------------------------

FunctionNode: Ptr=0x45438.
X=0, Y=18,...ID [000200010001000000]
Operator name: "description"    ATTRIBUTE
Arity=1, Input args=1, Other args=0

Annotations:
Key="avail"     Ptr=0x480e0
                            p : Place_tg
                            t : Tour
```

etc.

This is about half of the query tree.
The bottom leaf is at Y=32.

Figure 5: The EREQ query tree in text format (continued).

This output is produced by the functions DataNode::printNode() and FunctionNode::printNode(). A call to DataNode::printNode() calls the printNode() function for its child, recursively.

this depends on N(1) and not N(2), therefore the cost reduces as N(2) increases. Note that increasing N(2) without increasing P(2) is not a realistic relationship.

SDRT with and without reverse references are constant, although not the same value.

- Experiment 2.2. Since this attribute is not multivalued, there is no experiment .2 in this series.
- Experiment 2.3 and 2.4. Similar conflicts to 2.1 above have given cause to the steps in the graphs of NLRT for varying D (graph 2.3), and NLRT for varying d (graph 2.4). From graph 2.3 we see that for SDFT, cost is proportional to D, the number of distinct values for attribute A(2).
- Experiment 2.5 shows a linear relationship between P and cost for SDRT when r=0, (SDRT and r=1 is flat. There is a more complex relationship for cost when SDFT is considered. This is the effect of the function of Yao, which calculates the "hit rate" when searching for records from the disk.

5.0 Conclusions

5.1 Query Rep

The new query rep provides all the functionality of the old, plus more facilities, with one exception, and that is the fact that the annotations are not typed.. There are some unsolved problems with the Annotations, due to the fact that they are not typed, and therefore cannot be copied, deleted, or checked for equality in a general way.

5.2 Schema Manager

The Schema Manager has been written to enable access to type details, and to include type cost parameters. It does these successfully, although the functionality which has been added in comparison to the previous schema manager is only in relation to the cost parameters.

Schema Manager class provides a framework for adding further properties of types in the future.

5.3 Cost Model

5.3.1 Interpretation

There have been some problems in interpreting Bertino's model. One of these is the interpretation of a "class hierarchy". The idea is to assume that each hierarchy will be separate until a TOP_CLASS is reached. It seems that there will be a schema where the classes are related, and so the hierarchies will not be so distinct. I have assumed in the implementation that the named class C(i,j) is the root of a hierachy, but this may not be the intention of [BERT92].

Overall this cost model may provide the best model in the absence of a specific database model to base a cost model on.

5.3.2 Parameter difficulties

The [BERT92] cost model has been shown to have some problems. One of the worst of these is the difficulty of inventing values for the cost parameters in the absence of "real" data to work on. The problem is that it is almost impossible to keep all parameters constant and vary just one to see what effect this has

on the cost. This has particularly inhibited the testing of the model, perhaps it is a function of the models comprehensiveness.

One possibility would be to generate random values for the cost parameters, constrained as indicated in this paper, and use these to generate costs. If this were done many times, perhaps some trends could be seen.

6.0 Further work

6.1 Query Rep

There are functions not yet implemented, such as copy constructors, and operators such as equality and assignment. Should these be recursive?

The graphical output is at present only possible by converting the EREQ rep back to the qTree rep (using DataNode::createQueryTree()) and printing qTreeType. If the EREQ rep could be displayed graphically, it would then be possible to add selected fields from the rep to display; fields which are not present in the old rep.

6.2 Schema Manager

The next step would be to ensure that no functions from OptRepTable.c are called by the Rep classes or RepCost classes, except via the class SchemaMgr. This would provide an easy way to substitute the new schema for the old one at some future date, when the optimizer can access the new one.

6.3 Cost Model

The next step to incorporate this model into the EREQ query tree would be to use the cost model functions to calculate the costs of paths that include a FunctionNode of the type Image or Select. This would require modification of the function FunctionNode::BuildQueryPaths().

6.4 General

6.4.1 Location of files.

The files for this work are in the following directories.

Files for public use are in directory **/pro/oodb/opt/Rep/**. This directory should be used as the "include" and "library" directory in a makefile. This directory includes:

- header files **OptRepMeta.H**. **OptRep.H**, **OptRepCost.H**, **.OptRepLib.H**.
- the OptRep library **libOptRep.a**;
- the working driver program **makeRep**; and
- the data dictionary files **.data_dict**, **.type_dict** **.attr_dict** and **.text_dict**.
- The cost data file **.OptRepTypeCostData**. All these data files are for the Altair Travel Agency database.

All compilation is done in the directory **/pro/oodb/opt/Rep/work/.**

This includes

- .C files **OptRepMeta.C**, **OptRep.C**, **OptRepCost.C**. **makeRep.C**, **costExp.C**, **costTest.C**.
- **Makefile** for compilations of **makeRep**, **libOptRep.a**, **costExp**, **costTest.**

This directory also includes test output, executables of the driver files and copies of certain altair input files.

Files relating to the "gcl" rep are in directory **/pro/oodb/opt/Rep/RepLib/**. This contains the older version of the library, named **libRep.a**, which is now included in **libOptRep.a**. The header file **OptRepIntRep.h**, which contains enums for the operators etc. is in this directory, and is included in the .H files.

6.4.2 Makefile

The Makefile used is in directory **pro/oodb/opt/Rep/work/**. Compile options with this Makefile are:

With no target - compiles makeRep.

make exp - compiles costExp.

make lib - compiles the library libOptRep.a

make cst - compiles costTest.

The -DTESTING flag will cause many print statements to be activated. The library option should not be compiled with this flag set.

Appendix A: Class Definitions

The header files are summarized below.

Header file: OptRepMeta.H

---

class OptRepMetaClass

```
class OptRepMetaClass {

private:
  char fOptRepClassName[64];

public:
  char *OptRepClassName() { return fOptRepClassName; }
  void OptRepClassName(char *newName);

};
```

class vlist

```
/*
 * Generic linked list type to use with annotations.
 * Any data pointer can then be listed.
 *
 */
class vlist {
public:
  void *data;
  struct vlist *next;
};
```

class AnnotList

```
/*
 * Generic linked list type to use with annotations.
 * This has rather more functionality than vlist.
 *
 */
class AnnotList {
private:
  void *fData;
  int  fDataSize;  // to allow arbitrary copying of the AnnotList
  struct AnnotList *fNext;
  char *fAnnotName; // the annotation key.

public:
  void  *Data()  { return fData; }
  void  *getNthVar(int);
  int   getListSize();            // number of elements
  AnnotList *Next() { return fNext; }
  char  *Name() { return fAnnotName; }

  AnnotList();
  AnnotList(int);  // arg is size of the data
  AnnotList(AnnotList*,int);      // 2nd arg is size of the data
  ~AnnotList();

};
```

Header file: OptRep.H

Op

```
class Op  : public OptRepMetaClass {

  private:
    char fName[OP_SYM_LEN];
    int  fArity;
    int  fNumInputArguments;

  public:
    // constructor
    Op();
    Op(char*, int ,int);
    Op(Op&);

    // destructor
    ~Op();

  public:
    char *Name() { return fName; }
    operType OperType();
    int  Arity() { return(fArity); }
    int  NumInputArguments() { return( fNumInputArguments ); }
    int  NumOtherArguments() { return( fArity - fNumInputArguments ); }

    int  isAttribute();

    void PrintOp();

}; // class Op
```

class ArcNode

```
class ArcNode : public OptRepMetaClass { // Abstract Class

private:
  // fAnnotations should be a pointer to a property list
  Map(USLString,voidPtr) fAnnotations;

protected:
  // Keep some identification for this object:
  static int perGen[64]; // keep track of number per generation
  int   fDimX;           // number in generation
  int   fDimY;           // generation.
  char  fIDStr[64];      // record of ancestry, one char per generation
  int   fSiblingNum;     // which number sibling is this ?

public:
  ArcNode(qTreeType*, ArcNode*); // constructor from "George" rep
  ArcNode(ArcNode&);             // copy constructor

  ~ArcNode();

public:
  void  *GetAnnotation(USLString index);
  void  SetAnnotation(USLString index, void *value);
  Map(USLString,voidPtr)  Annotations() { return fAnnotations; }

  int numAnnotations();

  // Functions for the "path" annotations
  void     SetPathAnnot(PathString *qPath);  // put this path only in annot
  void     AddPathAnnots(vlist*);            // Add an annot to this one
  void     CopyPathAnnot(ArcNode*);  // copy another path annot to *this
  vlist*   CopyPathVList(vlist*);
  void     freePathVList(vlist*);

  // function to be used with PrintNode or PrintArc
  void PrintAnnotations();

public:
  char *IDStr()     { return fIDStr; }
  void setIDStr(ArcNode*);
  int  dimX()       { return fDimX;  }
  int  dimY()       { return fDimY;  }
  void setDimX(int value) { fDimX = value; }
  int  SiblingNum() { return fSiblingNum;  }
};
```

class Node

```
class Node : public ArcNode { // Abstract Class

public:
  Node(qTreeType*, Arc*);       // constructor from "George" rep
  Node(Node&);                  // copy constructor
  ~Node();

public:
  virtual Arc  *Parent(int) = 0;

public:
  qTreeType *CreateQueryTree(qTreeType*);

};
```

class FunctionNode

```
class FunctionNode : public Node {

private:
  Arc *fParent[MAX_DIM];

  Op   *fOper;
  int  fNumInputs;  // perhaps these should be covered by Op ??
  int  fNumOthers;

  // see Op. We have Arity = Input arguments + Other arguments
  FDArc *fInputs[MAX_ARITY];  // input arguments
  FDArc *fOthers[MAX_ARITY];  // non input arguments


public:
  // constructor
  FunctionNode();
  // constructor from "George" rep
  FunctionNode(qTreeType *gTree, DFArc *parentDF);
  FunctionNode(FunctionNode&);  // copy constructor


  // destructor
  ~FunctionNode();

public:
  Arc  *Parent(int dim=0) { return fParent[dim]; }
  FDArc *Child(int i,int dim=0);   // returns NthInput(i) up to (numInputs()-1)
                                   // then returns NthOther(i-numInputs())
  DataNode *GChild(int i,int dim=0);

  // normally the dimension is 0 (for default)
  FDArc  *NthInput(int n, int dim = 0);
  FDArc  *NthOther(int n, int dim = 0);
  int    numInputs() { return fNumInputs; }
  int    numOthers() { return fNumOthers; }
  void   IncInputs(FDArc *newFD);
  void   IncOthers(FDArc *newFD);


  DataNode *ReplaceNthInput(int n, InputDataNode *theNode, int dim = 0);
  DataNode *ReplaceNthOther(int n, OtherDataNode *theNode, int dim = 0);

  Op   *Oper() { return fOper; } // because operator is a reserved word in C++
  int  ChangeOper(Op);

  Boolean  buildQueryPaths(Boolean);

  // a function to print the contents of the node
  void PrintNode();
  void addIDStr();

public:
  qTreeType *CreateQueryTree(qTreeType*);

};
```

class DataNode

```
class DataNode : public Node { // abstract class

private:
  Arc   *fParent[MAX_DIM];
  DFArc *fChild[MAX_DIM];    // ?? or MAX_ARITY
  void  *fData;
  char  *fDataType;
  char  *fName;
  Boolean  fIsMultivalued; // Does the data represent a set (multivalued)
  int      fNumValues;     // Size of the set. This is less confusing than
                           // using the set word.

public:
  // constructor
  DataNode();                  // constructor from "George" rep
  // Takes an EQUAL query as arg. 2nd arg is flag to build Paths.
  DataNode(USLString, Boolean buildQP=FALSE);

  // Takes a gcl query tree as arg. 2nd arg is flag to build Paths.
  DataNode(qTreeType*, Boolean buildQP=FALSE);

    DataNode(qTreeType *, FDArc *); // recursive constructor from "gcl" rep
//  DataNode(DataNode&);   // copy constructor - must decide if this
                           // should copy a node or a tree.

  DataNode(qTreeType *, FDArc *);

  // destructor
  ~DataNode();

public:
  Arc  *Parent(int dim = 0)  { return fParent[dim]; }  // virtual in Node
  DFArc  *Child(int dim = 0) { return fChild[dim]; }

  FunctionNode *GChild(int dim = 0);

  void *Data(int dim = 0) { if (dim); return fData; }
  void SetData(void *data, int dim = 0){ if (dim); fData = data; }

  char  *DataType()  { return fDataType; }
  void  SetDataType(char *dataType) { fDataType = dataType; }
  char  *Name()      { return fName; }
  void  SetName(char *name)  { fName = name; }
  void  Splice(DataNode *theNewNode);  // splice the new Node in for this node

  Boolean IsMultivalued() { return fIsMultivalued; }
  int     NumValues() { return fNumValues; }    // size of set if multivalued

//
// question: this splice replaces one node with another -- how do we
// create shared structures using splice?  Or do we only allow that at
// rep creation time?

  Boolean  isLeafNode();
  Boolean  IsGlobalVar();  // Does this node represent a global variable
  Boolean  IsLocalVar();   // Does this node represent a local variable

  Boolean  buildQueryPaths(Boolean);

  // a function to print the contents of the node
  void PrintNode();
  void addIDStr();

private:
  void copyFields(DataNode*);

public:
  qTreeType *CreateQueryTree(qTreeType* QTN=NULL);

};
```

class InputDataNode

```
class InputDataNode : public DataNode {       // abstract class

public:
  // constructor from "George" rep
  InputDataNode(qTreeType*, FunctionNode*, FDArc*);
  InputDataNode(InputDataNode&);   // copy constructor

  ~InputDataNode();
};
```

class VInputDataNode

```
class VInputDataNode : public InputDataNode { // leaf variable node

public:
  // constructor from "George" rep
  VInputDataNode(qTreeType*, FunctionNode*, FDArc*);
  VInputDataNode(VInputDataNode&);   // copy constructor

  ~VInputDataNode();
};
```

class CInputDataNode

```
class CInputDataNode : public InputDataNode { // leaf constant node

public:
  // constructor from "George" rep
  CInputDataNode(qTreeType*, FunctionNode*, FDArc*);
  CInputDataNode(CInputDataNode&);   // copy constructor

  ~CInputDataNode();
};
```

class OtherDataNode

```
class OtherDataNode : public DataNode {       // interior other node

public:
  // constructor from "George" rep
  OtherDataNode(qTreeType*, FunctionNode*, FDArc*);
  OtherDataNode(OtherDataNode&);   // copy constructor

  ~OtherDataNode();
};
```

class Arc

```
class Arc : public ArcNode { // Abstract Class

public:
  Arc(qTreeType*, Node*);        // constructor
  Arc(Arc&);                     // copy constructor
  ~Arc();

};
```

class FDArc

```
class FDArc : public Arc { // represents arc from function to data

private:
  FunctionNode *fParent;
  DataNode     *fChild;

public:
  // constructor
  FDArc();
  FDArc(qTreeType*);
  FDArc(qTreeType*, FunctionNode*);
  FDArc(FDArc&);

  ~FDArc();

public:
  FunctionNode *Parent(int dim = 0) { if (dim); return fParent; }
  DataNode     *Child(int dim = 0)  { if (dim); return fChild; }
  void        SetChild(DataNode *theNode, int dim = 0) {
      if (dim);
      fChild = theNode;
  }
  void  MakeChild(qTreeType*, FunctionNode*);

public:
  Boolean  buildQueryPaths(Boolean);

  void addIDStr();

};
```

class DFArc

```
class DFArc : public Arc { // represents arc from data to function

private:
    DataNode     *fParent;
    FunctionNode *fChild;

public:
  DFArc();
  DFArc(qTreeType *, DataNode *);
  DFArc(DFArc&);

  ~DFArc();

  DataNode     *Parent(int dim = 0) { if (dim); return fParent; }
  FunctionNode *Child(int dim = 0)  { if (dim); return fChild; }

public:
  Boolean  buildQueryPaths(Boolean);

  void addIDStr();

};
```

OptRepCost.H

class SchemaMgr

```
class SchemaMgr : public OptRepMetaClass {
  protected:
    // The Map will not work as a static, only as a static*
    // It is initialized by the first call to PathString constructor.
    static Map(USLString, TypeDataPtr)  *typeMap;
    static fCountSM;

  public:
    SchemaMgr();
    ~SchemaMgr();

  protected:
    int  buildTypeMap();
    // Build a Map(String, TypeDataPtr) so that data no any type can be
    // located from the name of the type. The data to build the Map will
    // come A. From the data dictionary - This gives superType, subType, and
    // attributes of the type.
    // and  B. From the file of Cost Data which will give values of the
    // parameters required for the cost functions

  public:
    void printTypeMap();

    TypeDataPtr getTypeData(char*); // Argument is a type name.
    TypeDataPtr getTypeDataErr(char*,char*); // 2nd arg is msg
    Boolean     TDAttrExistsErr(char*,char*,char*,TypeData**,int*);

    // Access functions required for experiments - parameters are varied
    // one by one, specifying the Type and Attribute of the type..
  public:
    void Setfan(char*,char*,int);
    int  Getfan(char*,char*);
    void SetD(char*,char*,int);
    int  GetD(char*,char*);
    void SetN(char*,int);
    int  GetN(char*);
    void Setd(char*,char*,int);
    int  Getd(char*,char*);
    void Setr(char*,char*,int);
    int  Getr(char*,char*);
    void SetP(char*,int);
    int  GetP(char*);

  protected:
    Boolean FirstErr(char *msg=NULL);  // Set or Read a first time flag.

}; // end class SchemaMgr
```

class TypeData

```
/*
 * The class Cij is designed to contain the data referred to in
 * the E.Bertino paper.  There is a mapping function using
 * Map(USLString, CijPtr) to retrieve to correct
 * instance of this class for a particular type name.
 *
 */

// Maximum spread in hierarchy, UP or DOWN
#define MAX_FAM 20
// The meta-type "Type" may have many children
#define MAX_FAM_TYPE  500

// Maximum number of attributes in a type.
const int MAX_NUM_ATTR = 100;


class TypeData : public SchemaMgr {

// allow access by functions in class PathString
friend class PathString;

  protected:
    // structural data
    TypeData *fParent[MAX_FAM];    // superType of this type
    TypeData *fChild[MAX_FAM];     // subTypes  of this type
    int fNumParents;
    int fNumChildren;

    // The attribute array is a struct, since there are several parameters
    // which are different for each attribute.
    int  fNumAttributes;
    struct {
       char  *Name;
       char  *Type;
       int   IsSet;
       int   D;     // Parameter 3
       int   fan;   // Parameter 4
       int   d;     // Parameter 5
       int   r;     // Parameter 6
    } fAttr[MAX_NUM_ATTR];

  private:
    // Logical Data Parameters

    char fTypeName[64];  // The Equal type name

    int  fnc;     // members of this type.

    int  fNij;    // Parameter 1

    // Physical data parameters

    int   fPij;  // Parameter 2

  public:
    TypeData(char*); // the argument is the Type Name
    ~TypeData();

  public:
    // structural data
    TypeData *Parent(int dim)    { return fParent[dim]; }
    TypeData *Child(int dim)     { return fChild[dim];  }
    int NumParents()        { return fNumParents;  }
    int NumChildren()       { return fNumChildren; }

    // The following functions refer to Attributes of the type
    // described by this instance of TypeData.
    int     NumAttributes()    { return fNumAttributes;  }

    Boolean AttrExists(char*);
    Boolean AttrExists(char*,int*);  // index returned as 2nd arg

    char    *AttrName(int i)   { return fAttr[i].Name;     }
    char    *AttrType(int i)   { return fAttr[i].Type;     } // T or Set[T]
    Boolean AttrIsSet(int i)   { return fAttr[i].IsSet;    }

    int     AttrD(int i)    { return fAttr[i].D;    }
```

```
    int     Attrfan(int i) { return fAttr[i].fan; } // =1 by default
    int     Attrd(int i)   { return fAttr[i].d;   }
    int     Attrr(int i)   { return fAttr[i].r;   }

    void    SetAttrD(int i, int value)   { fAttr[i].D = value;   }
    void    SetAttrfan(int i, int value) { fAttr[i].fan = value; }
    void    SetAttrd(int i, int value)   { fAttr[i].d = value;   }
    void    SetAttrr(int i, int value)   { fAttr[i].r = value;   }

  public:
    int nc()              { return fnc; }

    int  N()              { return fNij;  }    // Parameter 2
    void SetN(int value) { fNij = value; }

    int P()               { return fPij;  }    // Parameter 3
    void SetP(int value) { fPij = value; }

  private:
    // The following functions get type and attribute information from the
    // data dictionary using the functions in OptReptable.c
    void readParent(); // read the superType of this type
    void readChild(); // not used

  public:
    void readAttributes(int varFlag=0); // read the attributes of the type
    void checkAttrTypes();
  protected:
    void addChild(TypeData*);
    void addMember(TypeData*);  // like addChild() but recursive

  public:
    char *TypeName()  { return fTypeName;  }

  public:
    int  readData();
    // read parameters from disk file.  Pass the name of the type as
    // parameter. This function will be called for each type.  This
    // returns 0 if successful.

};
```

class TypeDataType

```
/*
 * class TypeDataType  is the same as the TypeData class except that it contains
 * more fields for its children.
 *
 */
class TypeDataType : public TypeData {
  private:
    TypeData *fChildT[MAX_FAM_TYPE];

  public:
    TypeDataType(char*);
    ~TypeDataType();

    TypeData *Child(int dim) { return  fChildT[dim]; }

};
```

class PathString

```
class PathString : public SchemaMgr {
  private:
    int   fn;
    char  fName[MAX_PATHSTRING_LEN];

    // NOTE: The first field of this Map is fAttrTD[1] and refers to
    // C(1) (which is same as C(1,1)). fAttrTD is a 2 dimensional array
    // since typedef MapiTDP is itself a Map(int, TypeDataPtr).
    // C(1,1) is fAttrTD[1][1].
    // Note that Attribute A(i) is of type C(i+1,1).
    Map(int, MapiTDP)  fAttrTD;

    TypeData  *targetType;     // See [BERT92] p8 para 2.2
    ExStrategy  fExStrategy;

  public:
    PathString(char*);   // Path begins with an object in the data
    // dictionary
    PathString(char*,char*);  // Arg 2 is Type C(1)
    PathString(PathString&);  // copy constructor
    ~PathString();

    void parseString(char*);
    void addFirstObject(char*);
    char *getAndRemoveFirstToken(char*,char*);

    void addAttrI(char*);       // add type (i) to fAttrTD[*]
    int  addAttrJ(TypeData*,int*);  // add types (j) to fAttrTD[i][*]

    int  addAttributeToPath(char*); // Atribute name as a string
    int  addToPath(char*,char*);    // Atribute name and type as string
    int  addToPath2(char*,char*);   // Used by addAttrToPath & addToPath

    void getAttrType(char*,char*);  // Args are: Attribute, Type returned.
    void printPath2();  // Path with all the type names
    void printPath3();  // Path with sub types and names
    void printPath4();  // as 3 with parameter data

  public:
    int   n()       {  return fn; }
    int   len()     {  return fn; }

    char  *Name()   {  return fName; }  // string version of the path
    char  *attrName(int i);
    char  *attrName(int i ,char*);    // The ith Attribute name from the
                                      // string representation of the
      // PathString.  @nd arg is return value.

    TypeData   *C(int,int);  // returns C(i,j)
    TypeData   *C(int);      // returns C(i,1)
    TypeData   *C1();        // returns C(1,1)

    TypeData   *dom();       // returns C(n,1)

  public:
    Boolean  AttrTDInArray(char*,int);
    Boolean  AttrTDInArray(char*,int,int);

  public:
    // COMPUTED FUNCTIONS based on [BERT92] and [BERT91d]

    int    RefBy(int,int,int,int);
    double Pr11(int,int,int);
    double Pr12(int,int,int);
    int    E1(int,int,int,int);
    double PA(int);

    int    RefByh(int,int,int);
    double Pr111(int,int);
    double Pr121(int,int);
    int    E11(int,int,int);

    int    Ref(int,int,int,int);
    double Pr21(int,int,int);
    double Pr22(int,int,int);
```

```
    int    E(int,int,int);
    double Pr31(int,int);
    double Pr32(int,int);
    double PH(int);

    int    kbar(int,int);

    int    Refh(int,int,int);
    int    khbar(int);
    double power(char*,double,double); // as pow() with error message.

    // Physical data parameters
    int   NI(int i);
    int   NI2(int i);        // for TESTING mode only. Undefined if TESTING
                              // not defined during compilation.
    int   NIk1();
    int   V1(int);
    int   V2(int);
    int   V3(int);
    int   V4(int);
    int   targetTypeIn(int i);

    int  getCost(ExStrategy); // Argument is Execution Strategy

    int  AP(int i);
    int  AP2(int i);         // for TESTING only
    int  sumAP(int,int);

    int  SORT_NI(int i);
    int  sumSORT_NI(int,int);

    int  H(int,int,int);     // [Yao77]

    int  nc(int);
    int  D(int,int);
    int  D(int);
    int  N(int,int);
    int  Nh(int);
    int  fan(int,int);
    int  fan(int);
    int  d(int,int);
    int  d(int);
    int  k(int,int);
    int  kh(int);

    int P(int,int);
    int Ph(int);
    int r(int);

}; // end class PathString
```

Appendix B: Function Descriptions

The class layout is as shown on the diagram on page 26. The following paragraphs describe the function of each class, and the semantics of the class methods.
Note that the functions are **not** described where they are thought to be **self explanatory**.

B.1 Query Rep.

B.1.1: class ArcNode : OptRepMetaClass

This class is a superclass for the Node and Arc classes of the query rep.
The main function of this class is to provide functions for accessing the Annotations. In addition there are functions to keep track of the position of a node or an Arc in the tree (The IDString) and the position of a Node on the output plot (an X and Y coordinate).

```
ArcNode();
ArcNode(qTreeType*, ArcNode*)
```
Arguments are qTreeType node from the "gcl" rep, and a pointer to the parent of class ArcNode.

.

```
~ArcNode();

void  *GetAnnotation(USLString index)
void  SetAnnotation(USLString index, void *value)
Map(USLString,voidPtr)  Annotations()

int numAnnotations()
```
The number of different Annotations.

```
void     SetPathAnnot(PathString*);  // put this path only in annot
void     AddPathAnnots(vlist*);      // Add an annot to this one
void     SetCostAnnot(PathString*);  // 4 values of cost in annot
void     CopyPathAnnot(ArcNode*);  // copy another path annot to *this
void     CopyCostAnnot(ArcNode*);  // copy another cost annot to *this
vlist*   CopyPathVList(vlist*);
void     freePathVList(vlist*);
```
The above functions relate to the "path" and "cost" annotations only, and are used to copy the annotations up the tree as they are built, or calculated..

```
public:
void PrintAnnotations();
```
Called by printNode() to print the Annotations.

The following functions keep track of the shape of the tree.

```
char *IDStr()
void setIDStr(ArcNode*);
int  dimX()
int  dimY()       { return fDimY;  }
void setDimX(int value) { fDimX = value; }
int  SiblingNum() { return fSiblingNum;  }
```

B.1.2: class Op : OptRepMetaClass

:

`Op()`
Empty Constructor

`Op(char*, int ,int)`
Constructor used when building the rep from the "gcl" rep.
Arguments are Operator Name, Arity, number of Inputs. Arity is Number of Arguments, Input + Other.

`Op(Op&)`
Copy constructor.

`~Op();`

`char *Name()`
Returns operator name.

`operType OperType()`
returns operator type, from the enum operType. (OperType is defined in OptRepIntRep.h.

```
int  Arity()
int  NumInputArguments()
int  NumOtherArguments()
```

`int  isAttribute()`
Returns TRUE if the operator name is NOT predefined. This is taken to mean that the operator is in fact an attribute of the previous type. Called by FunctionNode::BuildQueryPaths() and FunctionNode::print-Node().

`void PrintOp()`
Called by FunctionNode::printNode() to print out the fields of this class.

B.1.3: class Node : ArcNode

The class Node, has no data fields.

```
Node();
Node(qTreeType*, Arc*);        // constructor from "George" rep
Node(Node&);                   // copy constructor
~Node();

virtual Arc  *Parent(int) = 0;
```

Returns the parent Arc of this node.

```
qTreeType *CreateQueryTree(qTreeType*)
```

Called by DataNode::CreateQueryTree() and FunctionNode::CreateQueryTree().

B.1.4: class FunctionNode : Node

```
FunctionNode();

FunctionNode(qTreeType *gTree, DFArc *parentDF)
```

Constructor from "gcl" rep

```
FunctionNode(FunctionNode&)
```

Copy constructor not yet implemented.

```
~FunctionNode();

Arc  *Parent(int dim=0) { return fParent[dim]; }
FDArc *Child(int i,int dim=0)
```

Returns NthInput(i) up to (numInputs()-1) then returns NthOther(i-numInputs())

```
DataNode *GChild(int i,int dim=0)
```

Returns Child of Child function above. This is a shorthand that skips the Arc class.

The argument dim in the following functions is not used at present.

```
FDArc  *NthInput(int n, int dim = 0);
FDArc  *NthOther(int n, int dim = 0);
int    numInputs() { return fNumInputs; }
int    numOthers() { return fNumOthers; }
void   IncInputs(FDArc *newFD);
void   IncOthers(FDArc *newFD);
DataNode *ReplaceNthInput(int n, InputDataNode *theNode, int dim = 0);
DataNode *ReplaceNthOther(int n, OtherDataNode *theNode, int dim = 0);

Op   *Oper() { return fOper; } // because operator is a reserved word in C++
int  ChangeOper(Op);
```

Functions relating to the operator of theis FunctionNode. Named Oper because operator is a reserved word in C++.

```
Boolean  BuildQueryPaths(Boolean)
```
This function will build the "path" and "cost" annotations. In the FunctionNode the cost annotations are calculated.

```
// a function to print the contents of the node
void PrintNode()
```

```
void addIDStr()
```
Creates the ID string by concatenating the correct entry for this node to that of the parent.

```
qTreeType *CreateQueryTree(qTreeType*)
```
Used to rebuild a query tree of type qTreeType.

B.1.5: class DataNode : Node

```
DataNode();
// Takes an EQUAL query as arg. 2nd arg is flag to build Paths.
DataNode(USLString, Boolean buildQP=FALSE);
```
Constructor. Used to create a EREQ query tree from an equal string. This constructor calls the "gcl" function createQueryTree() and passes the resulting qTreeType tree to the next DataNode constructor. The fields of the root DataNode are then copied into this DataNode. The second argument is a flag to build Paths The default is NOT to build the paths. This enables a tree to be built without having the .TypeCostData file available.

```
DataNode(qTreeType*)
```
Constructor taking a "gcl" qTreeType as argument. This is used to create a ROOT node.

```
DataNode(qTreeType*, FDArc*)
```
Constructor taking a "gcl" qTreeType and parent DFArc as arguments. This is used to recursively form the non-root DataNodes in the tree.

```
~DataNode()
```

```
Arc  *Parent(int dim = 0)
DFArc  *Child(int dim = 0)
```
See note regarding dim under FunctionNode.

```
FunctionNode *GChild(int dim = 0)
```
See note regarding GChild() under FunctionNode.

```
void *Data(int dim = 0)
void SetData(void *data, int dim = 0)
```
The above functions refer to the data field of the DataNode. This has not yet been used.

```
char  *DataType()
void  SetDataType(char *dataType)
char  *Name()
void  SetName(char *name)

void  Splice(DataNode *theNewNode);  // splice the new Node in for this node
```
This function not yet implemented.

```
Boolean IsMultivalued()
int     NumValues()
```
The above two functions refer to the "size" of a set, if the type of the data is multivalued (ie a SET).

```
Boolean  IsLeafNode()
Boolean  IsLocalVar()  // Does this node represent a local variable
```
If the variable is in the avail list, it is a Local Variable.

```
Boolean  IsGlobalVar()
```
A DataNode is assumed to be a global variable if it is not a local variable.

```
Boolean  BuildQueryPaths(Boolean)
```
This function will build the "path" and "cost" annotations. In the DataNode the PathString's are built.

```
void PrintNode()
```
A function to print the contents of the node

```
void addIDStr()

void copyFields(DataNode*)
```
Called by the constructor DataNode(USLString,Boolean) after a secondary node has been used to build a new query tree.

```
qTreeType *CreateQueryTree(qTreeType* QTN=NULL);
```
Recursively rebuilds the qTreeType tree from the EREQ rep (This rep). With no argument, builds a root node.

B.1.6: class InputDataNode : DataNode

```
InputDataNode(qTreeType*, FunctionNode*, FDArc*)
~InputDataNode()
```

B.1.7: class VInputDataNode : InputDataNode

```
VInputDataNode(qTreeType*, FunctionNode*, FDArc*)
~VInputDataNode()
```

B.1.8: class CinputDataNode : InputDataNode

```
CInputDataNode(qTreeType*, FunctionNode*, FDArc*)
~CInputDataNode()
```

B.1.9: class OtherDataNode : DataNode

```
OtherDataNode(qTreeType*, FunctionNode*, FDArc*)
~OtherDataNode()
```

B.1.10: class Arc : ArcNode

```
Arc(qTreeType*, Node*)
~Arc()
```

B.1.11: class DFArc : Arc

```
FDArc()
FDArc(qTreeType*, FunctionNode*)
~FDArc()
```

The argument dim in the following applies to the dimension and is always set to zero.

```
FunctionNode *Parent(int dim = 0)
DataNode     *Child(int dim = 0)
void        SetChild(DataNode*,int dim = 0)
```

```
void  MakeChild(qTreeType*, FunctionNode*)
```

This function decides whether the grand child is a VInput, CInput or Other DataNode.

```
Boolean  BuildQueryPaths(Boolean)
```

Called by DataNode::BuildQueryPaths. Copies the Annotations from its child to itself.

```
void addIDStr()
```

See FunctionNode::addIDStr().

B.1.12: class FDArc : Arc

The functions for FDArc are similar to DFArc, but returning different types. Refer to the listings above.

B.2 Schema Manager

B.2.1: class SchemaMgr : OptRepMetaClass

```
void
SchemaMgr::buildTypeMap()
```

Called by SchemaMgr constructor. The class SchemaMgr is a virtual class, so the constructor is only called by the sub classes DataType and PathString.

Builds an associative array of type Map(USLString, TypeDataPtr). The type information for this array comes from the "gcl" data dictionaries.

```
void
SchemaMgr::printTypeMap()
```

Called by the user.

Prints out details of all instances of TypeData in the TypeMap array.

```
TypeDataPtr
SchemaMgr::getTypeData(char*)
```

Argument is the type name.

Called by any function requiring access to details of a specific Type.

Returns a pointer to the specific instance of TypeData.

```
TypeDataPtr
SchemaMgr::getTypeDataErr(char*, char*)
```

Second argument is an error message.

Functionality as getType Data() above, but prints an additional error message if the type is not found.

```
Boolean
SchemaMgr::TDAttrExistsErr(char*, char*, char*, TypeData**, int*)
```

Arguments are (input) Type name, attribute name, error message.

(Output) Pointer to specified TypeData and index to the specified attribute.

Called by access functions which need data relating to a specific attribute of the type. If TRUE then the type instance and index of the attribute in that type are returned.

Returns TRUE if the specified attribute of the specified type is found, else returns FALSE.

Access functions are provided which supplement the PathString access functions. These provide access to the cost parameter fields in instances of TypeData by reference to their names. The first two Get functions take argument of Type name, and the rest take type name, attribute name. The Set functions take the same arguments plus a value as the last argument.

```
int   SchemaMgr::GetN(char*)          void SchemaMgr::SetN(char*, int)
int   SchemaMgr::GetP(char*)          void SchemaMgr::SetP(char*, int)

int   SchemaMgr::GetD(char*,char*)    void SchemaMgr::SetD(char*, char*, int)
int   SchemaMgr::Getfan(char*,char*)  void SchemaMgr::Setfan(char*,char*, int)
int   SchemaMgr::Getd(char*,char*)    void SchemaMgr::Setd(char*, char*, int)
int   SchemaMgr::Getr(char*,char*)    void SchemaMgr::Setr(char*, char*, int)
```

Boolean
FirstErr(char* msg=NULL)
If called with an argument, the static field in the function is set to TRUE and returns FALSE. The next call returns TRUE, subsequent calls return FALSE.

B.3 Cost Functions

B.3.1: class TypeData : SchemaMgr

An instance of the class exists for each type in the data schema. The class contains data on both the type realationships, or structure of the schema (ie superclass(es) and subclass(es)), and cost data. A structure array is provided in the class to store details relating to each attribute.

TypeData::TypeData(USLString)
This is the standard constructor for an instance of the DataType class. The argument is a type name. This is called by the function **SchemaMgr::buildTypeMap()**.

char*
TypeName()
Returns the name of this type.

The following functions return super type and sub type repectively.
TypeData* TypeData::Parent(int)
TypeData* TypeData::Child(int)

The following functions are self explanatory.
int TypeData::NumParents()
int TypeData::NumChildren()
int TypeData::NumAttributes()

TypeData::readData()
Called by SchemaMgr::buildTypeMap().
Reads the file of cost data, looks for this type name, and attribute names, and reads in the relevant type cost parameters.

Boolean
TypeData::AttrExists(char*)
Returns TRUE if the type has an attribute of the name specified.

```
Boolean
TypeData::AttrExists(char*, int*)
```
As above, but returns the index of the attribute in the second argument.

The following access functions relate to fields of each attribute. The first three fields may only be set from inside an instance of TypeData.

```
char*     AttrName(int i)
char*     AttrType(int i)
```
Returns the type of the attribute. The type may be "Set[T]" or "T", depending on whether the type is multi-valued.

```
Boolean   AttrIsSet(int i)
```
Returns TRUE if this attribute is multivalued.

```
int       AttrD(int i)                  void      SetAttrD(int i, int value)
int       Attrfan(int i)                void      SetAttrfan(int i, int value)
int       Attrd(int i)                  void      SetAttrd(int i, int value)
int       Attrr(int i)                  void      SetAttrr(int i, int value)
```

```
TypeData::nc()
```
Returns the number of members of the type hierarchy based at this type, including this type.

The following functions are access functions for the cost parameters N (Cardinality) and P (Number of disk pages used by the type).

```
int    N()                              void SetN(int value) { fNij = value; }
int    P()                              void SetP(int value) { fPij = value; }
```

```
void readAttributes(int varFlag=0); //
```
Read the attributes of the type from the "gcl" data dictionaries.
Called by the TypeData constructor, and, for VARIABLE only, called by buildTypeMap().

```
void checkAttrTypes()
```
Checks to see if the attributes of this type exist in the typeMap.

The following are private functions that get type and attribute information from the "gcl" data dictionaries using the functions in OptReptable.c

`void readParent()`

Reads from the table of types in .type_dict. to find super type(s). If a parent is found in the table, the parent's functions addChild() and addMember() are called to add this instance as a sub type of the super type.

Called by TypeData constructor.

`void readChild()`

Returns sub types of this type. This is not used, since the sub types are aften not in the table when the constructor is called.

The following are protected functions:

`void addChild(TypeData*)`

Increments the field fNumChildren, and adds the calling type to the array of fChildren.

Called by a sub type of the type.

`void addMember(TypeData*); // like addChild() but recursive`

Called by a sub type (or lower) of the type. See readParent() above.

B.3.2: class TypeDataType : TypeData

Only one instance of this class exists. It is for the meta-type "Type". The difference between this type and DataType is that this has a larger array for its sub types.

B.3.3: class PathString : SchemaMgr

PathString constructors:

PathString(char*)

Argument is the name of a variable in the data dictionary. [Note. The constructor needs to be revised to use SchemaMgr functions rather than OptRepTable functions.]

PathString(char*,char*)

Argument 1 is a variable name, argument 2 is the type of the variable. This constructor is used when the variable is a temporary variable declared in the tree. This is, I think, the closest to the Bertino model.

PathString(PathString&)

Copy constructor.

~PathString();

Destructor.

void parseString(char*)

Input argument is a path string as a string.

The assign an instance of TypeData to each Attribute in the path.

Called by PathString::PathString(char*, char*) and PathString::PathString(char*).

void addFirstObject(char*);

Input argument is a path string as a string.

This assumes that the first object in the path is a variable in the data dictionary, and starts the PathString with this variable and its type.

Called by PathString::PathString(char*).

char *getAndRemoveFirstToken(char*,char*);

Input argument is a path string as a string.

Output argument is the token from the front of the string. Note that the string is returned without the first token.

Called by parseString(), addFirstObject() , attrName().

void addAttrI(char*)

Input argument is a type name, may be of the form T or Set[T].

Adds the type T and its sub classes to the array fAttrTD using addAttrJ().

Called byPathString(char*,char*), addFirstObject(), addToPath2().

int addAttrJ(TypeData*,int*); // add types (j) to fAttrTD[i][*]

Input arguments are the type to be added and the current number of types in this hierarchy, index j.

Add the TypeData to the array fAttrTD as the next jth entryand increment the number of types in this hierarchy. This is the only place that an addition is made to fAttrTD.

Called by addAttrI(), addAttrJ().

`int addAttributeToPath(char*)`
Input argument is attribute name. as a string.
Adds the named attribute to the end of the PathString.
Called by parseString().

`int addToPath(char*,char*)`
Arguments are attribute name and type name.
The named attribute and its type are added to the path.
NOT Called.

`int addToPath2(char*,char*);`
Arguments are attribute name and type name.
The named attribute and its type are added to the path.
Called by addAttributeToPath(), addToPath().

`void getAttrType(char*,char*);`
Input argument is attribute name,
Output argument is attribute type.
Reads the last attribute name in the string representation of the path, and looks for this type in the `.type_dict`. [Note. This needs to be revised to use the Schema Manager. functions]
Called by addAttributeToPath(), addToPath().

The following three print functions print the path to the standard output in increasing detail.
`void printPath2()`
Print Path and its types on one line. No sub-types shown.

`void printPath3();` **`// Path with sub types and names`**
Print Path and its types and sub types..

`void printPath4();` **`// as 3 with parameter data`**
Print Path and its types and sub types.. This version prints in a verbose form all the parameters used for calculating the cost of the path.

```
int   n()
```
Returns the length of the path. Field fn.

```
char  *Name()
```
Returns the path as a string.

```
char  *attrName(int i);
```
Input argument is the number of the attribute in the path.
Output argument is the attribute name.
Returns the ith Attribute name from the PathString.

```
char  *attrName(int,char*)
```
As above, but output argument is the attribute name returned.

```
TypeData   *C(int,int);  // returns C(i,j)
TypeData   *C(int);      // returns C(i,1)
```
Input arguments are indices into the array fAttrTD, i,j.
The above functions return an instance of TypeData from the array fAttrTD. The second returns C(i,1), the root of the hierarchy based at C(i,1).

```
TypeData   *dom()
```
The *Domain* of the path, as described by Bertino. Returns C(n,1).

```
Boolean  AttrTDInArray(char*,int);
```
First argument is a message to identify the caller. The second argument is index i into the array fAttrTD.
This checks to see if the array fAttrTD[i] exists.Returns TRUE if found. Prints an error message and returns FALSE if not found. [Note that fAttrTD is a two dimensional array.]
Called by all functions that access the array fAttrTD.

```
Boolean  AttrTDInArray(char*,int,int);
```
Similar to AttrTDInArray above, but the third argument is index j into the array fAttrTD. This therefore checks to see if the element fAttrTD[i][j] exists.

The following are COMPUTED FUNCTIONS based on [BERT92] and [BERT91d]
For details of the functions it is necessary to read the papers.

```
int  getCost(ExStrategy)
```
This function calculates the cost in disk acces of accessing the path given in this instance of PathString. The argument is the execution strategy as described in the text. See paragraph 3.4.9 Execution Strategy on page 14.

The first function in each of the following groups is one called by the getCost() function or a secondary function of getCost().. The following in each group are called by the first function.

```
int    RefBy(int,int,int,int);
double Pr11(int,int,int);
double Pr12(int,int,int);
int    E1(int,int,int,int);
double PA(int);

int    RefByh(int,int,int);
double Pr111(int,int);
double Pr121(int,int);
int    E11(int,int,int);

int    kbar(int,int);
int    khbar(int);

int    Ref(int,int,int,int);
int    Refh(int,int,int);

double Pr21(int,int,int);
double Pr22(int,int,int);
int    E(int,int,int);
double Pr31(int,int);
double Pr32(int,int);
```

```
double power(char*,double,double)
```
This function is the same as the math library pow(), but prints an error message if it detects a domain error.

```
// Physical data parameters
int   NI(int i);
int   NI2(int i);
int   NIk1();
```
Number of records to be searched for. The function is divided for TESTING purposes only, to enable an intermediate result to be printed. See also AP() below. NIk1() calculates NI() when the argument k==1.

```
int   V1(int);
```
Called by NI(k) in case NLRT.
```
int   V2(int);
```
Called by NI(k) in case NLRT and SDRT.
```
int   V3(int);
```
Called by NI(k) in case SDRT.
```
int   V4(int);
```
Called by AP(k) in case SDRT
Argument to each of the above 4 functions is k. 1<=k<=n(). They are called by NI(k) or AP(k) as shown.

```
int   targetTypeIn(int i);
```
Returns TRUE if the targetType is a member of the type hierarchy i.

```
int  AP(int i);
int  AP2(int i);
int  sumAP(int,int);
```
Describes the number of accessed pages. The function is divided into AP() and AP2() at TESTING stage only. sumAP() sums AP() over a range.

```
int  SORT_NI(int i);
```
Calculates the number of disk accesses required to sort the records. Argument is int k, 1<=k<=n.

```
int  sumSORT_NI(int,int);
```
Sums the above over several class hierarchies.Arguments are i,j. 1<=i<=j<=n.

```
int  H(int,int,int);
```
This function is based on the paper [YAO77]. It describes the "hit rate" achieved when searching for records on the disk. Arguments are k, the number of records being searched for, m, the blocks containing the n records, and n, the number of records to be searched.

```
int  nc(int);
```
Number of types in the hierarchy based at C(i,1). Argument is int i. 1<=i<=n().

The following 13 functions are access functions as described in [BERT92]. For the description of each one. see Figure 4: Cost Parameters (continued) on page 17.

```
int  D(int,int);
int  D(int);
int  N(int,int);
int  Nh(int);
int  fan(int,int);
int  fan(int);
int  d(int,int);
int  d(int);
int  k(int,int);
int  kh(int);

int P(int,int);
int Ph(int);
int r(int);
```

The next six functions are Setting functions for the 6 cost parameters that are settable. The names are not SetN() etc, since this would hide the declaration of SetN() in class TypeData. The arguments of each one

are int i, int j, and int vlaue. The i and j refer to the 2 dimensions of the PathString type array; 1<=i<=n, 1<=j<=nc.

```
void  SetD(int,int,int);
void  SetN(int,int,int);
void  Setfan(int,int,int);
void  Setd(int,int,int);
void  SetP(int,int,int);
void  Setr(int,int,int);
```

B.3.4: Functions in file OptRepTable.c

The following functions from OptRepTable.c are used by the classes in files OptRep and OptRepCost, and were written as part of this project. For descriptions of the API functions provided by gcl, see the paper "Documentation for the Query Tree Interface".

```
char  *getTypeFromSet(char*,char*);     /* by acm */
```
Input arguments is type name, possibly as Set[T].
Output argument is type name T.

```
int   typeIsSet(char*);                 /* by acm */
```
Input arguments is type name, possibly as Set[T]
Returns TRUE if Set[T] or List[T].

```
int   typeExists(char*)
```
Argument is type name.
Returns TRUE if type exists in .type_dict.

```
int   attributeExists(char*,char*)
```
Input arguments are type name, attribute name.
Returns TRUE if the attribute exists.

```
char  *getAttribute(char*,int,char*,char*)
```
Input arguments are type name and index.
Output arguments are attribute name and attribute type. This type may be of the form "T" or "Set[T]".
Returns attribute name.

```
char  *getAttributeType(char*,char*,char*)
```
Input arguments are type name and attribute name.
Output argument is attribute type.
Returns attribute type.

```
int   objectExists(char*)
```
Argument is a variable name.
Returns TRUE if the named variable exists in .data_dict.

char *getObjectType(char*,char*,int)

Input argument is a variable name in .data_dict..

Output arguments are type nae and Boolean variable which is TRUE if the type is a set.

void readText()

Read a data schema written in a text file and put the data into the data dictionary files **.type_dict, .attr_dict and .data_dict**. This enables a data schema to be input far more easily than the previous method, in which each type, attribute and variable had to entered separately.

The following three functions all retrieve data from the gcl data dictionaries.

char *getObjectFromTable(int, char*)

Input argument is index into the variable table of the data dictionary.

Output argument is variable name. (Object refers to a variable).

Read the **.data_dict** and return the indexed entry. The variable name is returned.

char *getTypeFromTable(int, char*)

Input argument is index into the type table of the data dictionary.

Output argument is type name.

Read the **.type_dict** and return the type field of the indexed entry. The type name is returned.

char *getSuperTypeFromTable(int, char*);

Input argument is index into the type table of the data dictionary.

Output argument is a type name from the super type field of the type dictionary.

Read the **.type_dict** and return superType field of the indexed entry. The super type name is returned.

Appendix C: Emacs conversion function o2-equal-conv.

The following emacs lisp function "o2-equal-conv" can be used to convert a query in the O2 syntax as written in the file **test-queries** to the form used by the optimizer parser. The usage is described in the listing.

The file name is **o2-equal-conv.el** and is in the **Rep/altair** directory.

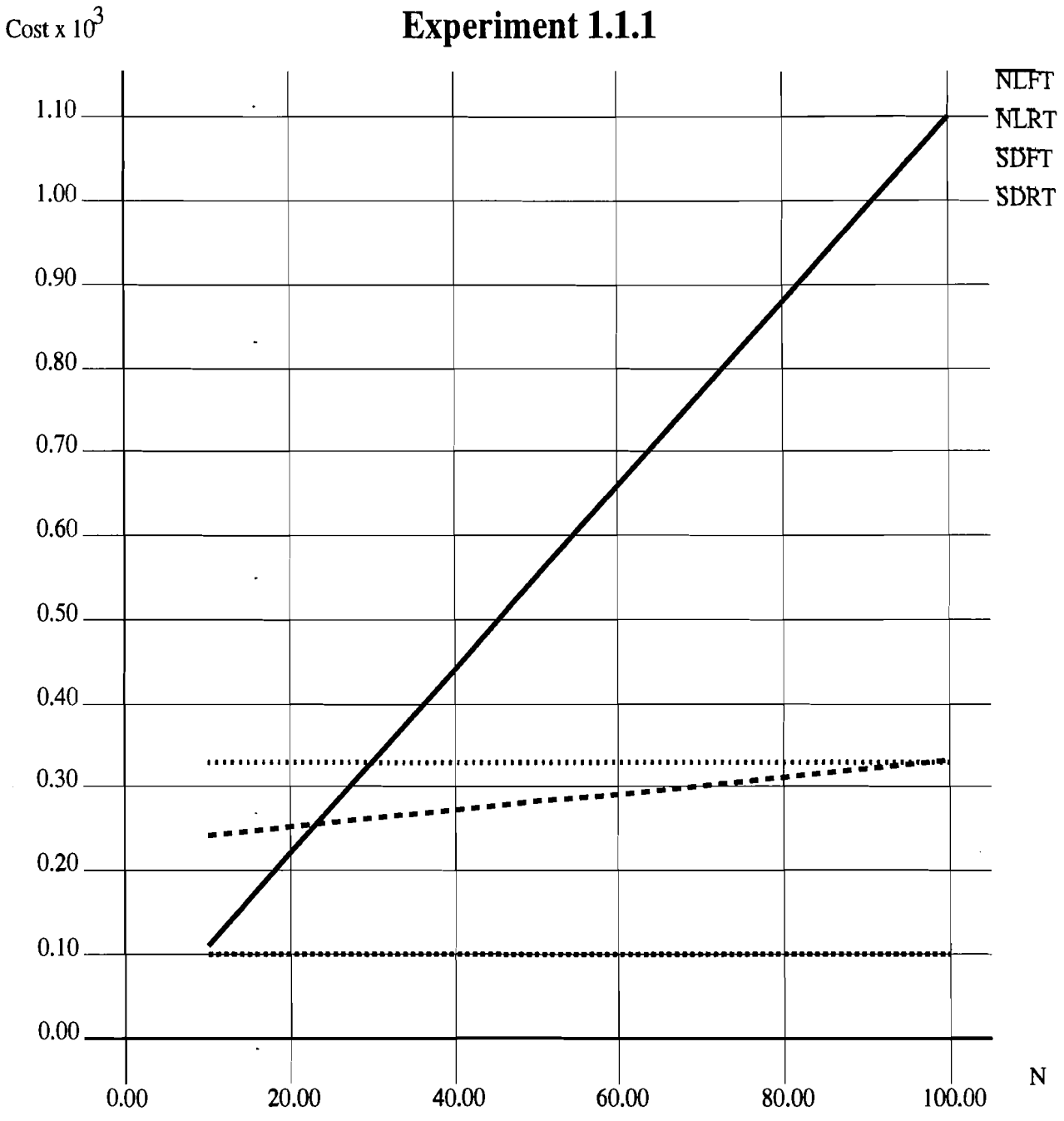
```
;; o2-equal-conv.el
;; To convert from o2 to Equal.

;; 1.  lambda(x)  --->  $x
;; 2.  M:m,T: ---> (M,m),T,
;; 3.  IN  ---> memberof
;; 4.  AND ---> &
;; 5.   = ---> ==


;;-----------------------------------------------------------------------

(defun o2-equal-conv ()
"Convert o2 syntax query to Equal syntax"
(interactive)
(setq close-b (string-to-char ")" ))
(save-excursion
  (while (not (eq  (point)  (point-max)))
    (o2-equal-conv-lambda)
    (forward-char)
    )
  )
(save-excursion
```

```
  (while (not (eq  (point)  (point-max)))
    (o2-equal-conv-colon)
    (forward-char)
    )
  )
(save-excursion (replace-string "[" "{"))
(save-excursion (replace-string "]" "}"))
(save-excursion (replace-regexp "AND" "&" t))
(save-excursion (replace-regexp "IN" "memberof" t))
(save-excursion (replace-regexp "=" "==" t))
(save-excursion (replace-string "." "@"))
)


(defun beginning-sexp ()
"Move point to parenthesis at start of current balanced expression."
(interactive)
(setq open-b1 (string-to-char "(" ))
(setq open-b2 (string-to-char "[" ))
(setq open-b3 (string-to-char "{" ))
(while (not (or (char-equal (preceding-char)  open-b1)
                                                  (char-equal (preceding-char)  open-b2)
                                                  (char-equal (preceding-char)  open-b3)
                                                  )
                                                      )
  (backward-sexp)
  )
(forward-char -1)
)

(defun o2-equal-conv-lambda ()
"Replace lambda(x) with $x"
(interactive)
(setq close-b (string-to-char ")" ))
(setq comma   (string-to-char "," ))
(cond ((and (<  (point)  (- (point-max) 7))
                                                     (string-equal (buffer-substring (point) (+
(point) 7))
                                                   "lambda(" ))
       (delete-char 7)
       (insert "$")
       (while (not (char-equal (char-after (point))  close-b))
                                                    (forward-char)
                                                    (cond ((char-equal (char-after (point))  comma)
                                                   (delete-char 1)  ;; in case of >1 variable
                                                   (insert "$")))   ;; replace "," with "$"
                                                    )
       (delete-char 1)
       )
      )
)


(defun o2-equal-conv-colon ()
"Replace X:x,Word():  with (X,x),(Word(),  OR  X:x,Word():  with (X,x),(Word,"
(interactive)
(setq colon (string-to-char ":" ))
(cond ((char-equal (char-after (point))  colon)
       (setq this-point (point))
       (backward-wsexp)
       (insert "(")
       (forward-wsexp)
       (delete-char 1) (insert ",")
       (forward-wsexp)
       (delete-char 1) (insert "),(")
       (save-excursion
                                                    (beginning-sexp) (forward-sexp) (forward-char -
1) (insert ")")
                                                    )
       (forward-wsexp)
       (while (not (char-equal (char-after (point))  colon))
                                                    (forward-char))
       (delete-char 1)
       (insert ",")
       )
      )
)


(defun forward-wsexp ()
"move forward a word, or if word followed by (, forward to end of )."
```

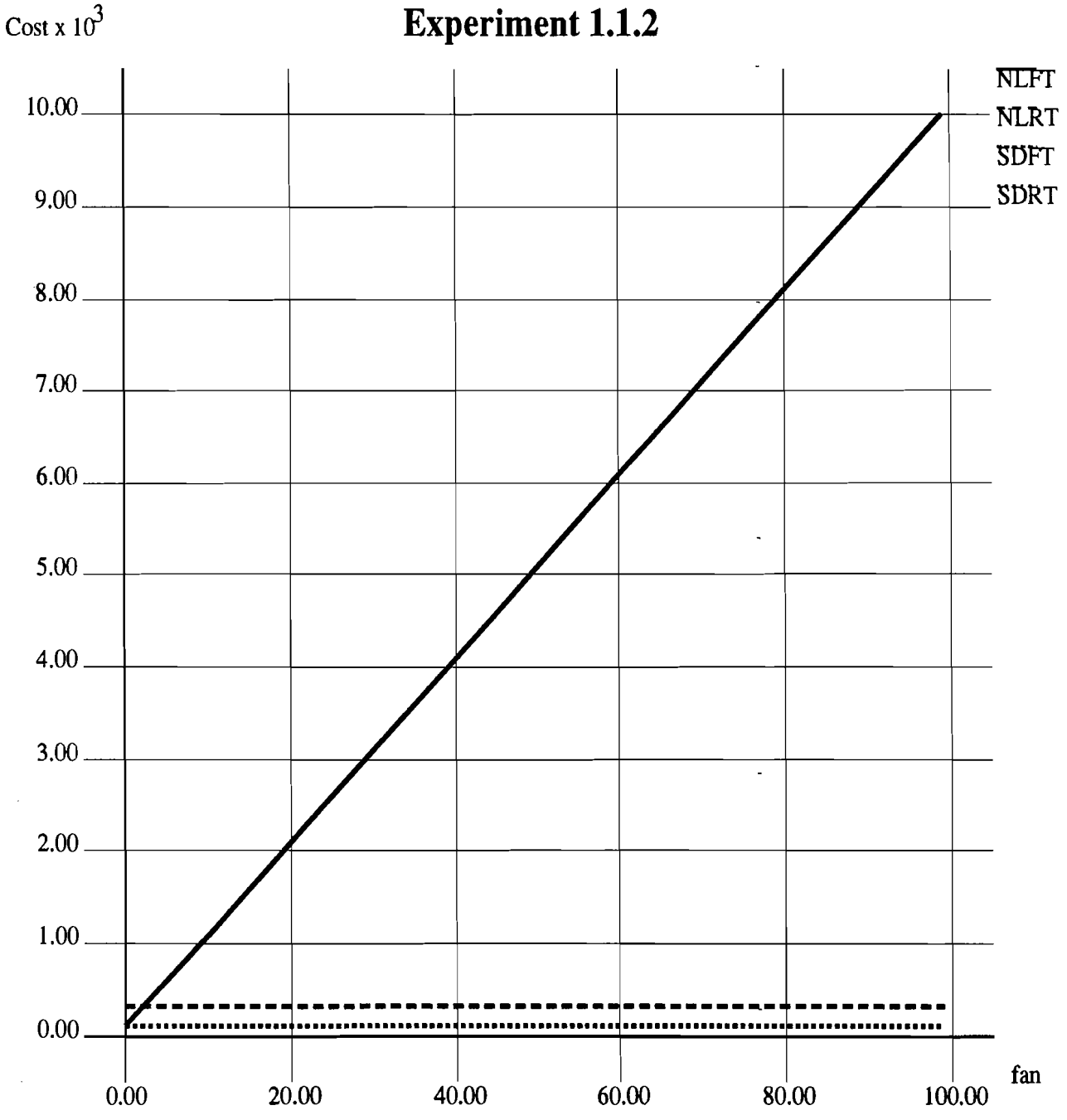
```
(interactive)
(setq open-b (string-to-char "(" ))
(forward-word 1)
(cond ((char-equal (char-after (point))  open-b)
       (forward-sexp)
       ))
)

(defun backward-wsexp ()
"move backward a word, or if word followed by (, backward to end of )."
(interactive)
(setq close-b (string-to-char ")" ))
(cond ((char-equal (preceding-char)  close-b)
       (backward-sexp)
       ))
(forward-word -1)
)

;; End of o2-equal-conv.el
;;---------------------------------------------------------------------
```

Appendix D: Experimental Results

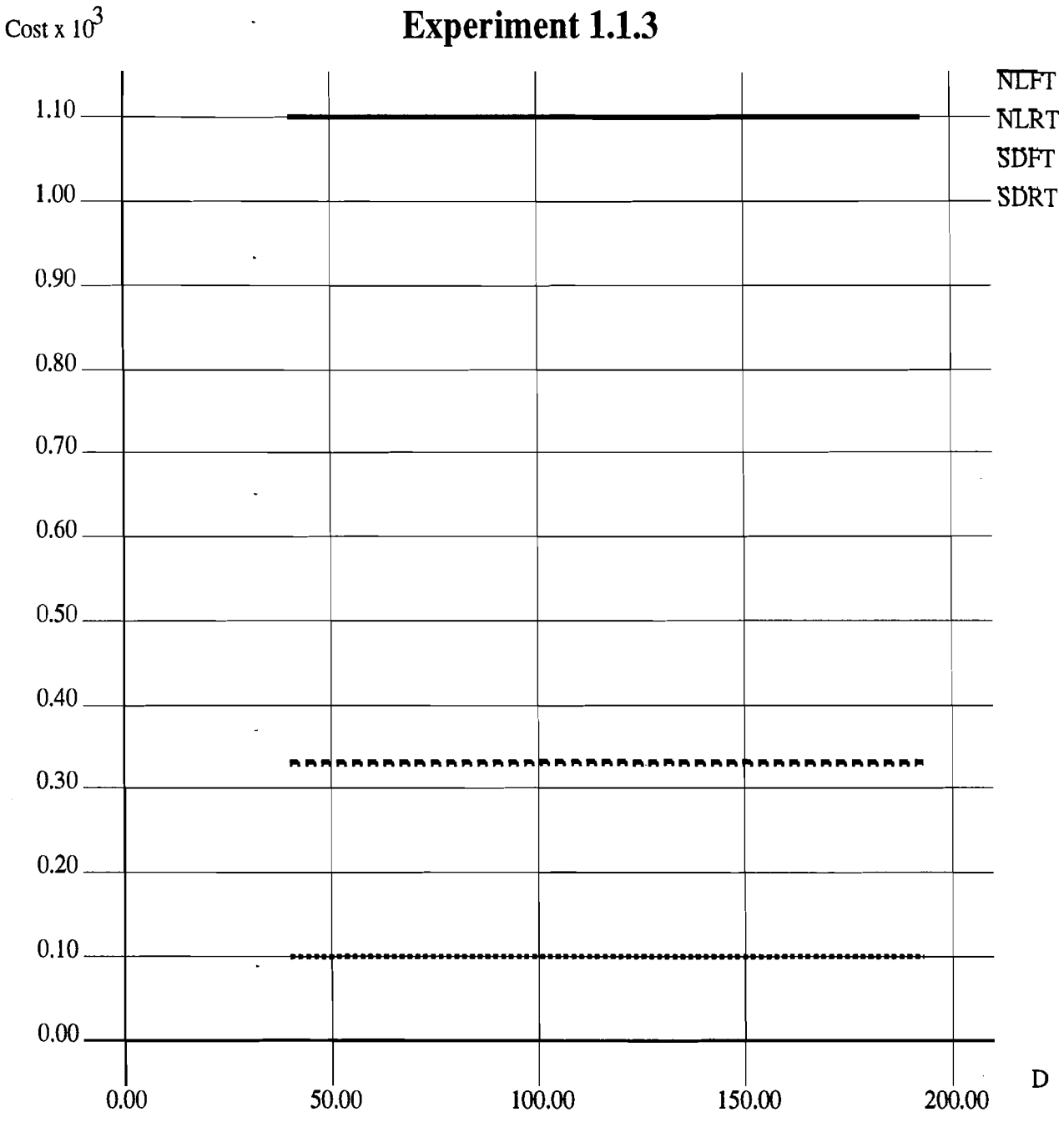
The following pages illustrate the experiments carried out using the cost model. A description of these experiments is in the text. See section 4.0 "Experiments" on page 16.



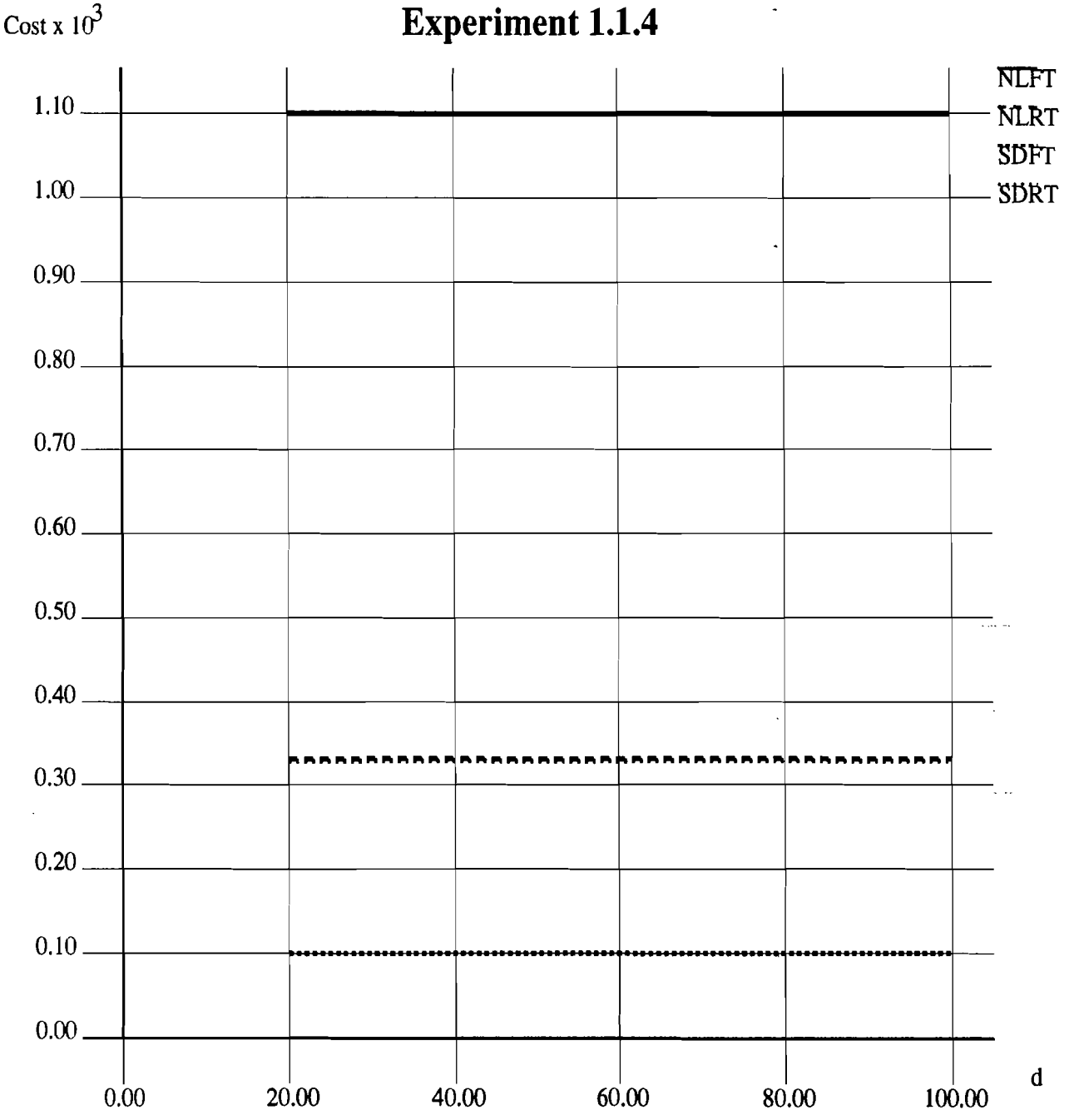

```
# File: "plotAll0101.txt".

#       Parameter          Cost          Cost          Cost          Cost
#               N         NLFT          NLRT          SDFT          SDRT
#
              10          110           242           100           329
              20          220           252           100           329
              30          330           262           100           329
              40          440           272           100           329
              50          550           282           100           329
              60          660           292           100           329
              70          770           302           100           329
              80          880           312           100           329
              90          990           322           100           329
             100         1100           332           100           329
```



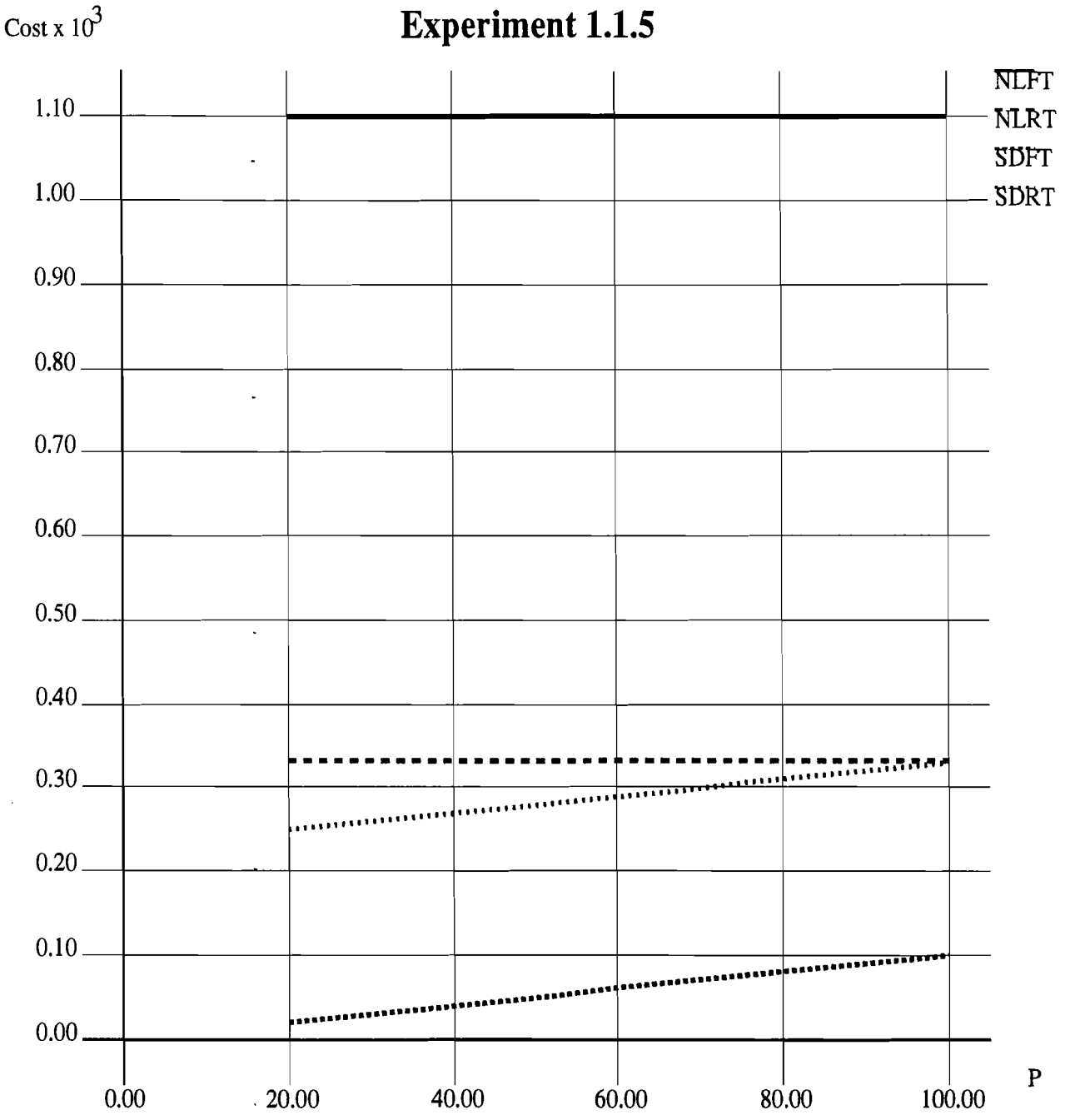
# Parameter fan	Cost NLFT	Cost NLRT	Cost SDFT	Cost SDRT
0	100	332	100	329
11	1200	332	100	329
22	2300	332	100	329
33	3400	332	100	329
44	4500	332	100	329
55	5600	332	100	329
66	6700	332	100	329
77	7800	332	100	329
88	8900	332	100	329
99	10000	332	100	329




```
# File: "plotAll0103.txt".

#       Parameter          Cost          Cost          Cost          Cost
#             D             NLFT          NLRT          SDFT          SDRT
#
             40             1100           332           100           329
             57             1100           332           100           329
             74             1100           332           100           329
             91             1100           332           100           329
            108             1100           332           100           329
            125             1100           332           100           329
            142             1100           332           100           329
            159             1100           332           100           329
            176             1100           332           100           329
            193             1100           332           100           329
```

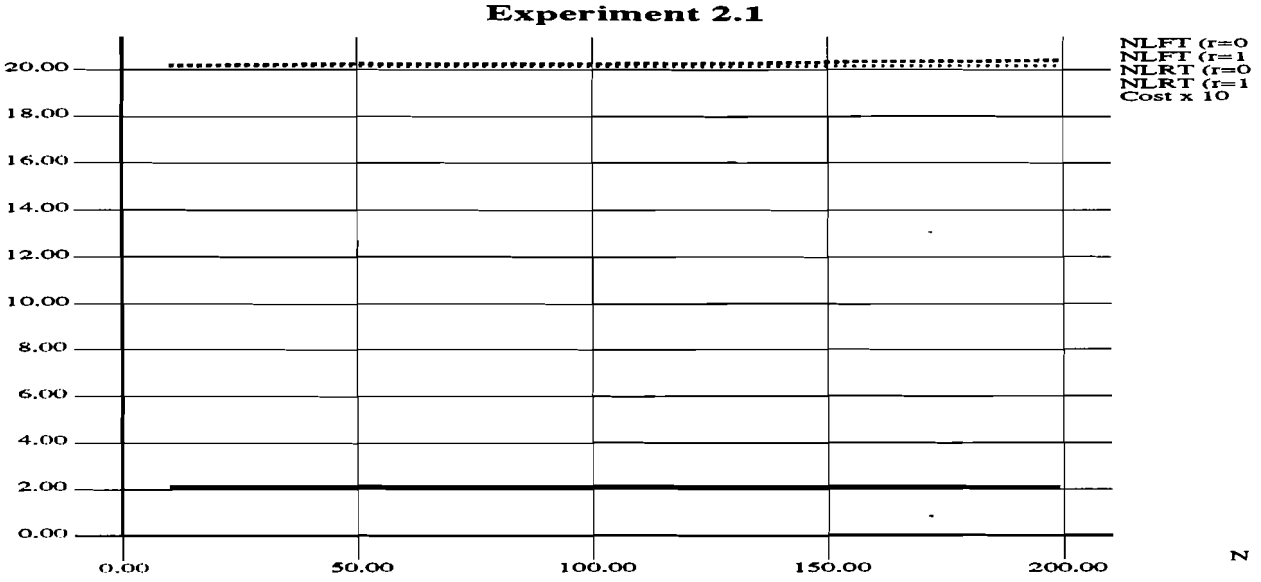


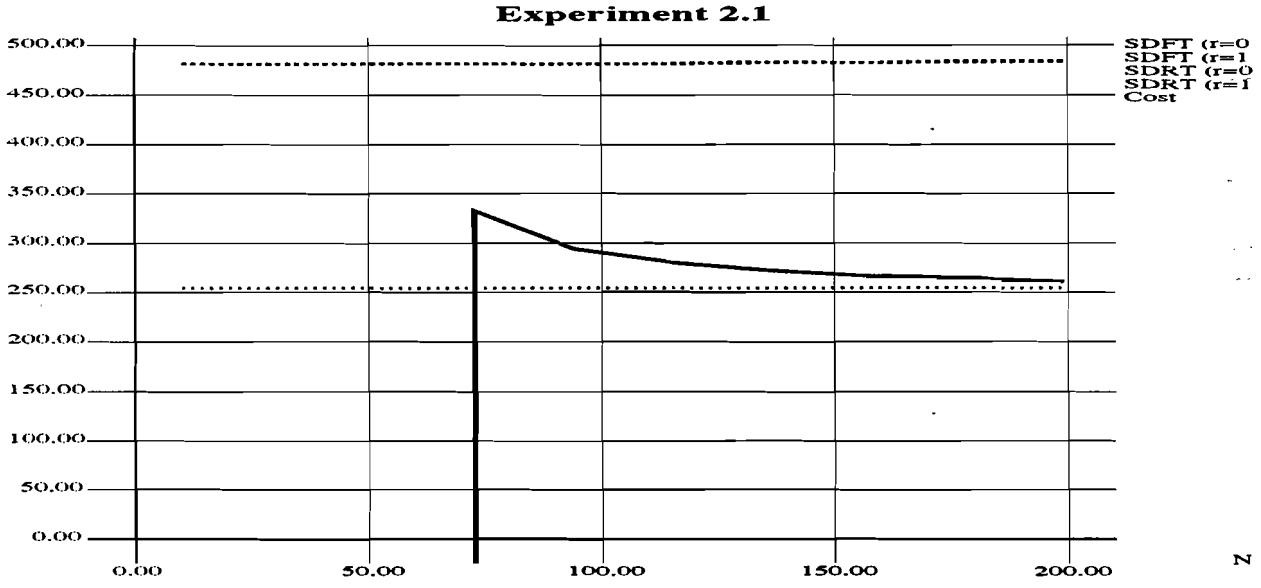

```
# File: "plotAll10104.txt".

#       Parameter          Cost          Cost          Cost          Cost
#             d          NLFT          NLRT          SDFT          SDRT
#
             20          1100           332           100           329
             28          1100           332           100           329
             36          1100           332           100           329
             44          1100           332           100           329
             52          1100           332           100           329
             60          1100           332           100           329
             68          1100           332           100           329
             76          1100           332           100           329
             84          1100           332           100           329
             92          1100           332           100           329
            100          1100           332           100           329
```




```
# File: "plotAll0105.txt".

#       Parameter          Cost          Cost          Cost          Cost
#               P          NLFT          NLRT          SDFT          SDRT
#
              20          1100           332            20           249
              28          1100           332            28           257
              36          1100           332            36           265
              44          1100           332            44           273
              52          1100           332            52           281
              60          1100           332            60           289
              68          1100           332            68           297
              76          1100           332            76           305
              84          1100           332            84           313
              92          1100           332            92           321
             100          1100           332           100           329
```

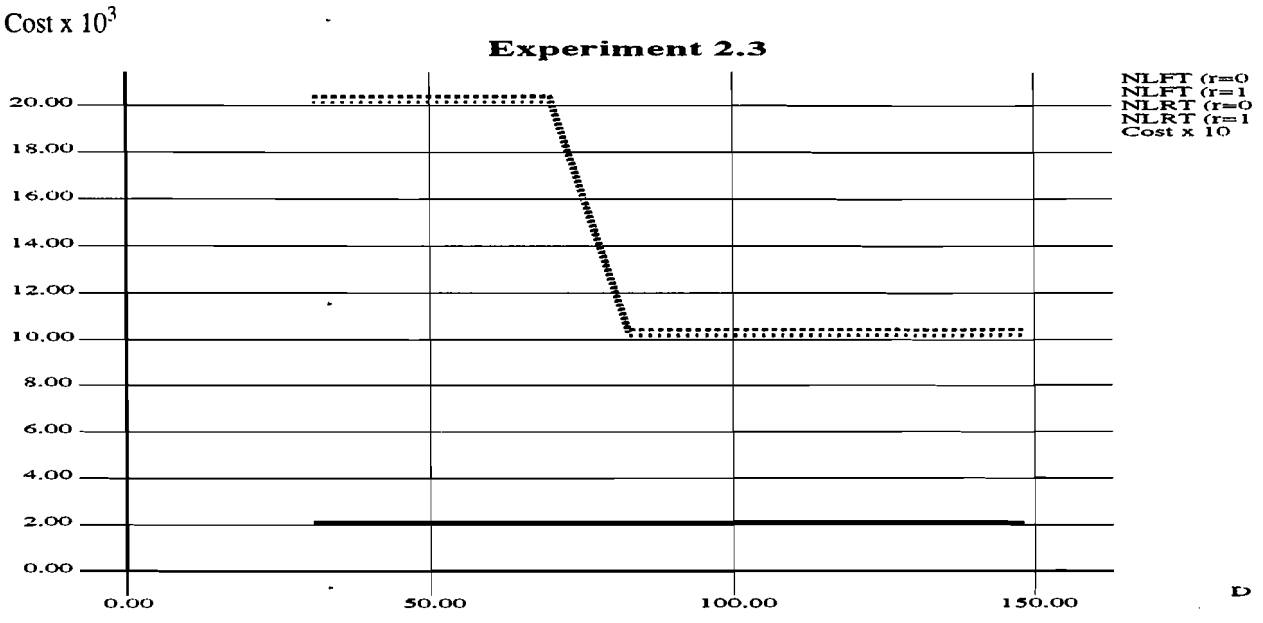
Cost x 10^3

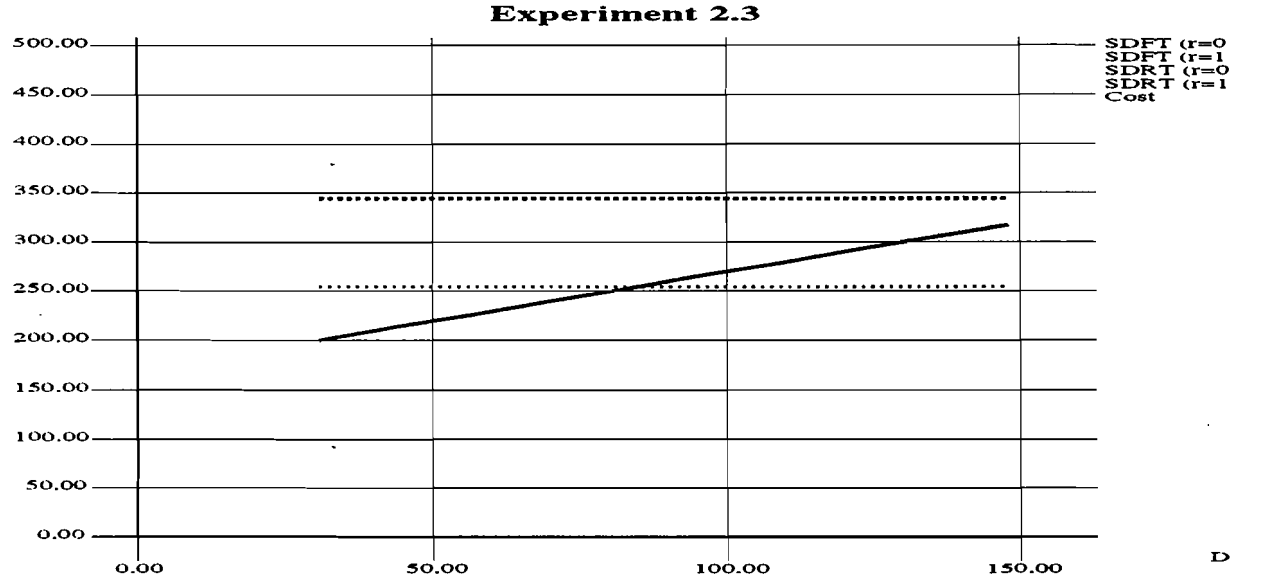





```
File name: "plotA110201.txt".
           Calculating costs of path: City.places_to_go.address.street.
           Changing parameter N for A(1): Name=places_to_go, type=Place_tg.
```

Parameter N	Cost NLFT r=0	Cost NLFT r=1	Cost NLRT r=0	Cost NLRT r=1	Cost SDFT r=0	Cost SDFT r=1	Cost SDRT r=0	Cost SDRT r=1
10	2100	2100	20197	20157	-2147483486	-2147483486	482	255
31	2100	2100	20218	20157	-2147483486	-2147483486	482	255
52	2100	2100	20239	20157	-2147483486	-2147483486	482	255
73	2100	2100	20260	20157	333	333	482	255
94	2100	2100	20281	20157	295	295	482	255
115	2100	2100	20302	20157	281	281	482	255
136	2100	2100	20323	20157	273	273	483	255
157	2100	2100	20344	20157	268	268	483	255
178	2100	2100	20365	20157	265	265	484	255
199	2100	2100	20386	20157	262	262	484	255

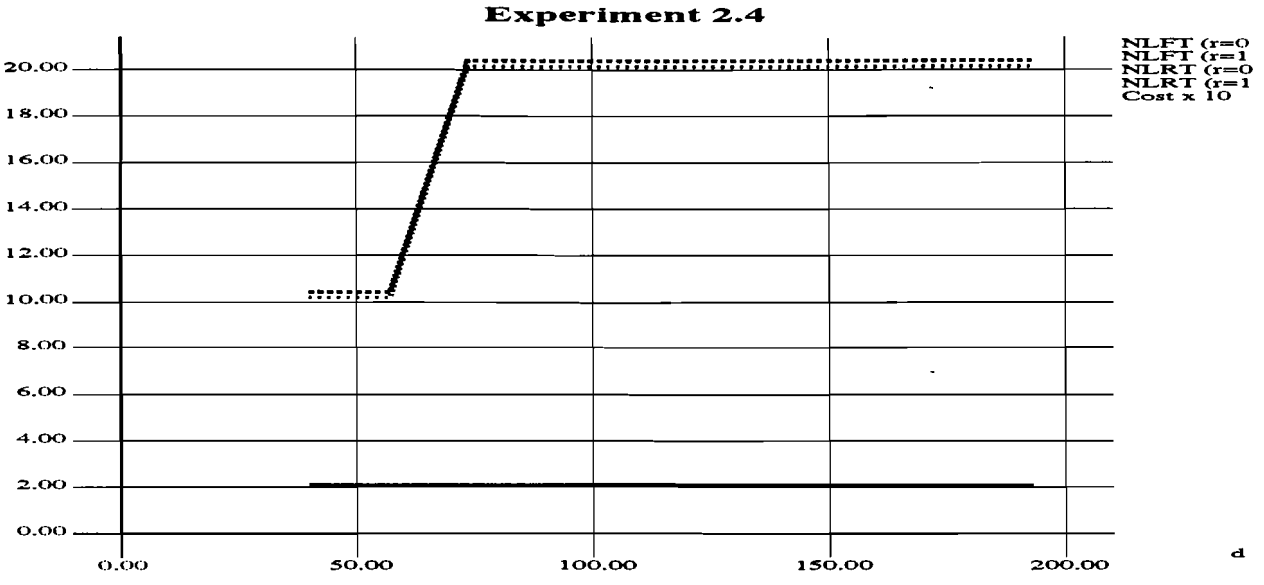


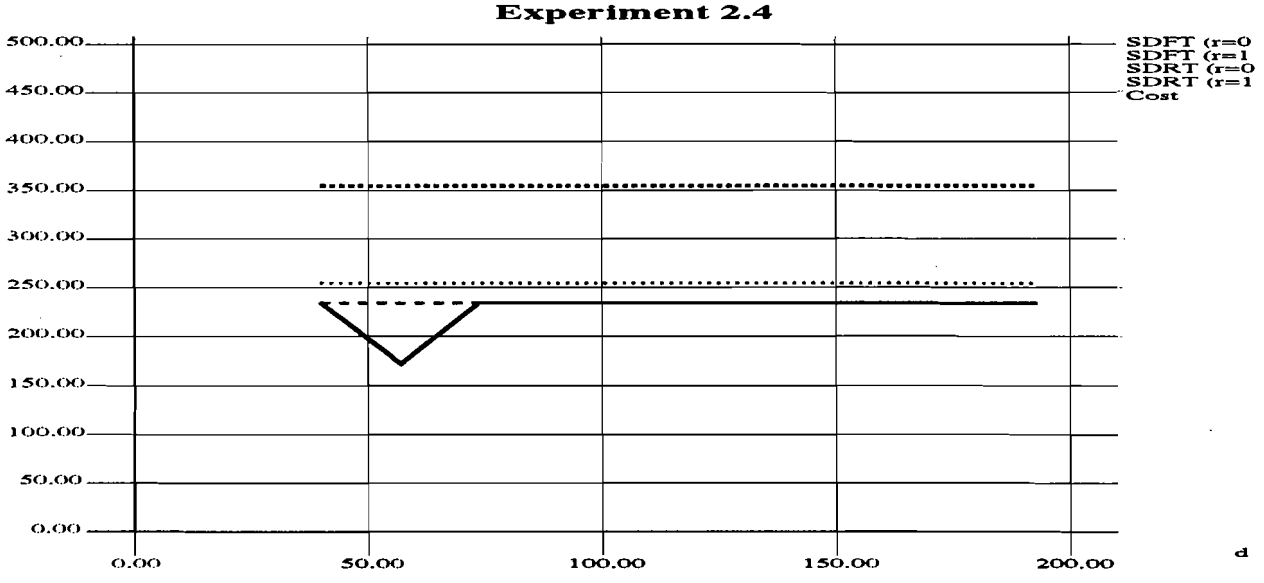



```
File name: "plotAll0203.txt".
           Calculating costs of path: City.places_to_go.address.street.
           Changing parameter D for type places_to_go.
```

Parameter D	Cost NLFT r=0	Cost NLFT r=1	Cost NLRT r=0	Cost NLRT r=1	Cost SDFT r=0	Cost SDFT r=1	Cost SDRT r=0	Cost SDRT r=1
31	2100	2100	20387	20157	201	201	345	255
44	2100	2100	20387	20157	214	214	345	255
57	2100	2100	20387	20157	227	227	345	255
70	2100	2100	20387	20157	240	240	345	255
83	2100	2100	10387	10156	253	253	345	255
96	2100	2100	10387	10156	266	266	345	255
109	2100	2100	10387	10156	279	279	345	255
122	2100	2100	10387	10156	292	292	345	255
135	2100	2100	10387	10156	305	305	345	255
148	2100	2100	10387	10156	318	318	345	255

Cost x 10^3



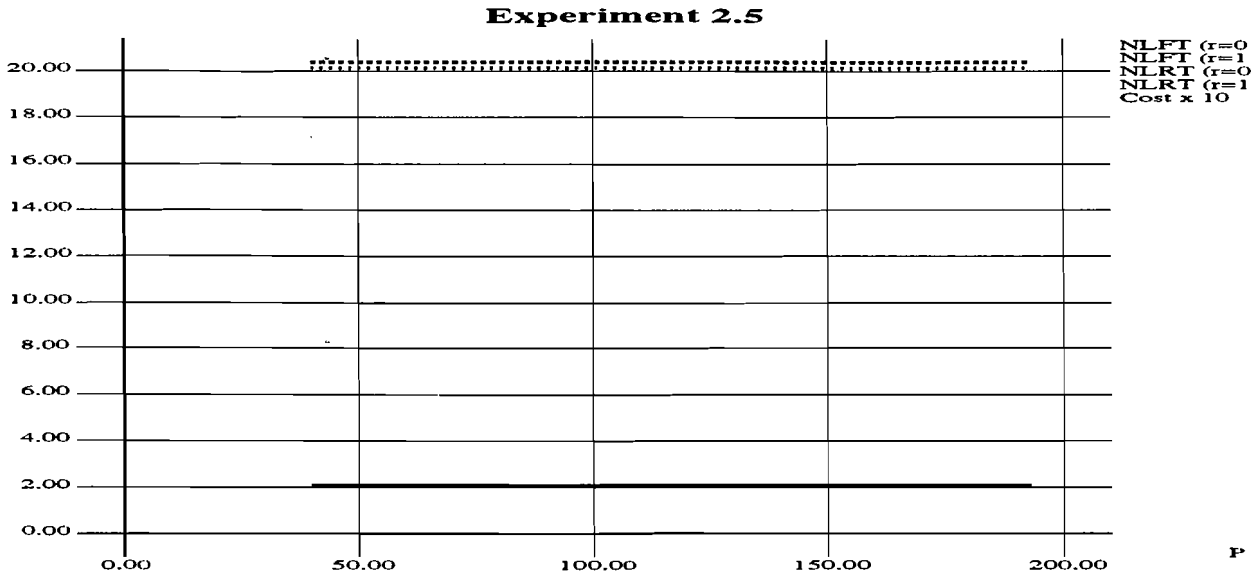


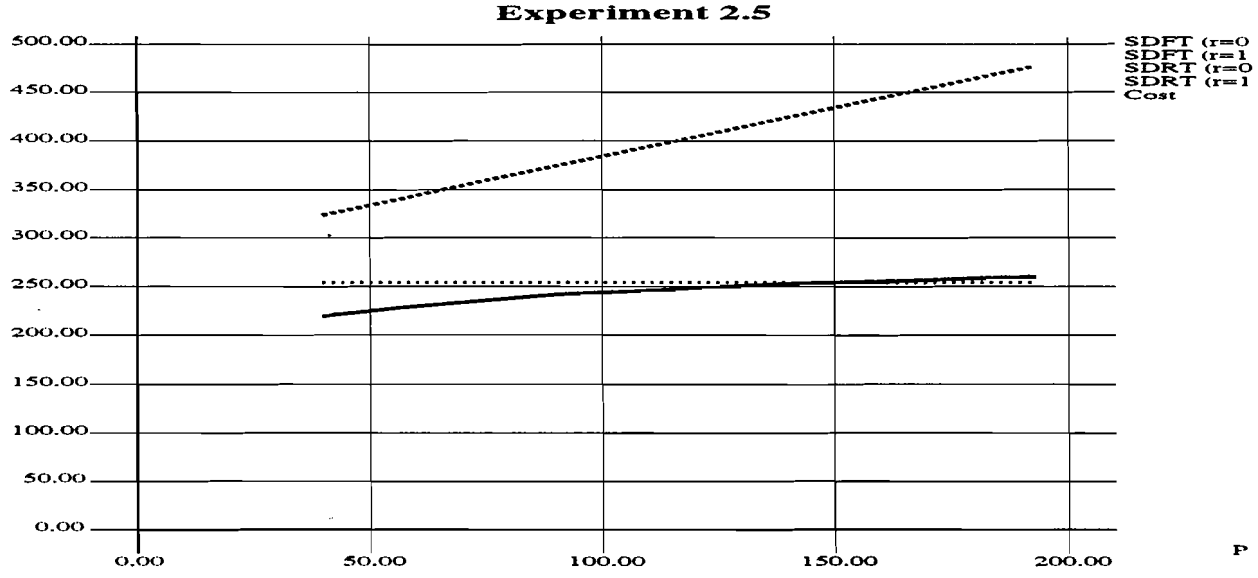

```
File name: "plot0204.txt".
           Calculating costs of path: City.places_to_go.address.street.
           Changing parameter d for type places_to_go.
```

Parameter d	Cost NLFT r=0	Cost NLFT r=1	Cost NLRT r=0	Cost NLRT r=1	Cost SDFT r=0	Cost SDFT r=1	Cost SDRT r=0	Cost SDRT r=1
40	2100	2100	10387	10156	235	235	355	255
57	2100	2100	10387	10156	172	235	355	255
74	2100	2100	20387	20157	235	235	355	255
91	2100	2100	20387	20157	235	235	355	255
108	2100	2100	20387	20157	235	235	355	255
125	2100	2100	20387	20157	235	235	355	255
142	2100	2100	20387	20157	235	235	355	255
159	2100	2100	20387	20157	235	235	355	255
176	2100	2100	20387	20157	235	235	355	255
193	2100	2100	20387	20157	235	235	355	255

```
File name: "plot0205.txt".
           Calculating costs of path: City.places_to_go.address.street.
```

Cost x 10^3





Changing parameter P for type places_to_go.

Parameter P	Cost NLFT r=0	Cost NLFT r=1	Cost NLRT r=0	Cost NLRT r=1	Cost SDFT r=0	Cost SDFT r=1	Cost SDRT r=0	Cost SDRT r=1
40	2100	2100	20387	20157	220	220	324	255
57	2100	2100	20387	20157	229	229	341	255
74	2100	2100	20387	20157	236	236	358	255
91	2100	2100	20387	20157	242	242	375	255
108	2100	2100	20387	20157	246	246	392	255
125	2100	2100	20387	20157	250	250	409	255
142	2100	2100	20387	20157	254	254	426	255
159	2100	2100	20387	20157	256	256	443	255
176	2100	2100	20387	20157	259	259	460	255
193	2100	2100	20387	20157	261	261	477	255

References

[BERT92] Bertino, E., Foscoli, P. An Analytical Model of Object-Oriented Query Costs. li, March 30 1992.

[BERT93] Bertino, E., Foscoli, P. On Modeling cost functions for object-oriented databases. January 20, 1993.

[MITCH91] An Architecture for Query Processing in Persistent Object Stores. Gail Mitchell, Stanley B. Zdonik, Umeshwar Dayal, June 1991.

[MITCH93] Extensible Query Processing in an Object-Oriented Database. Doctoral Thesis by Gail Mitchell April 1993; Chapter 7 titled Internal Query Representation.

[LO92] Query Tree Interface, Report on OODB Query Optimizer, by George C. Lo, February 1992.

[Yao77] Yao, S. B. Approximating block accesses in database organisations. ACM Comm. Vol 20, N. 4 (1977), 260-261.