BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M3

"Estimating Cadinalities of Sets in EPOQ"

by

Madhu Jalan

This research project by Madhu Jalan is accepted in its present form by the Department of Computer Science at Brown University in partial requirement of the requirements for the Degree of Master of Science.

March 1995

Date: 31 March 1995

Marian H. Nodine

Marian H. Nodine

Steven P. Reiss

Estimating Cardinalities of Sets in EPOQ

by

Madhu Jalan
Department of Computer Science
Brown University

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

March 1995

Estimating Cardinalities of Sets in EPOQ

Madhu Jalan

Abstract

Unlike relational databases, object oriented databases manipulate complex objects. Thus query optimization in object oriented databases is dependant on cost estimates of queries rather than built-in heuristics. An important part of any cost model is the technique used to evaluate the sizes of query sets. In this paper, we present various techniques to estimate the sizes of sets formed by the query operators of AQUA, which is the internal query algebra used in the query optimizer EPOQ.

1.0 Introduction

Relational Databases operate on simple uniform data. Because of the simple uniform structure of the data, query optimization is achieved by using built-in heuristics. The rules applied and the algorithms for searching strategies are all fixed, for a particular optimizer, irrespective of the data being manipulated.

Object Oriented Databases provide an environment for manipulating objects with an arbitrarily complex internal structure. These objects can be nested to any degree and also involve sharing of subobjects. Because of the complex nature of the data being manipulated, query optimization strategies cannot be fixed for a particular optimizer. They must consider the properties of the data being manipulated by the query before performing query rewrites. Query optimization is thus based on the cost of processing the query rather than built-in heuristics.

A very important component of an object-oriented query optimizer is the cost model that allows the optimizer to estimate the cost of different queries to determine the most efficient one. An important component of the cost model is a technique to estimate the cardinalities of sets that are formed as a result of a query or a subquery.

EPOQ is a query optimizer that uses a modular approach to query optimization. AQUA is the internal query algebra used by the optimizer. This paper describes the approach taken, and the actual implementation of the algorithms used by the cost model to estimate the sizes of sets in EPOQ.

Within the optimizer the query is maintained in the form of a tree. This tree is annotated with size and cost information at every stage. The sizes of sets are estimated starting at the bottom of the tree. This information is then propagated upwards. In this paper, we first give a description of the class for representation of size and cost information. We then discuss the size and cost information that is maintained in the Schema Manager. We then give a description of the various techniques used to evaluate the cardinalities of sets. After that we describe the internal representation of the size and cost annotations in the parse

tree and the various methods used to estimate the size of queries. After that we give a brief description of the database schema.

2.0 COInterval

The size and cost information is kept as an interval. We cannot get an exact estimate of the size and cost of a particular query. We give an upper bound and lower bound on the estimate of the size. Similarly we give an upper bound and lower bound on the estimate of the cost of execution of a query. We also keep information about the range of values that can be taken by a set (mset) of reals or integers. This is also kept as an interval. The class for the representation of the above information is called COInterval. The methods in the class are given below.

2.1 Class COInterval

This is the interval class for the representation of size, cost, range and sizesset (for tuple fields)

Public Methods

- **COInterval ();**

  Parameterless constructor.

- **COInterval (COInterval& s);**

  Copy Constructor.

- **COInterval (float upper, float lower);**

  Constructor.

- **~COInterval ();**

  Destructor.

- **void SetLower (float lower);**

  Sets the lower bound.

- **void SetUpper (float upper);**

  Sets the upper bound.

- **float Upper ();**

  Returns the upper bound of the interval.

- **float Lower ();**

  Returns the lower bound of the interval.

- **float Mean ();**

  Returns the mean value.

- **int operator == (COInterval& c);**

  Overloads the operator ==.

- **int operator <(COInterval& c);**

  Overloads the operator <.

- **int operator <= (COInterval& c);**

  Overloads the operator <=.

- **int operator >(COInterval& c);**

  Overloads the operator >.

- **int operator >= (COInterval& c);**

  Overloads the operator >=.

3.0 Schema Manager

The Schema Manager maintains information about the database which is used by the parser and the optimizer. The Schema Manager maintains two tables, one which contains type information and the other contains size and cost related information.

The first table contains information about the global types in the database. It maintains information about the user defined types and user defined tuples in the database. This table is know as the global types table.It maintains the name of the type being defined and a pointer to its structure.

The following table shows a sample of the global types table

TABLE 1. Table showing the Global Types table

Name	Type
Boolean	Boolean
String	String
Real	Real
Integer	Integer
Date	Date
Name	Name
Address	Address
TDate	Tuple : month: Integer day : Integer year : Integer
TAddress	Tuple : country : String zip_code: String city : String street : String number : Integer
TPerson	Tuple : name: Name birth_date : Date age: Integer address : Address
Money	Set [Integer]
Age	Set [Integer]
Set[Address]	Set [Address]

The second table contains cost and size information. For each element in the database, it maintains the following information.

1. **<Name>**
   Name of the set.
2. **<Type>**
   The type of the element.
3. **<Size [max, min]>**
   Estimate of the size of the object; kept as an interval. The size is maintained as an interval because we use statistical sampling to estimate the sizes of certain objects in the database. In this case, we give an estimate of the size which is in the form of an upper abound and a lower bound on the size of the set.
4. **<SKS>**
   The smallest known superset of the set (SKS). For example, we have a set, *Students*. The smallest known superset of this is the set of *People*. We maintain information about the set of *People* in addition to information about *Students*. This is because we use this information about supersets of sets to estimate the sizes of query sets.
5. **<Choose>**
   The *choose* record for complex objects and nested sets. The AQUA operator *choose* takes a set and returns a random element of it. Now for a simple set, the size of the element returned is *1*. But if we are choosing an element from a set of sets, the size of the set chosen is not necessarily *1*. Similarly, this chosen set might have it's own SKS, range and other related information. Thus for nested sets and complex objects we maintain information about the *choose* record of the set.
6. **<Range [max, min]>**
   The range of values that can be taken by integers, reals, sets (msets) of integers or sets (msets) of reals.
7. **<SizeSSet [max, min]>**
   Information about the number of distinct elements in the set, for each tuple field; maintained as an interval. For sets of tuples, we maintain information about each field of the tuple. For each field of the tuple, we need to know the number of distinct objects of that type that are there in the orignal tuple set. This information is kept in addition to information about smallest known supersets as it gives a better bound on the number of distinct objects in the set, for each tuple field, than the size of the SKS.

The Schema Manager also keeps information about image sets. Image sets are sets of objects that are referred to by another set. For example, the image set *Student.age* is the set of all age objects that are referred to by person objects in the collection named *Students*. The Schema Manager keeps one level of information. Thus the table containing one level of information, keeps information about sets of the form $S.m_1$ in addition to information about named sets.

The Schema Manager also maintains information that can be obtained through statistical sampling of the database. It maintains information about the *intersection* of sets of the same type. The supertype of the set of *Employees* and *Students* is *Person*. The Schema Manager would thus maintain information about the size of the intersection of the two sets *Employees* and *Students*.

The database also maintains information about the *flatten* record for nested sets and complex objects. This information assists us in making estimates of image sets resulting from flattening a nested set into a non-nested set.

The following table shows the Global Symbols Table.

TABLE 2. Table showing the Global Symbols Table

No	Name	Type	Size	Range	SizeSSet	SKS	Choose
1	People	Set[Person]	[1000, 1000]	-	-	-	-
2	Addresses	Set[Address]	[500, 500]	-	-	-	-
3	Students	Set[Students]	[200, 200]	-	-	1	-
4	Clients	Set[Client]	[300, 100]	-	-	1	-
5	Ages	Set[Age]	[74, 74]	[2, 76]	-	-	-
6	People.address	Set[Address]	[350, 350]	-	-	2	-
7	People.age	Set[Age]	[62, 62]	[10, 72]	-	5	-
8	Students.age	Set[Age]	[10, 35]	[25, 25]	-	7	-
9	Families	Set[Family]	[202, 202]	-	-	-	13
10	Families.address	Set[Set[Address]]	[202, 202]	-	-	-	14
11	Flatten(Families)	Set[People]	[684, 684]	-	-	1	-
12	Flatten(Families).address	Set[Address]	[202, 200]	-	-	2	-
13	Choose(Families)	Set[People]	[3.39,3.39]	-	-	11	-
14	Choose(Families).address	Set[Address]	[1.6, 1.6]	-	-	12	-
15	TPeople	Set[TPerson]	[900, 900]	-	-	-	16
16	Choose(TPeople)	TPerson	[1, 1]	-	-	-	-
17	Choose(TPeople):name	String	[1, 1]	-	[850, 850]	-	-
18	Choose(TPeople):birtdate	TDate	[1, 1]	-	[360, 360]	22	-
19	Choose(TPeople):age	Integer	[1, 1]	[2, 76]	[74, 74]	5	-
20	Choose(TPeople):address	Address	[1,1]	-	[325, 325]	2	-
21	TDates	Set[TDate]	[365, 365]	-	-	-	22
22	Choose(TDates)	TDate	[1, 1]	-	-	-	-
23	Choose(TDates): month	Integer	[1, 1]	[1, 12]	[12, 12]	-	-

TABLE 2. Table showing the Global Symbols Table

No	Name	Type	Size	Range	SizeSSet	SKS	Choose
24	Choose(TDates):day	Integer	[1, 1]	[1, 31]	[31, 31]	-	-
25	Choose(TDates): year	Integer	[1, 1]	[0, 99]	[100, 100]	-	-
26	Intersect(Clients, Students)	Set[Person]	[20, 20]	-	-	1	-

3.1 Schema Manager Classes and Methods

The class SchemaMgr contains information on the database schema for the optimizer. It SchemaManager contains two elements, the first is the global symbols table of type SMGlobalSymbols, and the other the global types table of type SMGlobalTypes. The class SMGlobalSymbols represents a table of elements of type SMGlobalSymElt. This table maintains the information related to the size and cost of queries. The class SMGlobal-Types represents a table of elements of type PASymtabElt. This table maintains information about the user defined types in the database.

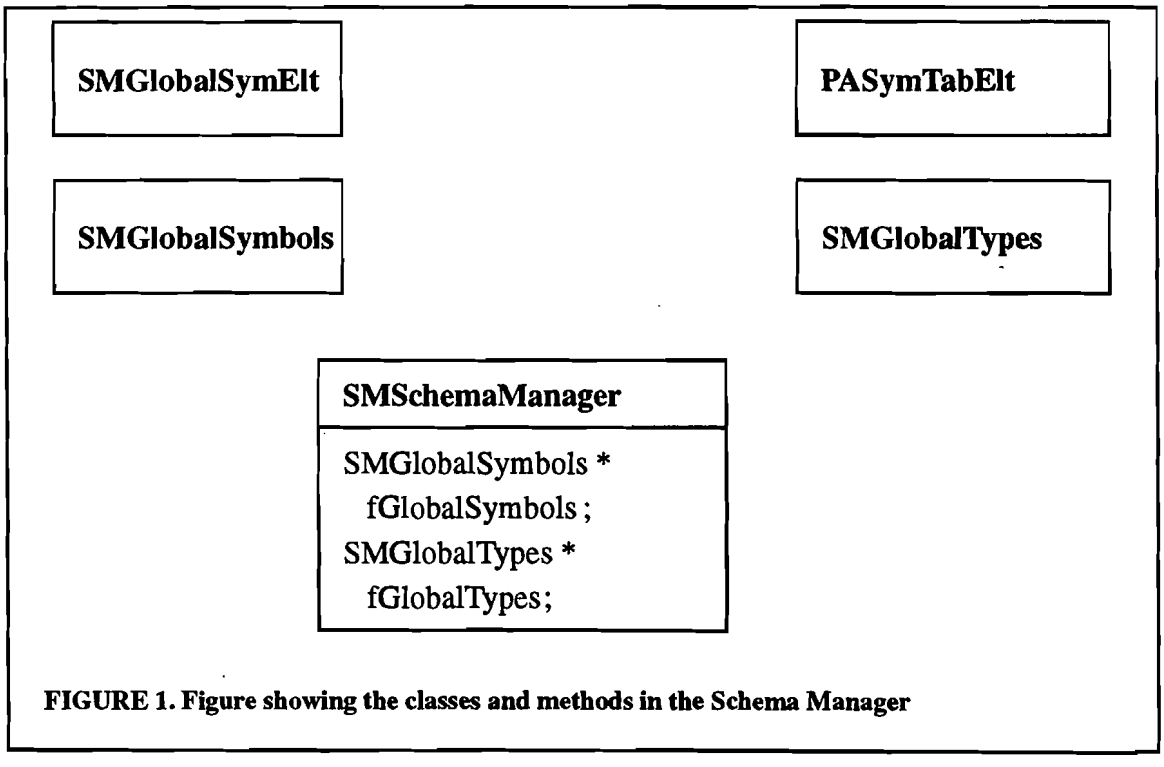

FIGURE 1. Figure showing the classes and methods in the Schema Manager

3.1.1 class SMSchemaMgr

The methods in the class SMSchemaMgr are as follows.

Public Methods

- **void SMSchemaMgr (char * type_file, char * symbol_file, char * dtypes_file, char * tuple_file);**

  Reads the type_file, the symbol_file, the tuple_file, the dtypes_file and creates a schema manager with the information from the four files in it. The symbol_file contains information about the global symbols in the database. The type_file contains information about user defined types in the database. The tuple_file contains information about the user defined tuples in the database. The dtypes_file contains information about the types derived from the global types in the database.

- **~SMSchemaMgr ();**

  Destructor.

- **Error AddUserTypes (char * type_file);**

  Adds the types specified in the type_file to the types table.

- **Error AddTupleTypes (char * tuple_file);**

  Adds the tuple types specified in the tuple_file to the types table.

- **Error AddDerivedTypes (char * dtypes_file);**

  Adds the types derived from the global user types to the global types table.

- **Error TYTypeData * LookupType (char * representation);**

  Looks up the type in the types table.

- **TYTypeData * LookupMethod (char * class_name, char * method_name);**

  Looks up user-defined type in the types table.

- **Error AddUserGlobals (char * globals_file);**

  Adds the globals specified in the types table to the symbols table.

- **TYTypeData * Lookup (char * name);**

  Looks up the name in the global symbols table and returns the associated type.

- **SMGlobalSymElt * LookupRecord (char * name);**

  Looks up the name in the symbols table and returns the associated extent record.

- **COInterval * LookupSize (char * name);**

  Looks up the symbol in the global symbols table and returns the size.

- **char * LookupSKS (char * name);**

  Looks up the name in the symbols table and returns the name of its smallest known superset (SKS).

- **COInterval * LookupRange (char * name);**

  Looks up the name in the global symbols table and returns the range of values that the object can take if it is a real or an integer or a set (mset) of reals or a set (mset) of integers.

- **COInterval * LookupSizeSSet (char * name);**

  Looks up the name of the object and returns the of size of the superset if it is the name of a tuple field. Returns NULL otherwise.

- **SMGlobalSymElt * LookupSKSRecord (char * name);**

  Looks up the name in the global symbols table and returns a pointer to its SKS record.

- **SMGlobalSymElt * LookupChoose (char * name);**

  Looks up the named symbol in the global symbols table and returns a pointer to its Choose record.

- **SMGlobalSymElt * LookupTupField (char * name, char * label);**

  Looks up the field of the named set of tuples and returns a pointer to it; returns NULL if the record does not exist.

- **SMGlobalSymElt * LookupIntersect (char * name1, char * name2);**

  Looks up the Schema Manager to see if a record exists for the *intersection* of the two sets and returns a pointer to it; returns NULL if the record does not exist.

- **int NumGlobalSyms ();**

  Returns the number of elements in the Global Symbols table.

- **SMGlobalSymElt * ithGlobalSymbol (int i);**

  Returns a pointer to the i^{th} record in the Global Symbols table. Counting starts at 0. Returns NULL if the record does not exist.

- **void PrintSchemaMgr ();**

  Prints the types table and the global symbols table.

3.1.2 class SMGlobalSymElt

The global symbol element class represents an element in the symbol table. It contains information about the name of the object, size, the range of values it can take (if its a set of integers or reals), name of its smallest known superset (SKS), a pointer to its SKS record and a pointer to its choose record. For each tuple field, it also maintains the number of unique field elements that are there in the set.

Public Methods

- **SMGlobalElt ()**

  Parameterless constructor.

- **SMGlobalSymElt (char * name, TYTypeData * type, COInterval * size, char * SKS, COInterval * range, COInterval * sizeSSet, SMGlobalSymElt * SKSrecord, SMGlobalSymElt * choose);**

  Constructor.

- **~SMGlobalSymElt ()**

  Destructor.

- **char * Name ();**

  The name of the symbol.

- **TYTypeData * Type ();**

  Returns the pointer to the associated type of the set.

- **COInterval * Size ();**

  Returns the size of the element.

- **char * SKS ();**

  Returns the name of the smallest known superset of the symbol.

- **COInterval * Range ();**

  Returns the range of values that the elements of the set can take.

- **COInterval * SizeSSet ();**

  Returns the size of the superset.

- **SMGlobalSymElt * SKSRecord ();**

  Returns the SKS record of the symbol.

- **SMGlobalSymElt * Choose ();**

  Returns the choose record of the symbol.

- **SetName (char * name);**

  Sets the name of the symbol.

- **SetType (TYTypeData * type);**

  Sets the type of the symbol.

- **SetSize (COInterval * size);**

  Sets the size of the symbol.

- **SetSKS (char * SKS);**

  Sets the name of the smallest known superset of the symbol.

- **SetRange (COInterval * range);**

  Sets the range of the symbol.

- **SetSizeSSet (COInterval * sizeSSet);**

  Sets the size of the superset.

- **SetSKSRecord (SMGlobalSymElt * SKSrecord);**

  Sets the pointer to the SKS record of the symbol.

- **SetChoose (SMGlobalSymElt * choose);**

  Sets the pointer to the choose record of the symbol.

3.1.3 SMGlobalSymbols

This class describes the global symbols table. It contains information about the sets in the database, the size of the set, name of its smallest known superset, pointer to its SKS record, a pointer to its choose record and the range of values it can take if its a set of reals or integers. For each tuple field, it also maintains information about the number of unique field elements that are there in the set.

Public Methods

- **SMGlobalSymbols ();**

  Parameterless constructor.

- **~ SMGlobalSymbols ();**

  Destructor.

- **SMGlobalSymElt * Lookup (char * name);**

  Looks up the name of the symbol in the global symbols table and returns a pointer to its record; returns NULL pointer if the symbol does not exist.

- **TYTypeData * LookupType (char * name);**

  Looks up the name in the table and returns a pointer to its associated type; returns error if the symbol does not exist in the table.

- **COInterval * LookupSize (char * name);**

  Looks up the name in the global symbols table and returns the size of the set; returns error if the symbol does not exist in the table.

- **COInterval * LookupRange (char * name);**

  Looks up the symbol in the table and returns the range of values that the object can take if it is a real or an integer or a set (mset) of reals (integers); returns an error if the symbol does not exist in the table.

- **COInterval * LookupSizeSSet (char * name);**

  Looks up the symbol in the table and returns the size of the superset if it is a tuple field; returns an error if the symbol does not exist.

- **char * LookupSKS (char * name);**

  Looks up the symbol in the table and returns the name of its SKS.

- **SMGlobalSymElt * LookupSKSRecord (char * name);**

  Looks up the symbol in the global symbols table and returns a pointer to its SKS record; returns an error if the symbol does not exist in the global symbols table.

- **SMGlobalSymElt * LookupChoose (char * name);**

  Looks up the symbol in the global symbols table and returns a pointer to its choose record; returns an error if the symbol does not exist in the global symbols table.

- **SMGlobalSymElt * LookupTupField (char * name, char * label);**

  Looks up the field of the named set of tuples in the table and returns a pointer to it; returns NULL if it is not found.

- **Error AddGlobal (char * name, COInterval * size, char * SKS, COInterval * range, COInterval * sizeSSet);**

  Adds the new symbol to the table after checking to ensure that the SKS of the record already exists in the table. If the symbol is the choose record for a set, then it checks to see if the symbol already exists in the table. If the two conditions are not satisfied, the record is not added to the table and an error message is returned.

- **int NumSyms ()**

  Returns the number of records in the table.

- **SMGlobalSymElt * ithGlobal (int i);**

  Returns a pointer to the i^{th} global symbol in the table; returns NULL if the symbol does not exist.

- **PrintGlobalSymbols ();**

  Prints the global symbols table.

3.1.4 class PASymtabElt

This is the global type class. This represents an element in the global symbols table. It contains the name of the user defined type and a pointer to it's type.

- **PASymtabElt (char * name, TYTypeData * t);**

  Constructor.

- **PASymtabElt ();**

  Parameterless constructor.

- **~PASymtabElt ();**

  Destructor.

- **char * Name ();**

  The name of the global type.

- **TYTypeData * Type ();**

  Returns a pointer to the type of the named global type.

3.1.5 SMGlobalTypes

This is the class describing the global types table which contains information about the user-defined types in the database.

Public Methods

- **SMGlobalTypes ();**

  Parameterless constructor.

- **~ SMGlobalTypes ()**

  Destructor.

- **TYTypeData * Lookup (char * name);**

  Looks up the symbol in the types table and returns a pointer to its type.

- **void PrintGlobalTypes ()**

  Prints the global types table.

4.0 Estimating the Size of Sets

Within the optimizer, the query is represented as an Extensible Annotated Tree (EAT). An EAT is composed of alternating layers of data nodes and function nodes connected by labelled arcs.

Data nodes represent the data that are being manipulated during the execution of a query. Such a node can represent either an object in the database (if it is a leaf node in the EAT); or an object built by a query, subquery, or other function (if it is an internal node in the EAT). The root of the EAT is a data node representing the result of the query.

Function nodes represent the actions that can be taken on the data. Child nodes represent the inputs to the function. Any function node has at least one child node. A function node also has one parent node, representing its output.

Arcs in the EAT represent the relationship between functions and data. An arc always connects a function node with a data node. Arcs are annotated with information on the availability and use of variables defined in the lambda expressions contained in the query.

The cost of the execution of a query and the size of the resultant set are calculated by function nodes. Data nodes are annotated with size and cost information, which is used during query optimization.

The information required to estimate the sizes of query sets is maintained in an extent table in the Schema Manager. This information in the Schema Manager is restricted to information about constant sets in the database, image sets and information that can be inferred using statistical sampling.

After a query is parsed, type checking is done. Then the size of the data sets, obtained as a result of the execution of a query, a subquery or other function, is calculated bottom up. The extent record for the leaf nodes is obtained from the Schema Manager. The extent records for the data sets that are the result of some query or subquery are then calculated starting at the bottom of the tree. The result is then propagated upwards. This section describes the methods used to estimate the sizes of data sets depending on the query operator.

The extent record of the data node is maintained in the form of a table which contains the following information about the object.

1. **<Name>**
   The name of the object.
2. **<Size [max, min]>**
   An estimate of the size of the set, which is kept as an interval. The size is maintained as an interval because we cannot calculate an exact estimate of the size. We thus give an upper bound and a lower bound on the size of the estimated set.
   The operators *forall*, *exists* and *mem* return a boolean. They are often used as a predicate for *select*. Thus for these operators we estimate size as a fraction between *0* and *1* to use as a probability that the operator evaluates to True.
3. **<Cost [max, min]>**
   An estimate of the cost; which is kept as an interval. The cost of executing a query is dependant on the size of the query set. Since we cannot calculate the exact size of a set, we cannot calculate the exact cost of a query. We thus give an upper bound and a lower bound on the estimated cost of a query.
4. **<Range [max, min]>**
   The range of values an object can take if it is of type real or integer, or if it is a set (mset) of reals or integers.This gives an upper bound and a lower bound on the values that can be taken by the object.
5. **<SKS>**
   Information about its smallest known superset (SKS).
6. **<Choose>**
   Information about the sets *choose* record if it is a complex object or a nested set or a set of tuples.
7. **<SizeSSet [max, min]>**
   Information about the number of distinct elements in the set, for each tuple field; maintained as an interval. For sets of tuples, we maintain information about each field of the tuple. For each field of the tuple, we need to know the number of distinct objects of that type that are there in the set.

4.1 Set and Multiset Operators in AQUA

The following is a list of AQUA operators that operate on sets (multisets) and the techniques used to estimate the sizes of the sets formed as a result of the query.

Guideline : When in doubt make estimate err on the side of being too big.

4.1.1 Operator Apply

The *apply* operator takes a set and applies a function to each element of the set to get a set of objects of a new type. The set might be a flat set or a nested set.

Consider the query, Q_1 = ***apply*** *(lambda (s) s.address.city) (People)*
In the above query the function *s.address.city* is applied to each member of the set *People* to return the set of all cities in which students live.

As given in [1], Cherniack gives an estimation technique for the AQUA operator, *apply*, which is based on the assumption that for any function f and subset of f's domain D, if $Image_f(D)$ is the image of f under D, then

$$\frac{|D|}{|Image_f D|} \approx \frac{|SKS(D)|}{|Image_f(SKS\langle D\rangle)|}$$

This assumption is used to infer the size of the set *People.address.city* as follows

$$\frac{|People \cdot address|}{|People \cdot address \cdot city|} = \frac{|Addresses|}{|Addresses \cdot city|}$$

Thus solving the above equation, we get,

$$|People \cdot address \cdot city| = \left(\frac{|Addresses \cdot city|}{|Addresses|}\right) \cdot |People \cdot address|$$

In general, for any set $s = S.m_1.m_2.....m_n$

$$s \cdot size = \frac{SKS(s) \cdot size \cdot \left(S \cdot m_1 \dot{\ldots} \cdot m_n\right) \cdot size}{SKS(S \cdot m_1 \cdot \ldots \cdot m_n) \cdot size}$$

The size of the sets, where the function is a path expression, is calculated using the above estimation technique.

The fully annotated parse tree for the above query is given in Figure 1 below.

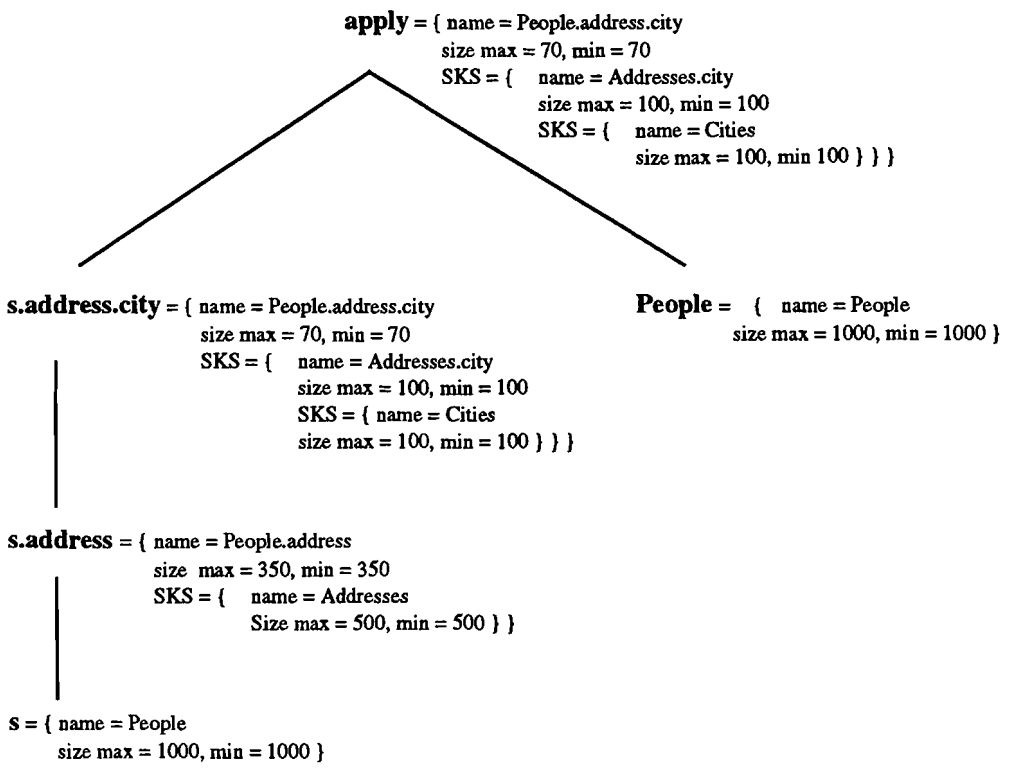

FIGURE 2. Shows the fully annotated parse tree for Q_1

The EAT for the apply operator has two data nodes. The first called the input node, and the second called the other node because it does not represent data that will be processed, but instead represents how the input data is processed. The input node is the data node representing the input data set. The other node represents the function applied to the input data set. The size and the extent record of the output set can be determined as follows.

1. If the name of the data set represented by the other node if of the form *Choose(Set).* Then the size of the resultant data set is that of *Set* or a parent of *Set.*

   For example, consider the query,

   Q_2 = ***flatten*** *(**apply** (lambda (f)* ***apply*** *(lambda (p) p.address) (f)) (Families))*

   Here *Families* has type *Set[Set[Person]].* In the above query we are trying to get the set of addresses at which each member of any family lives. As can be seen in Figure 2, the name of the other node for the outermost *apply* is of the form *Choose(Families).address.* This is the average size of the set of addresses for any average family. From this using the methods described above we estimate the size of the resultant data set which is the set of sets of addresses at which each member of each family lives. This set is then flattened to get the required set of addresses.

   The fully annotated parse tree for the above tree is given in Figure 2 below.

2. If the name of the data set represented by the other node is not of the above form, then the size of the resultant data set is that of the other node. This is because the resultant set is the output of the given function, *f*, applied to each member of the original input data set. This is illustrated in Figure 1 given above.

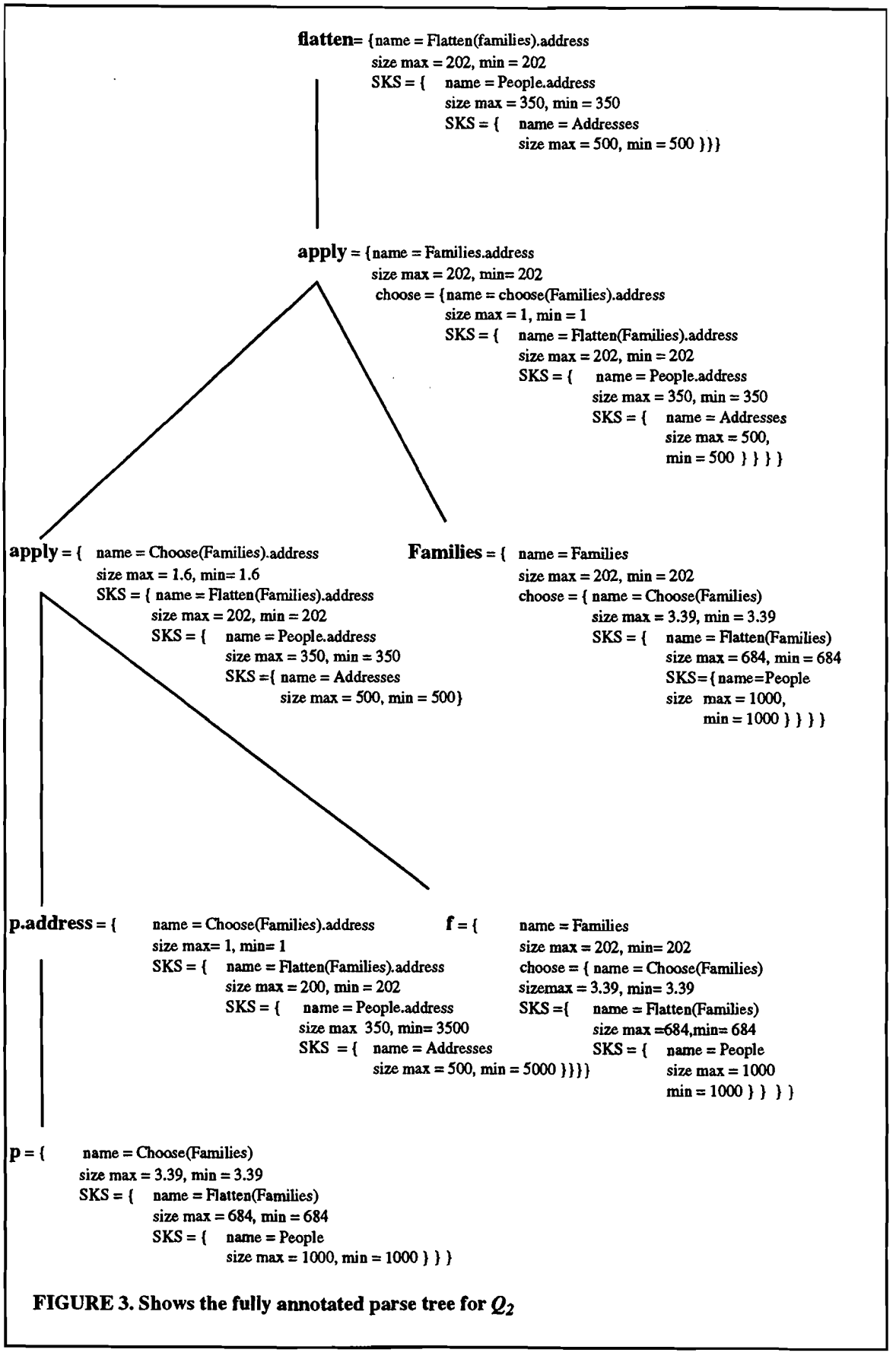

FIGURE 3. Shows the fully annotated parse tree for Q_2

4.1.2 Arithmetic Operators

The operators "+", "-", "/" and "*" take in two integers (reals) and return the result of applying one of the above operators.

The function node for arithmetic operators have two input nodes. The two input nodes might be integers or reals. The resultant size is *1*.

4.1.3 Operator Not

The query operator, *not*, is similar to the set operator complement.

Let the size of the input set be *Size.of.S*. The size of *S* is between *0* and *1*. Then the size of the resultant set is *(1 - Size.of.S)*.

4.1.4 Operator Choose

Choose, takes in one input set and returns a random element of the set. This is used for nested sets, to give representative information on the elements on the elements of the set (that are also sets).

For input sets that are sets of sets, the Schema Manager maintains a *choose* record. If the extent record for the input set has a *choose* record, then the size of the resultant set is that of the *choose* record; otherwise the size is assumed to be *1*.

4.1.5 Operator Intersection

The AQUA operator, *intersection*, is similar to the set operator, intersection.

Records are maintained in the Schema Manager for *intersection* the same types. But this record is maintained when an *equality* is defined as an argument and not as an *equivalence*. The EAT for the *intersection* operator has two input data nodes and two other data nodes.

Let the two input sets be S_1 and S_2 and the resultant set be R. The nodes for the lambda variables are annotated with information about the superset of S_1 and S_2. The size of the resultant set can be calculated as follows.

3. $S_1 = S_2$, then the $Size.of.R = Size.of.S_1$
4. S_1 is a superset of S_2, then $Size.of.R = Size.of.S_1$
5. S_2 is a superset of S_1, then $Size.of.R = Size.of.S_2$
6. If there exists a record for $Intersect\ (S_1, S_2)$ in the Schema Manager,
   then $Size.of.R = Size.of.Intersect\ (S_1, S_2)$

7. If there exists an intersect record for supersets of S_1 and S_2. Let the superset of S_1 and S_2 be SS_1 and SS_2 respectively.
   Let *Size.of.M = min (Size.of.S_1, Size.of.S_2)* and *SS = superset (M)*,

$$Size \cdot of \cdot R = \frac{Size \cdot of \cdot Intersect(SS_1, SS_2)}{SS} \times Size \cdot of \cdot M$$

8. Otherwise, *Size.of.R = min (Size.of.S_1, Size.of.S_2)* * DEFAULT_INT_FRACTION
   It is assumed that a certain fraction of the objects in the two sets are equal. DEFAULT_INT_FRACTION is taken as this default value.

Consider the query, Q_3 = ***intersection*** *(=, Person) (Students, People)*
The fully annotated parse tree for the above query is given below.

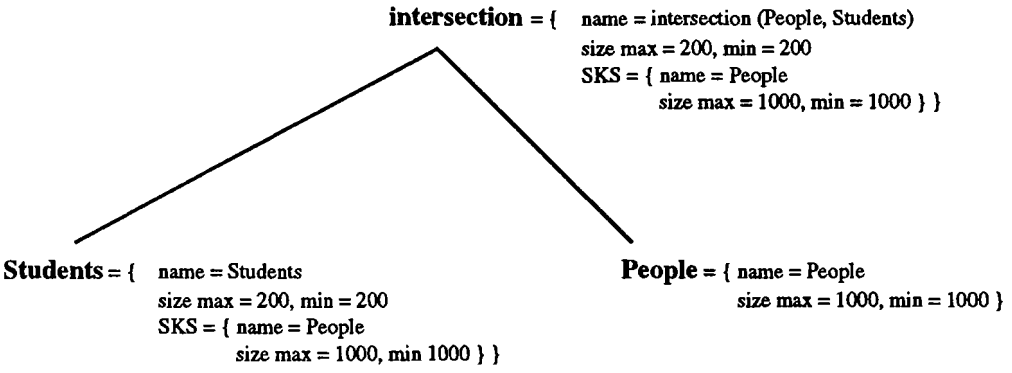

FIGURE 4. Shows the fully annotated parse tree for Q_3

4.1.6 Operator Difference

The *difference* operator is similar to the set difference operator. But thus record is maintained when an *equality* is defined as an argument and not as an *equivalence*.The EAT for the *difference* operator has two input data nodes and two other data nodes.

Let the two input sets be S_1 and S_2. The nodes for the lambda variables are annotated with information about the superset of S_1 and S_2. Then the size of the *difference* is
Size.of.Difference (S_1, S_2) = Size.of.S_1 - Size.of.Intersect (S_1, S_2)
Size.of.Intersect (S1, S2) can be calculated using techniques used to calculate the *intersection* of sets (Section 4.1.5).

Let us consider the query,
Q_4 = ***diff*** *(=, Person) (People, Students)*
The size of the *intersection* of the sets *Students* and *People* is calculated using the methods for calculating the *intersection* of sets (look at operator *Intersection,* Section 4.1.5). This is then used to calculate the size of the above set.

The fully annotated parse tree for the above query is given in Figure 3.

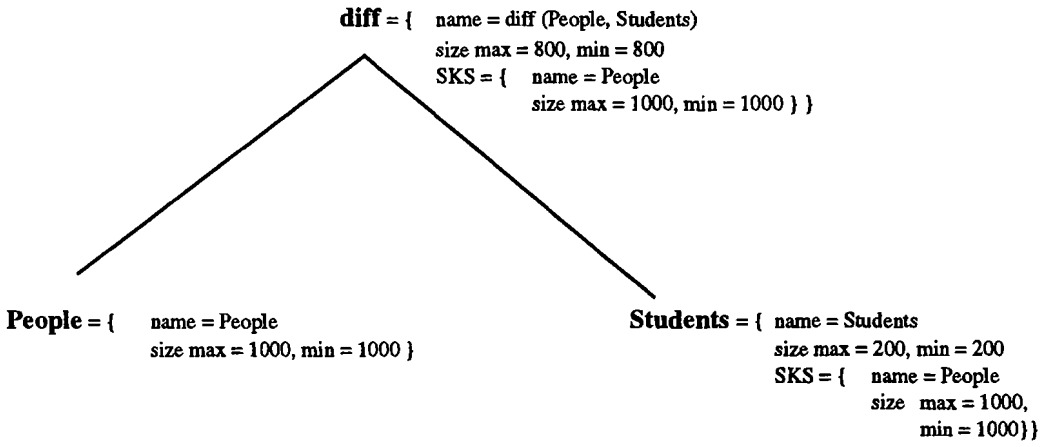

FIGURE 5. Shows the fully annotated parse tree for Q_4.

4.1.7 Operator Union

The *union* operator is similar to the set union operator. But thus record is maintained when an *equality* is defined as an argument and not as an *equivalence*. The EAT for the *union* operator has two input data nodes and two other data nodes.

Let the two *input* sets be S_1 and S_2. The nodes for the lambda variables are annotated with information about the superset of S_1 and S_2. Then the size of the union of the two sets is
$Size.of.Union\ (S_1, S_2) = Size.of.S_1 + Size.of.S_2 - Size.of.Intersect\ (S_1, S_2)$
where *Size.of.Intersect (S1, S2)* can be calculated using the techniques used to calculate the *intersection* of sets (Section 4.1.5).

Let us consider the query,
Q_5 = ***union*** *(=, People) (Students, People)*
The size of the *intersection* of the sets *Students* and *People* is calculated using the methods for calculating the *intersection* of sets (similar to the query operator *Intersection,* Section 4.1.5). It is then used to calculate the size of the above set.

4.1.8 Operator MSet

The operator takes the input element and converts it into a multiset.

The *mset* queries can be of two types:

1. Q_6 = ***mset*** *(E)*, where *E* is the input element.
   In this case, the size of the resultant set is *1*.
2. Q_7 = ***apply*** *(lambda (x)* ***mset*** *(x)) (S)*
   In this case, the operator *mset* is applied to each member of the input set (mset) *S*. The size of the resultant set is thus equal to the size of the original set (mset) *S*.

4.1.9 Operator Set

The operator takes the input element and converts it into a set.

The *set* queries can be of two types:

1. Q_8 = ***set (E)***, where *E* is the input element.
   In this case, the size of the resultant set is *1*.
2. Q_9 =***apply (lambda (x) set (x)) (S)***
   In this case the operator set is being applied to each member of the input set or multiset. If the input object *S*, is a set the size returned is equal to the size of the original set as there are no duplicates in the input set. If *S* is a multiset, the set operator is supposed to eliminate duplicates. But we have no means of eliminating duplicates from multisets yet and so the size of the set returned is equal to the size of the original multiset.

4.1.10 Operator Select

Select returns all the elements of a set that satisfy the given predicate. The predicates can be of two types. The first kind include predicates that compare field values with constants. The second kind allow field values to be compared with one another. We are dealing with predicates of the first kind only. Also, when we are comparing field values with constants, we are dealing with field values and constants of type integer only. We are not dealing with reals and string yet.

Selectivity is an estimate of the fraction of the input set that is present in the resultant set. In determining the selectivity we assume that each attribute used in the predicate is evenly distributed over a fixed domain.

Select queries are of the form ***select (Pred) (Set)***. The size of the resultant set, *R*, is given by *R.size = (Pred.sel * Size.of.Set)*.

The predicates can be of the following types and their selectivity can be estimated as follows:

1. Simple Predicate. This is of the form, *Exp Comp Constant* or *Constant Comp Exp*. Where the comparators are "=", ">", ">=", "<", "<=" and "!=".
   The size of *Pred.sel* for a predicate of the form *Exp Comp Constant* is given below,

   Comparator "=" , *Pred.sel = 1/Size.of.Exp*

   Comparator ">", *Pred.sel = TopRange (Exp.of, Constant)/Size.of.Exp*

   Comparatotr "<", *Pred.sel = BottomRange (Exp.of, Constant)/Size.of.Exp*

   Comparator ">=", *Pred.sel = (Exp = Constant) and (Exp > Constant)*

   Comparator "<=", *Pred.sel = (Exp = Constant) and (Exp < Constant)*

   Comparator "!=", *Pred.sel = (1 - 1/Size.of.Exp)*

   The two functions, *BottomRange* and *TopRange*, that are used to estimate the selectivity are defined as follows.

Function BottomRange:

```
BottomRange(o : Extent Record, e : Constant) : float
      if (e > o.highest) return (0)
      else if (e > o.lowest) return (e - o.lowest)
      else return (Exp.of.Size)
```

Function TopRange:

```
TopRange(o : Extent Record, e : Constant) : float
       if (e < o.lowest) return (0)
       else if (e < o.highest) return (o.highest - e)
       else return (Ext.of.Size)
```

2. *Pred1 and Pred2.*
   *Pred.sel = Pred1.sel * Pred2.sel*
3. *Pred1 or Pred2*
   *Pred.sel =(Pred1.sel + Pred2.sel) - Pred1.sel * Pred2.sel*
4. *not Pred1*
   Pred.sel = (1 - Pred1.sel)
5. *True*
   Pred.sel = 1
6. *False*
   Pred.sel = 0

Consider the query, Q_{10} = ***select*** *(lambda (s) s.age = 15) (Students)*

This query returns a set of people whose age is equal to *15*. The fully annotated parse tree for the above query is given in the figure below.

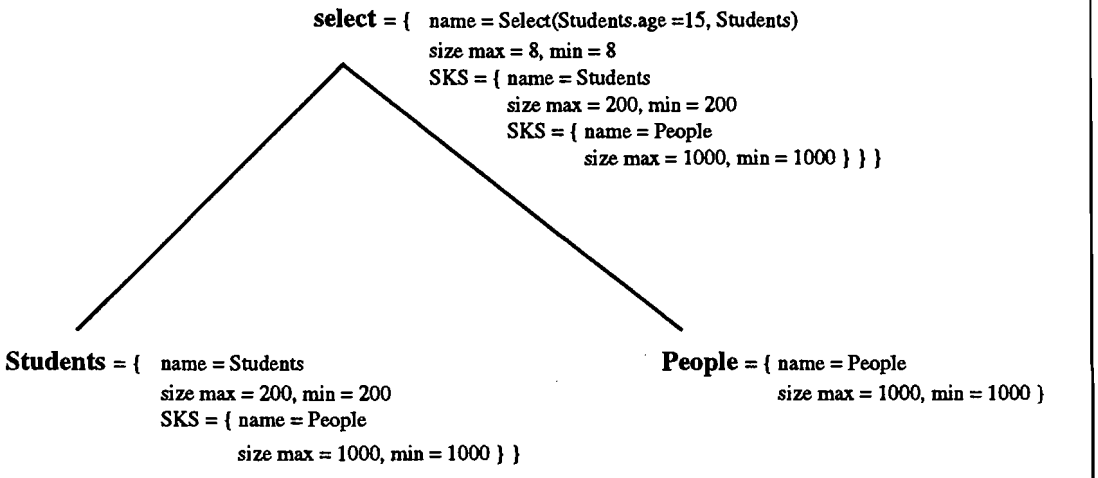

FIGURE 6. Shows the fully annotated parse tree for Q_{10}

4.1.11 Operator Forall

The operator, *forall*, returns True or False depending on whether all the elements of a given set satisfy a given predicate. We estimate a fraction between *0* and *1* to use as the probability that the *forall* evaluates True. This is because *forall* is often used as a selectivity predicate for *select* (Section 4.1.10). The predicates for *forall* are exactly the same as those for *select.*

Consider the query, Q = ***forall*** *(Pred) (Set).* Let the fraction of elements that satisfy the given predicate be *Pred.sel* (this is kept as the size of the predicate).
Then the size of the resultant set, R, is given by, $R = (Pred.sel)^{Size.of.Set}$

4.1.12 Operator Exists

The operator, *exists*, returns True or False depending on whether any element of a given set satisfies a given predicate. It is usually used as a selectivity predicate for *select* (Section 4.1.10), and hence we estimate its size as the probability that the *exists* operator evaluates to True. The predicates for *exists* are exactly the same as those for *select.*

Consider the query, Q = ***exists****(Pred) (Set).* Let the fraction of elements that satisfy the given predicate be *Pred.sel* (this is kept as the size of the predicate).
Then the size of the resultant set, R, is given by, $Size.of.R = Pred.sel.$

4.1.13 Operator Dup_elim

The AQUA operator, *dup_elim* takes in a set of elements and removes duplicates from the set, where the equality is tested using the user-defined input equality function *eq*.

Dup_elim queries can be of four kinds.

1. Q_{11} = ***dup_elim*** *(True) (Set)*

   The size of the resultant set is 1 as all the elements belong to the same equivalence class.

2. Q_{12} = ***dup_elim*** *(False) (Set)*

   The size of the set is equal to the size of the original set as this will place each element of the set into a different equivalence class.

3. Q_{13} = ***dup_elim*** *(lambda (x, y) (x.tail = y.tail)) (Set)*

   This is equivalent to the query, apply (lambda (x) x.tail) (Set). Thus similar techniques are used to find the size of these sets as are used to find the size of image sets.

4. Q_{14} = ***dup_elim*** *(lambda (x, y) (p_1 and p_2 and .... p_n) (Set)* where each p_i is the comparing two attributes of *Set*, of the same type; i.e. each p_i is of the form *x.tail = y.tail.*

   The size of the set formed by p_i can be calculated as above.

   Let *R* be the resultant set, then the size is given by,

   Size.of.R.max = minimum ($p_1 * p_2 * .....p_n$, *Size.of.Set)*

   Size.of.R.min = maximum (p_1, p_2, ... p_n)

Consider the query,

Q_{15} = ***dup_elim*** *(lambda (x, y) x.age = y.age and x.address = y.address) (People)*

The fully annotated parse tree for the above query is given in Figure 7.

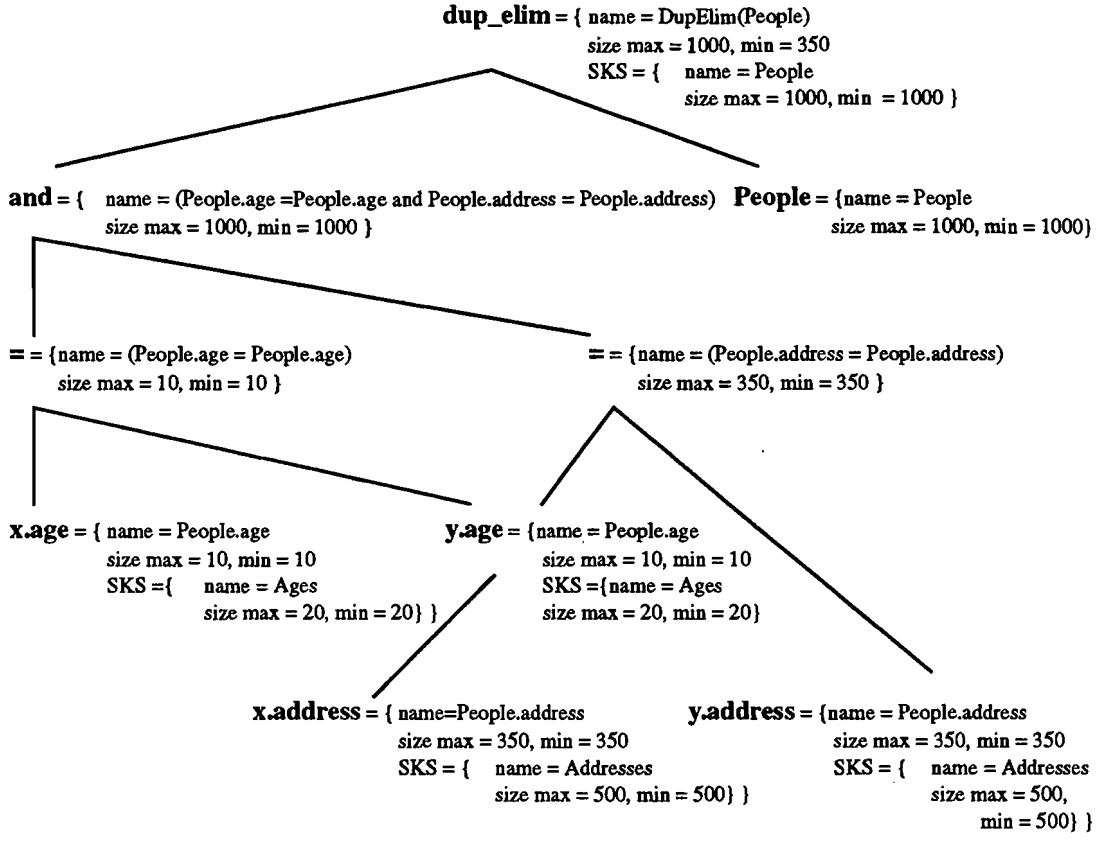

FIGURE 7. Shows the fully annotated parse tree for query Q_{15}

4.1.14 Operator Group

The query operator, *group*, takes in a set and groups its elements according to an equivalence class by function *f*. It returns a set of tuples with one tuple per equivalence class. Each tuple has two fields, *fst* and *snd*. The first item is the value returned when the function, *f*, is applied to the input set and second is the set of elements that satisfy the given function, *f*.

Group queries can be of three kinds.

1. Q_{16} = ***group*** *(lambda (x) True) (Set)*

   The size of the set is *1* as all the elements satisfy are put into the same equivalence class.

2. Q_{17} = ***group*** *(lambda (x) False) (Set)*

   The size of the output set is equal to the size of the original set as each of the elements of the set are put into a different equivalence class.

3. Q_{18} = ***group*** *(lambda (x) x.tail) (Set)*

   Here the function *x.tail* is being applied to each element of *Set*. The size of the set formed by this can be calculated in a manner similar to the one that is used to calculate the size of image sets. If *R* be the resultant set, and *A* = ***apply*** *(lambda (x) x.tail) (Set)*,

then the size of the resultant set is given by, *R.of.Size = Size.of.A*
The size of the second field *snd,* is given by *Size.of.snd = Size.of.Set/R.of.Size*
The size of the first field *fst,* is given by ,
*Size.of.fst = Size.of.**choose** (**apply** (lambda (x) x.tail) (Set))*
The size of the *choose* record of the set is calculated in a manner similar to that described in Section 4.1.4.

For sets of tuples, we keep a record for each of the fields of the tuple. In addition to the information about the size of the set, we also maintain information about the number of distinct objects there are, in the set for each tuple field. For example, the above set of tuples has two fields, *fst* and *snd.* We maintain information about the number of distinct objects *fst,* and the number of distinct objects *snd* in the set.

Consider the query, Q_{19} = ***group** (lambda (x) x.address) (People)*

The extent records for the above query, are given below.

TABLE 3. Table showing records in the set group (lambda (x) x.address) (People)

No.	Name	Size	SizeSSet	SKS	Choose
1	Group(People.address, People)	[350, 350]	-	-	2
2	Choose(Group(People.address, People))	[1, 1]	-	-	-
3	Choose(Group(People.address., People)):fst	[1, 1]	[350, 350]	6	-
4	Choose(Group(People.address, People)):snd	[1000/350, 1000/350]	[350, 350]	5	-
5	People	[1000, 1000]	-	-	-
6	People.address	[350, 350]	-	7	-
7	Addresses	[500, 500]	-	-	-

4.1.15 Operator Flatten

The AQUA operator, *flatten,* removes one level of nesting in a set of sets.

At present, we infer the size of the resultant set from the extent records of the input set. If the choose record of the input set has an SKS, then the size of the resultant set is equal to the size of this SKS. If this record does not exist, we cannot find the size of the resultant set.

4.1.16 Operator Mem

Mem, takes in a set and an input element. It returns TRUE, if the input element is an element of the input set, using the user defined equality function *eq,* to test for equality. We estimate a fraction between 0 and 1 to use as the probability that *mem* evaluates to True. This is because *mem* is often used as a selectivity predicate for *select.*

Mem queries are of four types.

1. Q_{20} = ***mem*** *(true, a) (S)*
   The size of the set is *1*.

2. Q_{21} = ***mem*** *(false, a) (S)*
   The size of the set is *0*.

3. Q_{22} = ***mem*** *(=, a) (S)*
   Let *R* be the resultant set. The nodes of the lambda variables *x* and *y*, are annotated with information about set *S*. If the input set is a set of reals (integers), then range of the input element *a* is compared to the range of the input set *S*. If *a* lies within this interval then the size of the resultant set, *R.size = 1/Size.of.S* else *R.size = 0*. If the input set is not a set of reals (integers), *R.size = 1/Size.of.S*.

4. Q_{23} = ***mem*** *(lambda (x, y) Pred, a) (S)*
   Let *R* be the resultant set. The nodes of the lambda variable *x* and *y*, are annotated with information about set *S*.
   Mem predicates can be of two types,
   1. *Pred = x.tail = y.tail,*
      in this case *size.of.x.tail = size.of.y.tail =* ***apply*** *(lambda (x) s.tail) (S)*
      *Pred.size = 1/****apply*** *(lambda (x) s.tail) (S)*
   2. *Pred =* $(p_1$ *and* $p_2 \ldots . p_n)$ where each p_i is of the form *x.tail = y.tail* for *i = 1 to n*
      The size of each p_i is calculated as above. *Pred.size =* $(p_1 * p_2 * \ldots . p_n)$

   The size of the resultant set is given by *R.size = 1/Pred.size*

Consider the query,
Q_{24} = ***mem*** *(lambda (x, y) x.age = y.age,* ***choose*** *(Employees)) (Students)*

The fully annotated parse tree for the above query is given in the figure below.

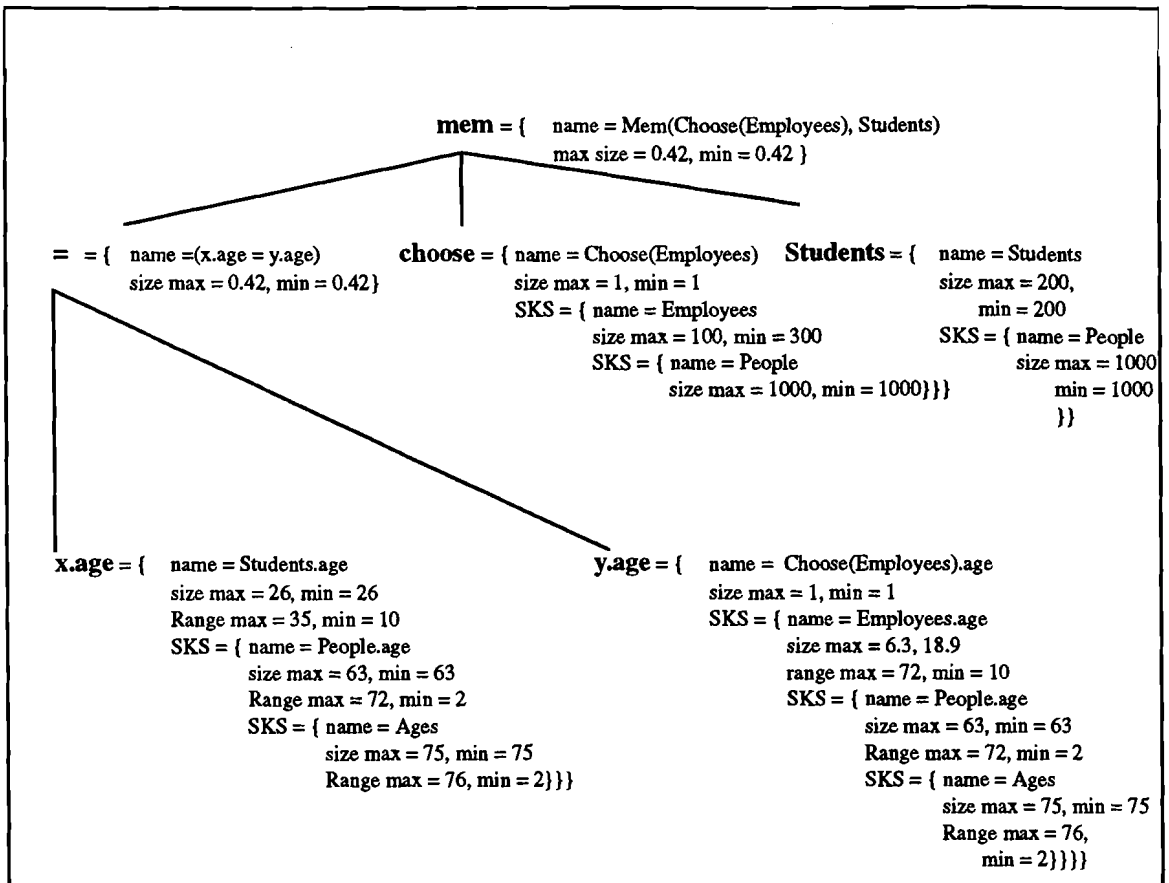

FIGURE 8. Shows the fully annotated parse tree for Q_{24}

4.2 Tuple Operators in AQUA

Tuple operators, are those that operate on tuples or sets of tuples only. The operators *tuple*, *tup_concat* and *tup_select* operate on tuples. The operators, *nest* and *unnest*, operate on sets of tuples. For each set of tuples S we maintain the following information:

1. The name of the tuple set.
2. The size of the tuple set.
3. A pointer to the record for its smallest known superset, if any.
4. When we choose a random element of the set, we get a single tuple of the set. Thus we maintain a pointer to its choose record, whose name is of the form *Choose(S)*. This is the record for a single tuple. We keep this record because whenever we consider the a tuple field we are considering that field of one particular tuple and not a field for a set of tuples. The records we maintain for the set are thus in the form *Choose(S):tuple_field.*
5. A record for each field of the tuple.

For each field of the tuple, we keep the following information

1. The name of the field, which is of the form *Choose(S):field_name*

2. The size of the tuple field.
3. A pointer to its choose record if the field contains a set of objects.
4. A pointer to its SKS record.
5. The range of values it can take, if the field contains a set of objects.
6. The number of unique field elements for each field of the tuple (SizeSSet). This is because we cannot always get an accurate estimate of the number of unique field elements from the information about the smallest known superset of the field. An example explaining why this information needs to be kept is given in Section 4.2.1 (see Q_{27}) below.

Let us consider the set of tuples, *TStudents* with the fields, *name*, *age*, *major*. Then the extent record kept for the above set of tuples is given below.

TABLE 4. Table showing the records for the set of tuples TStudents

No	Name	Size	Range	SizeSSet	SKS	Choose
1	TStudents	[100, 150]	-	-	-	2
2	Choose(TStudents)	[1, 1]	-	-	-	-
3	Choose(TStudents):name	[1, 1]	-	[10, 15]	-	-
4	Choose(TStudents):age	[1, 1]	[16, 48]	[33, 33]	6	-
5	Choose(TStudents):major	[1, 2]		[4, 8]	-	-
6	Age	[81, 81]	[1, 82]	-	-	-

4.2.1 Operator Tuple

The operator *tuple* takes in a number of labels and an equivalent number of data items and makes a labelled list of data items by associating the two lists pairwise.

Tuple queries can be of two types. Either a single tuple is created or a set of tuples is created depending on the form of the query.

1. Q_{25} = ***tuple*** *(a, b) ("16", "ft")*
   The size of queries of this kind is obviously *1*. The size of each of the fields, *a* and *b*, of the tuple is the size of the corresponding input set.
2. Q_{26} = ***apply*** *(lambda (x) << name: x.name, age:x.age, major: x.major>>) (Students)*
   In this case, the query operator, *tuple*, is applied to each path expression of the form *x.tail* where *x* is each element of the set *Students*. Let the size of the above set be *S*.

   Then the maximum and minimum size of the set is given by:
   *S.max = min (Size.of.Students, (Size.of.Students.name * Size.of.Students.age * Size.of.Student.major))*
   *(Size.of.Students.name * Size.of.Students.age * Size.of.Student.major)*, gives the size of all possible combinations of the three attributes *name*, *age* and *major* of the set of *Students*. This gives the upper bound on the estimate of the maximum number of tuples

there can be in the set. But we limit this by the size of the set of *Students* because we cannot possible have more tuples in the resultant set than in the input set as it is the elements of this input set that are being converted into tuples.

S.min = max (Size.of.Students.name, Size.of.Students.age, Size.of.Student.major)
This gives a lower bound on the size of the set. This gives a lower bound on the estimate of the combinations of the three attributes *name, age* and *major* of the set of *Students.* Thus there are atleast *max(Size.of.Students.name, Size.of.Students.age, Size.of.Student.major)* many distinct tuples in the set.

We also maintain records for each of the fields, *name, age* and *major.* The size of each is equal to the size of the choose record for each of the sets *Students.name, Students.age and Students.major* respectively. The size of their supersets is equal to the size of the input set *Students.name, Students.age* and *Students.major* respectively.

The fully annotated parse tree for the above query is given in Figure 8, below.

Consider the query,
Q_{27} = ***apply*** (lambda (x) ***tuple*** *(a, b) (x.major, select (lambda (p) p.age = 21) (People)) (Students)*

Let the resultant set be *R.* The field *a* of each tuple of *R* is of type *Major* while the field *b* is of type *Set[Person].*
Here each tuple of the set *R,* has the same elements in field *b.* Thus the size of the set is equal to the size of the set *People.major.*

The table below shows the elements in the above resultant set.

TABLE 5. Table showing the elements in the set formed by the query apply (lambda (x) tuple (a,b) (x.major, select (lambda (p) p.age = 21) (People)) (Students)

No	Name	Size	SizeSSet	SKS	Choose
1	SetOfTup(a,b)	[50, 50]	-	-	2
2	Tup(a,b)	[1,1]	-	-	-
3	Tup(a,b):a	[1, 1]	[50, 50]	5	-
4	Tup(a,b):b	[16.21, 16.21]	[1, 1]	6	-
5	Major	[75, 75]	-	-	-
6	People	[1000, 1000]	-	-	-

As can be seen, the size of the superset of field *b* is 1 though the size of its superset is *1000.* Thus the number of unique field elements for the field *b* is *1.* If we were to select all possible field elements b (i.e. a query of the form *apply (lambda (x) tup_select (b) (x)) (Set)*) from this set we would get an accurate estimate from the record of its SizeSSet than we would if we used the size of its smallest known superset (SKS).

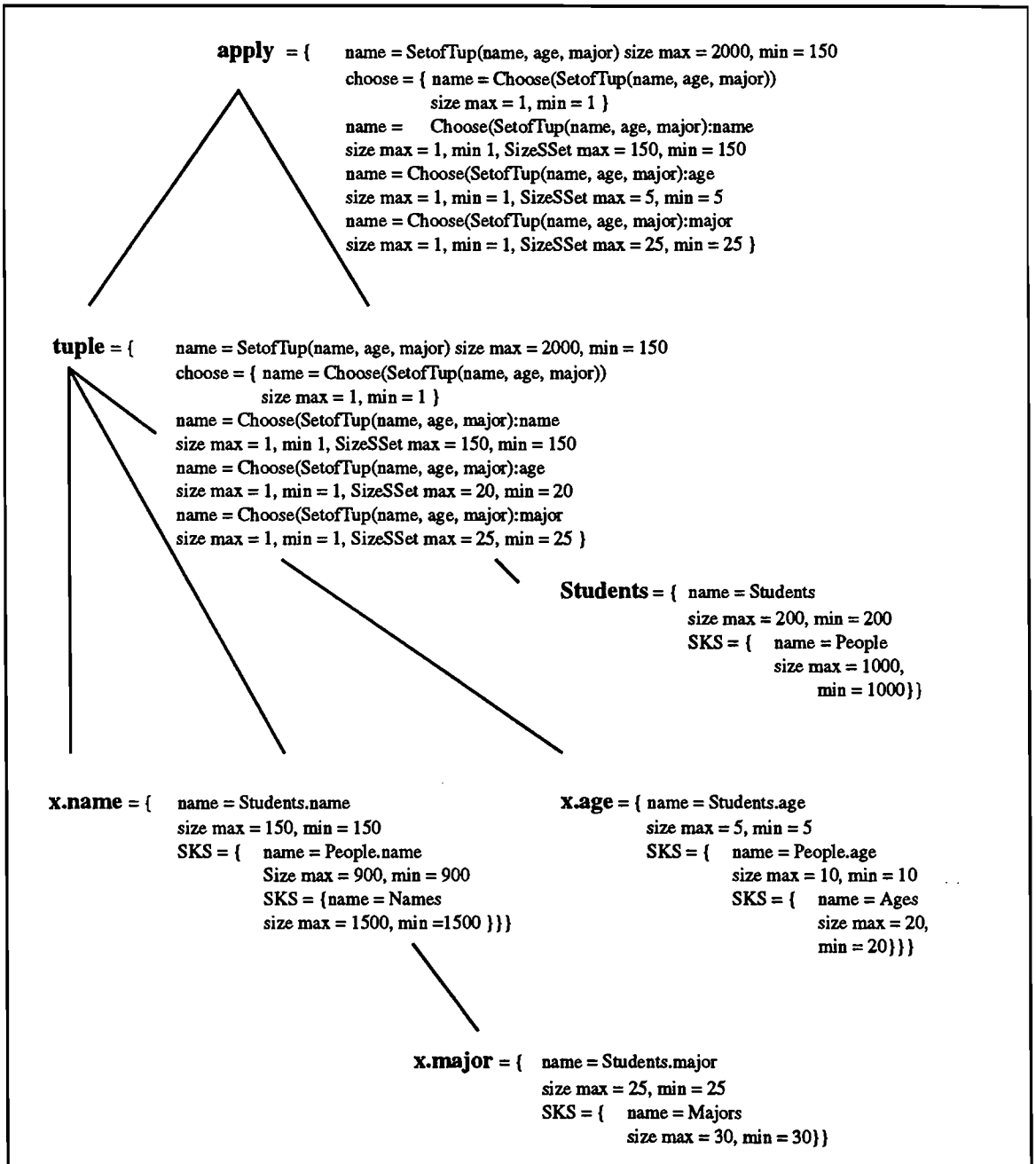

FIGURE 9. Shows the annotated parse tree for Q_{26}

4.2.2 Operator Tup_concat

The operator *tup_concat* takes in two tuples and concatenates them into a single tuple.

Tup_concat queries can be of two kinds.

1. Q_{27} = ***tup_concat*** *(<<a: "a", b:"b">>, tuple (x) ("string"))*

   In this case, only a single tuple is being created. Hence the size of the set is *1*.

2. Q_{28} = ***apply*** *(lambda (x)* ***tup_concat*** *(x,* ***tuple*** *(a) ("hi")) (TStudents)*

In this case a set of tuples is being created instead of a single tuple. The two input elements might either both be sets of tuples or one might be a tuple and the other a set of tuples. If an input element is a set of tuples, as above, then the operator, *tup_concat*, is applied to each element of the input set. Therefore in this case the size of the resultant set is the product of the size of the two input elements. This is because each tuple in each set is unique. So the concatenation of any one from one set with any one from the other set is unique.

4.2.3 Operator Tup_select

The operator *tup_select*, selects a specific field of the tuple based on the input label.

Tup_select queries can be of two kinds.

1. Q_{29} = ***tup_select*** *(name)* ***(choose*** *(TStudents))*

The records for each field of a tuple is kept in the extent record table. Thus the size of the selected field is obtained directly from the extent record table.

2. Q_{30} = ***apply*** *(lambda (x)* ***(tup_select*** *(name) (x)))(TStudents)*

In this case, the function *tup_select* is being applied to each element of the set TStudents. The result is a set of elements of the tuple field *name* selected from the set TStudents. To calculate the size of the resultant data set, we use an approach similar to the one used to calculate the size of image sets.

Let us consider the query, Q = ***apply*** *(lambda (x)* ***(tup_select*** *(field) (x)) (S)*
Let *SKS(S)* be the smallest know superset of S and *SKS(S)*. *field.SizeSSet* be the total number of unique elements in the field labelled *field*. Then if R is the resultant set,

$$R \cdot size = \frac{(SKS((S) \cdot field \cdot SizeSSet)) \cdot (SKS(S) \cdot field \cdot SizeSSet)}{SKS(SKS(S) \cdot field) \cdot size}$$

The above approach is then used to calculate the size of the resultant sets

4.2.4 Operator Nest

The query operator, *nest*, operates on a set of tuples. The operator takes a user defined equality, *eq*, a label, L, and a set of tuples, S. We do a pairwise comparison of the elements of S using the user defined equality function, *eq*. The tuples which satisfy the user defined equality are collected into a set. The result of applying the query operator, *nest*, is a set of tuples with two fields *fst* and *snd*. Each element of the field *snd* corresponds to the elements of S minus the field labelled L, that satisfy the equality function *eq*. The field *fst* corresponds to any element of the field labelled L corresponding to the set of elements of the field *snd*.

Nest queries can be of two kinds.

1. Q_{31} = ***nest*** *(lambda (x, y) (((**tup_select** (name) (x)) = (**tup_select** (name) (y)) and*
*((**tup_select** (age) (x)) = (**tup_select** (age) (y)))), major)*
(TStudents)

Let the resultant set be *R*. The size of *((**tup_select** (name) (x)) = (**tup_select** (name) (y))* is obviously the size of the set *(**tup_select** (name) (TStudents))*. Similarly, the size of *((**tup_select** (age) (x)) = (**tup_select** (age) (y))* is the same as the size of the set *(**tup_select** (age) (TStudents))*.
Let *P* be the size of, *(((**tup_select** (name) (x)) = (**tup_select** (name) (y)) and ((**tup_select** (age) (x)) = (**tup_select** (age) (y)))), major)*.

*P.max = minimum ((Size.of.(**tup_select** (name) (TStudents)) **
*Size.of.(**tup_select** (age) (TStudents)), Size.of.TStudents.max)*
*P.min = maximum (Size.of.(**tup_select** (name) (TStudents),*
*Size.of.(**tup_select** (age) (TStudents))*

The size of the predicate is calculated in a manner similar to *dup_elim* (Section 4.1.13) *P* gives the number of equivalence classes into which we divide the input set. Each of these equivalence classes give us a single tuple.

Therefore size of the resultant set R, *R.size = P.size*

The size of the field *fst = 1*,
Let the size of its superset be *fst.SizeSSet*. Then *fst.SizeSSet = min (R.size, major.size)*

The size of the field *snd = TStudents.size/R.size*
Let the size of its superset be *snd.SizeSSet . Then* snd.SizeSSet = *R.size*

In general for a query *Q* = ***nest*** *((sp$_1$ and sp$_2$ ..... sp$_n$), label) (S)*,

where *sp$_i$ = **tup_select** (l$_i$) (x) = **tup_select** (l$_i$) (y), i = 1 to n.*

*predicate.size.max = minimum (sp$_1$ * sp$_2$ * ..... * sp$_n$, Size.of.S.max)*

predicate.size.min = maximum (sp$_1$, sp$_2$, ... sp$_n$). If *R* is the resultant set, then *R.size = predicate.size*

2. Q_{32} = ***nest*** *(=, major) (TStudents)*

In this case, the equality function is defined over all the fields of *TStudents* other than *major*. Each tuple of the set *TStudents* has three fields, *name, age* and *major*. The equality function is thus assumed to be defined over the fields *name* and *age* which is similar to Q_{28}. Therefore size of the resultant set can be calculated in exactly the same way as Q_{28}.

In general, for queries of the form *Q = nest (=, l$_1$) (S)*,

If the set *S* has fields labelled l_1, l_2, ...l_n. Then the predicate "=" is equivalent to the predicate *p = (sp$_2$ and sp$_3$ and ..... sp$_n$)*, where each *sp$_i$ = (**tup_select** (l$_i$) (S) = **tup_select** (l$_i$) (S)) for i = 2 to n.*

*p.max = minimum ((sp$_2$ * sp$_3$ * .... sp$_n$), Size.of.S.max)*
p.min = maximum (sp$_2$, sp$_3$, ... sp$_n$)

The size of the resultant set, *R*, is given by, *R.size = p.size*

The size of the field *fst = 1*,
Let the size of its superset be *fst.SizeSSet.* Then *fst.SizeSSet = min (R.size, major.size)*

The size of the field *snd = TStudents.size/R.size*
Let the size of its superset be *snd.SizeSSet . Then* snd.SizeSSet = *R.size*

4.2.5 Operator unnest

Operator *unnest* is the inverse of operator *nest*. It takes a nested set and then unnests the field *snd* to return a set of tuples of type similar to the set that was nested in the first place.

Consider the query *Q = **unnest** (snd) (S)*.
It is assumed that the set *S* was nested and has two fields, *fst* and *snd*. Then the size of the resultant set is the product of the size of the set *S* and the size of the tuple field labelled *snd* in the tuple set, *S*. The records for both can be found in the extent record table. The size of *snd* gives us the average size of each set labelled *snd*. Since we are unnesting on the field *snd*, the product of the size of set *S* and the size of the tuple field labelled *snd* gives us the size of the original set.

5.0 Internal Representation of Cost Information

As discussed earlier, within the optimizer, the query is maintained in the form of an EAT. The EAT is composed of alternating layers of function nodes and data nodes connected by labelled arcs.

The function nodes represent the action that can be taken on the data. They thus contain the methods to calculate the size and cost of a query.

The data nodes represent the data that is being manipulated during the execution of a query. It contains the information about size and cost of the execution of a query. This information is maintained in the extent record table.

The extent record table maintains the following information,

1. The name of the set.
2. An estimate of the cost; which is kept as an interval.
3. An estimate of the size of the set, which is kept as an interval.
4. The range of value that the set can take, if it is a real, an integer or a set of reals or integers.
5. Information about its smallest known superset (SKS).
6. Information about its choose record if it is a complex object or a nested set or a set of tuples.

The following figure shows the different classes related to the cost model.

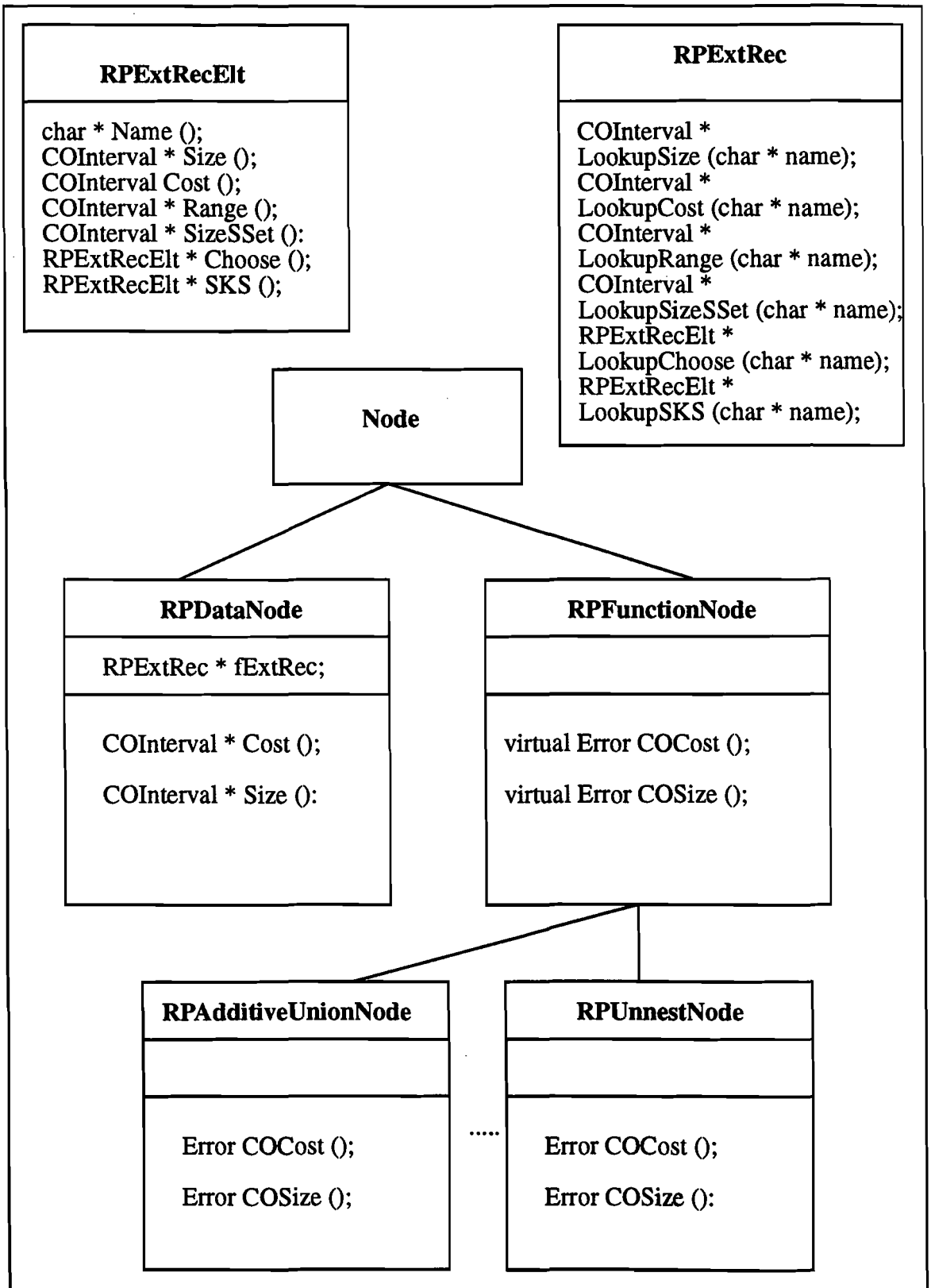

FIGURE 10. Figure showing classes and methods related to the cost model.

5.1 Methods Dealing with Size and Cost Annotations

5.1.1 RPDataNode

The following are the methods in RPDataNode dealing with extent records.

Public Methods

- **RPExtRec * ExtRec ();**

  Returns a pointer to the extent table.

- **Error insertExtRec (char * name, COInterval * size, COInterval * cost, COInterval * range, COInterval * sizeSSet, RPExtRecElt * SKS, RPExtRecElt * Choose);**

  Inserts the record at the end of the table.

- **Error insertExtRec (RPExtRec * extrec);**

  Copies the contents of extrec at the end of the table; returns an error message if it cannot.

- **void SetOCost (COInterval * cost);**

  Sets the override cost.

- **void ClearOCost ();**

  Clears the override cost.

- **void SetOSize (COInterval * size);**

  Sets the override size.

- **void ClearOSize ();**

  Clears the override size.

- **COInterval * OCost ();**

  Returns the override cost.

- **COInterval * OSize ();**

  Returns the override size.

- **COInterval * Cost ();**

  Returns the override cost if it exists; else returns the estimated cost; returns an error if the cost has not been computed.

- **COInterval * Size ();**

  Returns the override size of the set if it exists; else returns the estimated size; returns an error if the size has not been computed.

- **float SizeMax ();**

  Returns the maximum override size if it exists; else returns the maximum size calculated.

- **float SizeMin ();**

  Returns the minimum override size if it exists; else returns the maximum size calcu lated.

- **float CostMax ();**

  Returns the maximum override cost if it exists; else returns the maximum cost calculated.

- **float CostMin ();**

  Returns the minimum override cost if it exists, else returns the maximum cost calculated.

- **Error inferCost ();**

  Calculates the cost of the query. Returns OK if it is successful; else returns NOT_OK. This is a method that calculates the cost of a query starting at the top of the EAT.

- **Error inferSize ();**

  Calculates the size of the set returned by the query. Returns OK if it is successful; else returns NOT_OK. This calculates the size of a query, starting at the top of the EAT.

- **void DelCostAnn ();**

  Starting from the top of the tree, it deletes all the size and cost annotations.

- **Error ReInferSize ();**

  Calculates the size of the set returned by the query again. Returns OK if it is successful; else returns NOT_OK.

- **Error getExtRec ();**

  If the data node is a leaf node, the extent record for the node is obtained from the Schema Manager. If the data node is not a leaf node, the function returns NOT_OK.

- **void PrintExtRec ();**

  Prints the extent record table for the set.

5.1.2 class RPExtRec

The extent record for a data set is kept in the form of a table. The table has information about the size of the set, its cost, the range of values it can take if it is a set of reals or integers, its SKS record and choose record.It also maintains information about the number of unique field elements in the set for each field of the tuple. The first record in the table is always the record for the data set.

Public Methods

- **RPExtRec ();**

  Constructor.

- **~ RPExtRec ();**

  Destructor.

- **COInterval * LookupCost (char * name);**

  Looks up the cost of the set in the extent table. If the symbol does not exist in the table it returns an error.

- **COInterval * LookupSize (char * name);**

  Looks up the size of the set in the extent table. If the symbol does not exist in the table it returns an error.

- **COInterval * LookupRange (char * range);**

  Looks up the range of the set in the extent table. If the symbol does not exist in the table it returns an error.

- **RPExtRecElt * SKS (char * name);**

  Looks up the SKS of the named symbol. If the symbol does not exist in the table it returns an error.

- **RPExtRecElt * Choose (char * name);**

  Looks up the choose record of the named symbol. If the symbol does not exist in the table it returns an error.

- **Boolean LookupName (char * name);**

  Looks up the name of the symbol in the table and returns TRUE if it exists; else returns FALSE.

- **LookupRecord (char * name);**

  Looks up the name of the symbol in the table and returns a pointer to its record. If the symbol does not exist in the table it returns an error.

- **RPExtRecElt * LookupTupField (char * setname, char * label);**

  Returns a pointer to the record for the named field of the set.

- **RPExtRecElt * ithExtRec (int i);**

  Returns a pointer to the i^{th} record in the table. Counting starts at 0.

- **Error CopyExtRec (RPExtRec * extrec);**

  Copies records into the extent record table. Returns an error message if it cannot do so.

- **Error AddExtRec (char * name, COInterval * size, COInterval * cost, COInterval * range, RPExtRecElt * SKS, RPExtRecElt * choose);**

  Adds the extent record at the end of the table.

- **void PrintExtRec ();**

  Prints the extent record table.

- **int NumElts ();**

  Returns the number of elements in the extent record table.

- **RPExtRecElt * Elements ();**

  Returns a pointer to the array containing the extent records.

5.1.3 class RPExtRecElt

RPExtRecElt represents an element in the extent record table. It contains information about the size of the set, its cost, range of values it can take, its SKS and choose record. It also maintains information about the number of unique field elements in the set, for each field of the tuple.

Public Methods

- **RPExtRecElt ();**

  Parameterless constructor.

- **RPExtRecElt (char * name, COInterval * size, COInterval * cost, COInterval * range, COInterval * sizeSSet, RPExtRecElt * SKS, RPExtRecElt * choose);**

  Constructor.

- **~ RPExtRecElt ();**

  Destructor.

- **char * Name ();**

  Returns the name of the symbol.

- **COInterval * size ();**

  Returns the size of the set represented by the symbol.

- **COInterval * Cost ();**

  Returns the cost of the set.

- **COInterval * Range ();**

  Returns the range of values the set can take if it is a set of reals or integers.

- **COInterval * SizeSSet ();**

  Returns the size of the superset for tuple fields.

- **RPExtRecElt * SKS ()**

  Returns a pointer to its SKS record.

- **RPExtRecElt * Choose ();**

  Returns a pointer to the symbol's choose record.

- **void SetSize (COInterval * size);**

  Sets the size of the set.

- **SetCost (COInterval * cost);**

  Sets the cost of the set.

- **SetRange (COInterval * range);**

  Sets the range of values that can be taken by the object if it is a real or an integer or if it is a set of real or integers.

- **SetSizeSSet (COInterval * sizeSSet);**

  Sets the size of the superset for fields of tuples.

- **SetSKS (RPExtRecElt * SKS);**

  Sets the SKS record of the symbol.

- **SetChoose (RPExtRecElt * Choose);**

  Sets the choose record of the symbol.

5.2 Methods Dealing with Estimation of Size and Cost of a Query

5.2.1 class RPFunctionNode

Public Methods

- **virtual Error COCost ();**

  Calculates the cost of a query. Returns an error message if it cannot.

- **virtual Error COSize ();**

  Calculates the size of the resultant data set of a query. Returns an error message if it cannot.

5.2.2 class RPApplyNode

Public Methods

- **Error COSize ();Section**

  Calculates the size of the resultant data set. Returns an error message if it cannot. An error is returned if the extent records of the two child data nodes cannot be calculated. See Section 4.1.1 for a description of the technique used to calculate the size of the output set.

5.2.3 class RPArithOpNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set. Returns an error message if it cannot. An error is returned if the extent records of the two child data nodes cannot be calculated. See Section 4.1.2 for a description of the technique used to calculate the size of the out put set.

5.2.4 class RPBoolOpNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the boolean operators *and* and *or.* Returns an error if the size cannot be calculated.

5.2.5 class RPCompOpNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the comparative operators "=", "!=", ">", ">=", "<", "<=". If the parent operator is *dup_elim, nest, partition* or *mem,* then the function returns an error if the size of their *intersection* cannot be calculated. If the parent node is *select, forall, group, exists* or *select,* then the function calculates the size of the set only if a set of reals (integers) is being compared to a real (integer).

5.2.6 class RPChooseNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant set for the operator *choose*; returns an error if the extent record for the child node cannot be calculated. See Section 4.1.4 for a description of the technique used to calculate the size of the output set.

5.2.7 class RPDiffOpNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant set for the operator *difference*; returns an error if the extent records for the two input data sets cannot be calculated. See Section 4.1.6 for a description of the technique used to calculate the size of the output set.

5.2.8 class RPDup_ElimNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant set for the query operator *dup_elim*; returns an error if the extent records for the two input data nodes cannot be calculated. See Section 4.1.13 for a description of the technique used to calculate the size of the output set.

5.2.9 class RPExistsNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant set for the query operator, *exists*; returns an error if the extent record for the two input data sets cannot be calculated. The size returned is always between *0* and *1*. See Section 4.1.12 for a description of the technique used to calculate the size of the output set.

5.2.10 class RPForallNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *forall*; returns an error if the size of the two input data sets cannot be calculated. The size is always between *0* and *1*. See Section 4.1.11 for a description of the technique used to calculate the size of the output set.

5.2.11 class RPFlattenNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *flatten*; returns an error if the size of the input data node cannot be calculated. The size cannot be calculated if the *choose* record of the input set does not have an SKS record.See Section 4.1.15 for a description of the technique used to calculate the size of the output set.

5.2.12 class RPGroupNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *group*; returns an error if the size of the two input data sets cannot be calculated. See Section 4.1.14 for a description of the technique used to calculate the size of the output set.

5.2.13 class RPIntersectNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *intersect*; returns an error if the size of the two input data sets cannot be calculated. See Section 4.1.5 for a description of the technique used to calculate the size of the output set.

5.2.14 class RPMSetNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *mset*; returns an error if the size of the input set cannot be calculated. See Section 4.1.8 for a description of the technique used to calculate the size of the output set.

5.2.15 class RPMemNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *mem;* returns an error if the size of the input data sets cannot be calculated. The size of the set is always between *0* and *1*. See Section 4.1.16 for a description of the technique used to calculate the size of the output set.

5.2.16 class RPMethodNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator, *method*; returns an error if the size of the input set cannot be calculated. The size cannot be calculated if the resultant data set does not have a smallest known superset whose size can either be calculated or looked up in the SchemaManager.

5.2.17 class RPNestNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator, *nest*; returns an error if the size of the input sets cannot be calculated. At present, a set of tuples can be nested on one field only. See Section 4.2.4 for a description of the technique used to calculate the size of the output set.

5.2.18 class RPNotNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *not*; returns an error if the size of the input node cannot be calculated. It is assumed that the size of the input set in between *0* and *1*. The size of the resultant data set is always between *0* and *1* .See Section 4.1.3 for a description of the technique used to calculate the size of the output set.

5.2.19 class RPSelectNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator, *select*, returns an error if the size of the input sets cannot be calculated. See Section 4.1.10 for a description of the technique used to calculate the size of the output set.

5.2.20 class RPSetNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *set*; returns an error if the size of the input set cannot be calculated. See Section 4.1.9 for a description of the technique used to calculate the size of the output set.

5.2.21 class RPTupConcatNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *tup_concat*; returns an error if the size of the input set cannot be calculated. It assumes that no two labels of the input tuples are the same. See Section 4.2.2 for a description of the technique used to calculate the size of the output set.

5.2.22 class RPTupleNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *tuple*; returns an error if the size of the input field nodes cannot be calculated. See Section 4.2.1 for a description of the technique used to calculate the size of the output set.

5.2.23 class RPTupSelectNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *tup_select*; returns an error if the size of the input field nodes cannot be calculated. If the input data is a tuple the size cannot be calculated if there is no record for the field of the tuple that is being selected. If the input data nodes is a set of tuples then the size of the tuple cannot be calculated if there is no record for the field of the tuple set that is being selected or if the tuple set does not have an SKS. See Section 4.2.3 for a description of the technique used to calculate the size of the output set.

5.2.24 class RPUnionNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *union*; returns an error if the size of the input field nodes cannot be calculated. See Section 4.1.7 for a description of the technique used to calculate the size of the output set.

5.2.25 class RPUnnestNode

Public Methods

- **Error COSize ();**

  Calculates the size of the resultant data set for the query operator *unnest*. Returns an error if the size of the input tuple set cannot be calculated. At this point, it is assumed that the unnesting is done only on the field *snd*. The size cannot be calculated if there are not records for the fields *fst* and *snd*. See Section 4.2.5 for a description of the technique used to calculate the size of the output set.

6.0 Database Schema

The database schema is stored in 4 files, TravelSchema, TravelTuples, DerivedTypes and TravelGlobals. The following is a description of the format of each of the files and the information contained in each one.

6.1 File TravelSchema

This file contains information about the global user defined types in the database. The global types Boolean, String, Integer and Real are present in the table. The rest are stored in the file TravelSchema. The following is the format for each record.

Class *class_name* : *super_type* (*number* **attr**)
attribute *attr_name* : *bulk_type* **none** *elt_type*

Here, the words highlighted are the keywords which are present for every record in the TravelSchema. *class_name* is the name of the user type being defined, *super_type* is the name of its supertype, *number* represents the number of attributes of the user defined type. This is then followed by the record for each of the attributes of the user defined type. *attr_name* represents the name of the attribute, *bulk_type* represents whether it is a set (*set*), a multiset (*mset*) or neither one of the two (*none*). *elt_type* represents its type.

A part of the file TravelSchema is given below in Figure 11.

```
Class Date : none (3 attr)
attribute month : none none Integer
attribute day : none none Integer
attribute year : none none Integer

Class Name : none (3 attr)
attribute first_name : none none String
attribute middle_name : none none String
attribute last_name : none none String

Class Address : none (5 attr)
attribute Country : none none String
attribute zip_code : none none String
attribute city : none none String
attribute street : none none String
attribute number : none none Integer

Class Person : none (4 attr)
attribute name : none none Name
attribute birthdate : none none Date
attribute age : none none Integer
attribute address : none none Address

Class Student : Person (2 attr)
attribute school : none none String
attribute major : set none String
```

FIGURE 11. Shows the format of file TravelSchema

6.2 File TravelTuples

This file contains information about the user defined tuples in the database. The following is the format for each record in the file.

Tuple *tuple_name* : **none** (*number* **fields**)
field *field_name* : *bulk_type* **none** *elt_type*

The words highlighted are the keywords which are present for every record in the file. *tuple_name* is the name of the tuple, *number* represents the number of fields of the given tuple. This is then followed by the records for each of the tuple fields. *field_name* repre-

sents the name of the field, *bulk_type* represents whether it is a set (*set*), multiset (*mset*) or neither one (*none*). *elt_type* represents its type.

A part of the file TravelTuples is given in Figure 12.

```
Tuple TDate : none (3 fields)
field month : none none Integer
field day : none none Integer
field year : none none Integer

Tuple TAddress : none (5 fields)
field country : none none String
field zip_code : none none String
field city : none none String
field street : none none String
field number : none none Integer

Tuple TPerson : none (5 fields)
field name : none none Name
birthdate : none none Date
field age : none none Integer
field address : none none Address
```

FIGURE 12. Shows the format of the file TravelTuples

6.3 File DerivedTypes

This file contains the types that are derived from the already existing user defined types in the database. For example, if we wanted to have a type for the set of set of Addresses, we would keep the record for it in this file. The following is the format for each record in the file.

Type *elt_name* **:** *bulk_type* **none** *elt_type*

The words highlighted are the keywords which are present for every record in the file. *elt_name* is the name of the type being added to the database, *bulk_type* represents whether it is a set (*set*), multiset (*mset*) or neither one (*none*), and *elt_type* represents its type.

A part of the file DerivedTypes is given in Figure 13.

```
Type Money : none none Integer

Type Age : none none Integer

Type Set[Address] : set none Address

Type Family : set none Person
```

FIGURE 13. Shows the format of the file DerivedTypes

6.4 File TravelGlobals

This file contains information about all the global symbols in the database. It maintains information that is needed to estimate the size and the cost of queries. Each record in the file has the following format.

global_name :
Type : *bulk_type none elt_type*
Size : *S1 S2*
SKS : *sks_name*
Range : *R1 R2*
SizeSSet : *SS1 SS2*

The words highlighted are the keywords which are present for every record in the file. *global_name* is the name of the global_symbol. *bulk_type* represents whether it is a set (*set*), multiset (*mset*) or neither one (*none*). *elt_type* represents its type. *S1*, *S2* is it's size, which is kept as an interval, *sks_name* is the name of it's smallest known superset, *R1*, *R2* is it's range (*-1*, *-1* if it is not of type real or integer). *SS1*, *SS2* is the size of its superset if it is the record for a tuple field, (*-1 -1* otherwise).

A part of the file TravelGlobals is given in Figure 14.

```
People :
Type : set none Person
Size : 1000 1000
SKS : NIL
Range : - 1 -1
SizeSSet : -1 -1

Students :
Type : set none Stdudent
SKS : NIL
Range : -1 -1
SizeSSet : -1 -1

Families :
Type : set none Family
Size : 75 75
SKS : NIL
Range : -1 -1
SizeSSet : -1 -1

Flatten(Families) :
Type : set none Person
Size : 202 202
SKS : People
Range : -1 -1
SizeSSet : -1 -1

Choose(Families) :
Type : set none Person
Size : 3.39 3.39
SKS : Flatten(Families)
Range : -1 -1
SizeSSet : -1 -1
```

FIGURE 14. Shows the format of the file TravelGlobals

7.0 Further Work

We need to extend the work described above to estimate the sizes of some other query operators of AQUA not covered above. The following is a list of the work still to be done.

We do not know how to handle multisets yet. We do not know how to estimate the sizes of the output sets returned by the query operators *convert*, *dup_elim* (when it is applied to a multiset), *union* (when it is applied to a multiset), *additive_union*, *intersect* and *diff*.

For set operators, we do not know how to estimate the sizes of sets returned by AQUA operators *fold*, *LFP*, *join* and *outer_join*.

For the operator *dup_elim*, the size estimation technique does not give a good upper bound. The formula in [3], does not work.

Consider the query Q = *(lambda (x, y) sp_i) (Set)*. It gives the size of the resultant set to be *R.size*,

$$R \cdot size = p - \sum_{k=k1}^{k2} (p-k) \cdot Prob \cdot (p-k \cdot slots \cdot are \cdot empty)$$

where p is the approximated size, and $p = k1$. $k1 = max\ (sp_i)\ i = 1...n$ where each sp_i is of the form $x.tail = y.tail$ and $k2 = product\ (sp_i)\ i = i$ .....n

The formula for calculating the value of *Prob(p - k slots are empty)* always gives a value of 0 and hence the upper bound is always p.

The formula given by Mark Nodine to calculate the size of the set for dup_elim queries of the form Q = ***dup_elim*** *(lambda (x, y) x.tail = y.tail) (Set)*. Let the size of the resultant set be R

$$R \cdot size = \langle \frac{1}{Set \cdot size} + \frac{Set \cdot Size - 1}{Set \cdot Size \times Set \cdot tail \cdot Size} \rangle \times Set \cdot Size$$

This under estimates the size of the set and so we do not implement it.

For the AQUA operators *select, forall, exists* and *mem*, we do not have a formula for calculating the selectivity if we allow field values to be compared to one another or if a field value is being compared to a string or a real.

For the operator *flatten*, if we do not know the size of the superset of the *choose* record of the input set, we do not know how to calculate the size of the resultant set.

For the operators *union, intersect* and *difference*, if we do not have any information about the relation between the input sets or the record for size of the intersection of the two input sets (maintained in the Schema Manager for certain sets), we do not know how to calculate the size of the resultant set. Besides we cannot calculate the sizes of sets if they use an *equality* rather than an *equivalence*. When we do not have information about the intersection of sets, we use a constant selectivity. This is defined in the file Globals.H as
`#define DEFAULT_INT_FRACTION 0.2`

The naming convention used when the operator apply is used on nested sets is misleading. Consider the query,

Q = ***flatten*** *(**apply** (lambda (f) **apply** (lambda (p) p.address) (f)) (Families))*
Here *Families* has type *Set[Set[Person]]*. In the above query we are trying to get the set of *addresses* at which any member of each family lives. We are calling this set *Families.address*. Since *Families* is a set of set of *Persons*, we are trying to find the set of addresses *Families.person.address* and not *Families.address*s as the type *Family* does not have any attribute *address*. Thus the above set is named incorrectly.

We also need to add another query operator *partition*. This operator takes a set and a user defined equality function *eq*. It then groups the elements of the input set into different equivalence classes according to the user defined equality function *eq*. The resultant set is then a set of sets where each set is the set of elements that are equal according to the user defined equality function *eq*. The user defined equality function is similar to the function defined for *dup_elim*. Also the size of the resultant set is calculated in a similar manner.

Consider a query Q = ***partition*** *(lambda (x,y) pred) (Set)*. Let the size of the resultant set be *R*.
Then *R.size* = *pred.size*. (This is calculated in a manner similar to *dup_elim*, Section 4.1.13).
Let the *choose* record of the set be *Choose(R)*.
Then *Choose(R).size* = *Set.size/pred.size*.
The original input set *Set*, is then set to be the smallest known superset (SKS) of this choose record.

8.0 Conclusion

Due to the complex nature of the data being manipulated by Object Oriented Databases, built-in heuristics cannot be used to optimize queries. Query Optimization is data dependant and hence cost estimations have to be used to decide whether a particular query rewrite is valid or not. An important component of this cost model of the optimizer is the techniques used for estimating the cardinalities of sets.

In this paper we have described the implementation of various techniques to estimating set cardinalities. We have used the notion of sets and supersets to estimate the sizes of query sets. It is seen that in order to perform efficient size estimations we have to maintain a lot of cost and size related information in the Schema Manager. We maintain the size of image sets to atleast one level and also the extent records for the intersection of any two sets which have the same supertype as long as one is not the superset of the other.

We have presented the techniques used to evaluate the sizes of sets formed as a result of various query operations. The sizes calculated can be used to estimate the cost of performing certain query operations. This can then be used by the optimizer to evaluate different optimization strategies.

References

[1] Mitch Cherniack. *Estimating Image Set Cardinalities*

[2] Gail Mitchell. *Extensible Query Processing in an Object-Oriented Database*. Ph.D. thesis, Brown University, 1993.

[3] Marian H. Nodine, M. Cherniack, T.W. Lueng. *Estimating Query Cost in an Object Oriented Database*

[4] Marian H. Nodine, F.B. Abbas, M. Cherniack. *The EPOQ Query Optimizer for Object-Oriented Databases, Design Specification*