Implementation of PDIndex & Query Optimization in *BODHI*

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Nagesh Bhattu Sristy

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

January 2002

Contents

List of Figures

Acknowledgements

I would like to thank my project guide *Dr. Jayant Haritsa* for giving excellent guidence through out the project. I am grateful to him for all the encouragement, suggetions he has given in the course of this project. I extremely liked and inspired by his working motto and the attitude towards any problem.

I would specially thank my batch-mates *leela*,*kanth*, *bandi* and *pankaj* for creating a nice working atmosphere in DSL. At times they would work like anything(crazy) which has nice impact on others as well. I thank *srikanta* for his useful suggestions in the project work. I would express special thanks to my colleagues *maya*,*vikram*, *amit* and *suresha* for helping out of our troubles. I really thank Subramanyam for arranging a good Satsanga (meeting arranged for discussing good) which provided us the motivation.

I can not express in words the encouragement given by my parents and the moral support given by my sisters. I heartfully thank each one of them for their efforts which made me to be in this position.

Finally, I thank All Mighty for serving me in all the above forms and motivating me all the time.

Abstract

BODHI [8] is a bio-diversity database system, built around an Object-Oriented micro kernel *Shore*[1] and *lambda-DB*[3] a rule-based query optimizer. *BODHI* provides three different types of services to the users namely Object Services, Spatial services and Sequence services.

Object services module maintains indices and provides efficient access to the objects present in the *BODHI* system. Object Oriented systems have nested aggregation relationships among different classes. Path Dictionary index is one of the well known indexing schemes used for indexing aggregation hierarchies. In this report we discuss complete implementation and integration of Path Dictionary index with the *BODHI* system. A statistics framework is developed for queries involving Path Dictionary index. Performance of Path Dictionary index and optimizations further possible are discussed in detail using the taxonomy data model in *BODHI*.

Chapter 1

Introduction

BODHI is a Bio-diversity database system intended for storing plant bio-diversity information. It is based on object-oriented modeling approach and is built around an Object-Oriented micro kernel *Shore* which provides the basic object storage services. The queries which are expressed in Object Query Language(OQL) are optimized using a query optimization framework, which is built around a rule-based query optimizer called lambda-DB. *BODHI* seamlessly integrates diverse types of data in Bio-diversity queries like taxonomic characteristics, spatial distributions and genetic sequences. A typical query asked will deal with all the three components of the system and accesses huge amounts of data. So query optimization and maintaining indices is very crucial to handle these complex queries. So efficient access methods specific for each type of data are developed such as *Path-Dictionary* for accessing taxonomy data, R*Tree and Hilbert-RTree for accessing spatial data and *pyramid-tree* for multi-dimensional data.

Different classes of an object-oriented system are related to each other through aggregation hierarchies or inheritance hierarchy. Object-Oriented systems use relationships to form a nested aggregation hierarchy. These hierarchies can be arbitrarily long. Queries which ask for objects along the hierarchy given an object of higher level in the hierarchy are expressed as path expressions or simply paths. For illustrations we will be using the paths present in the taxonomy data model [10] of *BODHI* system through out this report. Figure 1.1 shows a partial view of the data model. Two of the paths in this hierarchy are

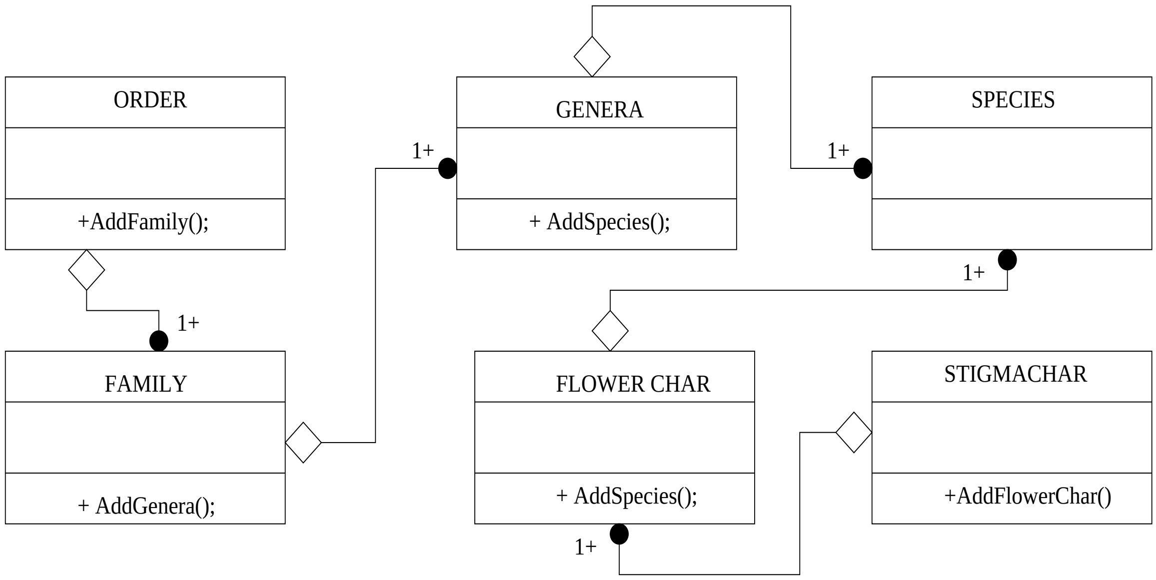


Figure 1.1: Taxonomy Data Model

- *Species.FlowerChar.Stigma*
- *Order.Family.Genera.Species*

The relationships can be *1:1 or 1:N or N:1 or M:N*. In this model we have 1:N and N:1 relationships. In this model a set of Species objects have a common Flower characteristic. A set of Flower characteristics have a common Stigma characteristic. This relationship is depicted by the first path. Similarly a set of species belong to a single genus, a set of genera have a common family and a set of families are nested in a single order. This is represented by the second path. We call the the classes to the left of a given class in the path as parent classes and the classes to the right of it as children. We use the terms parent and child to refer to the starting and ending classes of a path. If such a path appears in a query then the access methods serving their purpose must return a single child object or a set of children objects for each parent object. This is called forward traversal. Similarly fetching parent objects of a given child object is called backward traversal.

Since the objects themselves may spread over different disk pages, accessing them by using a nested loops evaluation requires tremendous effort and this way of accessing the deeply nested attributes is proved to be quite inefficient by many indexing schemes. In object-oriented systems every object has a unique identifier(OID). With in the index this OID is used to represent the object instead of the entire object. The indexing scheme used should be efficient for both forward and backward traversals of the hierarchy. Path Dictionary index is an efficient indexing scheme which does this with minimum storage overhead. Since it facilitates both forward and backward traversals efficiently, PDIndex opens up many opportunities for query optimization. Since most bio-diversity information are naturally related by such aggregation hierarchy, PDIndex is the key component to maintain the Object Services module of *BODHI*. In this report we discuss implementation of PDIndex and integration of PDIndex with query processor and different optimizations considered using its features.

1.1 Organization of Paper

The rest of the report is organized as follows. Chapter 2 describes the basic data structure of Path Dictionary Index. The implementation and complete integration of Path Dictionary with *BODHI* system is presented in Chapter 3 and 4. Optimizations which are explored by using PDIndex to its full potential and implementation details of these optimizations are discussed in Chapter 5. In Chapter 6 we present the descriptions of queries along with performance results. In Chapter 7 the present status and future directions to enhance system performance are discussed.

Chapter 2

Indexing for Aggregation Hierarchy Traversal

The queries that can be asked on aggregation hierarchy can be classified into the following categories:

- TP : The target class is an ancestor of the predicate classes.
- PT : The target class is a nested class of the predicate classes.
- MX : The target class is an ancestor of some predicate classes and nested class of some predicate classes

where target classes are the classes from which objects are retrieved and predicate classes are the classes involved in predicates of the query. The first two are of more interest to us. The first type of query naturally requires forward traversal of path. The second type of queries require back-ward traversal of path. If O is an object of Species then O.FlowerChar.Stigma represents a path. If such a path appears in the predicate then its natural order imposes forward traversal of paths. But the back-ward traversal might also be needed if we have the query plan of fetching Stigma objects first and then traversing the path back-wards and fetching the Species objects. For detailed discussion of these queries see [4]. Standard database solution for this problem is to build an indexing scheme. The well-known indexing schemes in OO literature are Path Index [5], Multiple Index [6], Field

Replication technique [7]. The first two techniques are based on keeping reverse(backward) links along the hierarchy. Due to these reverse links they can efficiently support TP queries. Since PT queries implicitly suggest for-ward traversal of hierachy, they are not benefitted by the reverse links that are maintained. Field Replication techniques as their name suggests ask for replicating frequently accessed fields in higher level classes of the hierarchy. But the problems with these techniques is their scalability and their dependence on the domain specific knowledge such as what are the frequently queried attributes. The types of queries supported is also limited.

If both ways of traversal are efficient then we can effectively use this in optimizing the queries that are being asked by transforming the query to use back-ward traversal if the query is asking forward or vice versa. The way in which this transformation is done depends on which type of query and the parent-child relationship of the path *1:1 or 1:N or N:1 or N:M*. It is further discussed in the optimizations section. The indexing scheme which supports both the types of queries efficiently is Path Dictionary Index. It is the best aggregation hierarchy indexing scheme in Object-Oriented world till date. One of the main reasons is the isolation of index structure from the physical Objects. Hence like the B^+ tree index in relational world it can be created and destroyed on the fly which is the most important aspect from maintenance point of view. It scales well with the number of attributes in the nested class over which we want to maintain indexes because of it's neatly seggregated architecture. It is proved to be best in terms of storage overhead that will be added by indexing schemes. It has three components.

1. *Path Dictionary*
2. *Identity Index*
3. *Attribute Index*

The architecture of the system is given in Figure 2.1. Here Path Dictionary is the central data structure on which Identity index is built. Attribute indexes are used for predicate evaluation.

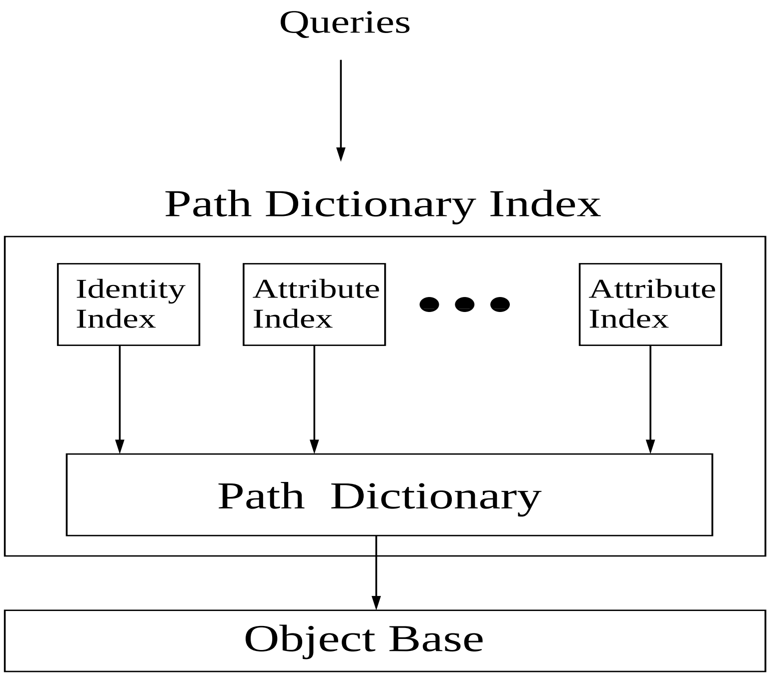


Figure 2.1: Path Dictionary Index

This problem has not been looked at in the context of Object-Oriented databases after Path Dictionary technique. But it is looked at in semi-structured databases but the problem is different from our problem since in semi-structured indexing the schema is not fixed. Here in this problem we have to build an index for a particular path for which schema already exists and it should efficiently support queries only for this path. So Path Dictionary Index is chosen for aggregation hierarchy indexing in *BODHI* system.

2.1 Path Dictionary

It is the central data structure which stores the OIDs of objects in a tree structure so that, given an object of one class, an object of another class along the hierarchy can be retrieved efficiently. It is implemented as a collection of s-expressions where each s-expression is a single stream of objects along the path.

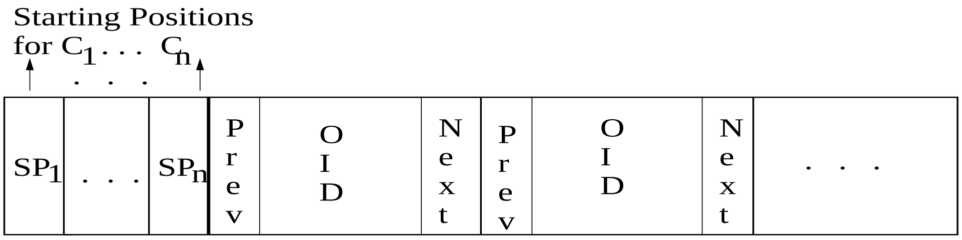


Figure 2.2: Data structure of an s-expression.

2.1.1 S-Expression Schema

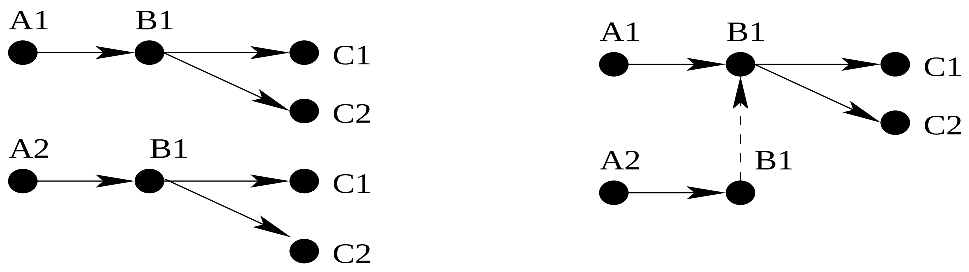
S-Expression represents an individual stream of OIDS of objects along the path. It is a tree structured representation of objects along the path. The s-expression for the path $C_1, C_2, ..C_n$ is defined as follows:

- $S_n = \theta_n$, where θ_n is the OID of an object in class C_n or null.
- $S_i = \theta_i(S_{i+1}[, S_{i+1}])$ $1 \leq i < n$, where θ_i is the OID of an object in class C_i or null, and S_{i+1} is an s-expression for the path $C_{i+1} \dots C_n$.

S_i is an s-expression at i^{th} level in which the list associated with θ_i contains recursively the OIDs of all descendant objects of θ_i. Except for the objects in C_n, every object on the path has a descendant list, which may be empty. So every object of class C_1 or C_2 or ... or C_{n-1} can have a set of objects of the immediate descendant class associated with it. This essentially defines a *1:N* relationship.

2.1.2 S-Expression Data Structure

Figure 2.2 illustrates the data structure of an s-expression for $C_1C_2 \dots C_n$. SP_i in the header points to the first occurrence of θ_i in the s-expression. Following the SP_i fields is a series of $< Prev, OID, Next >$ entries. The $Prev$ and $Next$ values associated with θ_i points to the previous and next occurrence of θ_i in the s-expression. Using the SP_i, $Previous$ and $Next$ values, we can easily trace the nested relationship among the objects in an s-expression. For example, to obtain the descendant list associated with θ_i, we simply collect the OIDs stored after θ_i until we reach the OID pointed to by θ_i's $Next$ value.



(a) A1(B1(C1,C2)
A2(B1(C1,C2)

(b) A1(B1(C1,C2)
A2(B1)

Figure 2.3: Modified structure of s-expression

Due to inherent tree structure of s-expression, it stores the parent object only once in the index and there by achieving good compression. So it works well for *1:1 or 1:N* relationships. Since the basic structure allows *1:N* relationships we can uniformly switch the order of classes in an *N:1* relationship and store it as *1:N* in the s-expression. Since each object is stored only once in s-expression the space occupied is optimal.

If *M:N* relationships are stored in this structure then the parent objects might be sharing the same set of objects nested under them then they will be replicated twice in the Path Dictionary. So the above mentioned optimality of storing every object only once does not hold here. So for *M:N* relationships the storage cost is not optimal. One option to minimize the storage overhead is to compress the s-expressions for *M:N* relationships by allowing sub s-expressions (by creating a new s-expression for sub-paths if they found to be redundant) and putting indirect links from the s-expressions which are referring to this redundant sub-s-expression. This is illustrated in the Figure 2.3. Here ABC is a path and A1(B1(C1,C2)) is one instance of such path and A2(B1(C1,C2)) is another instance of the path. Since the relationship of path is *M:N* the sub path B1(C1,C2) is replicated in two s-expressions. So it can be compressed as in Figure 2.3(b) by putting an indirect link(dashed-line). Hence the storage required to store s-expressions is minimized. But the efficiency of answering the queries asked is effected.

In the previous scheme of storage if a query asks for fetching all the children of a parent object then the position of the parent object in the s-expression is found and all its nested children of the queried child class are fetched by directly following next links of

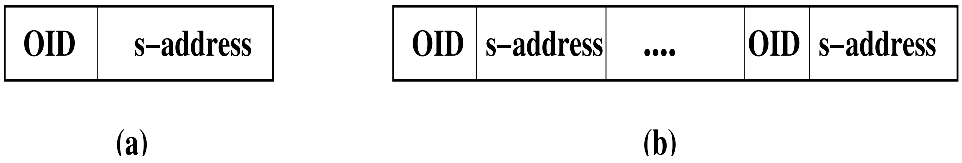


Figure 2.4: Data structure of leaf and non-leaf nodes of Identity Index.

the child class. But under the current scheme it is not not known how many indirect links of the inter mediate classes are present in the s-expression. So the only way to answer the query is to access each object along the path and see if there are any such indirect links and accessing the linked s-expressions if there are any. These sub s-expressions will be on a different disk page. So the access cost will increase when compared to the normal way of storing without worrying about such redundancies. But the accessing cost, which is more critical than the storage cost, will increase. Not only that it will result in small s-expressions and so external fragmentation would become a problem. So we have chosen the same implementation for even *M:N* relationships.

2.2 Identity Index

It indexes all the objects in the Path dictionary and says in which portion of the path dictionary namely in which s-expression a given OID is present. So identity index is an interface to the PDIndex to the rest of the system. If a path occurs in the query then the parent object is in the identity index and all the s-expressions in which this parent object is present are searched for the queried child class objects. A OID can be present in multiple s-expressions if the relationship is a *M:N*. Figures 2.4(a) and 2.4 (b) show the structures of leaf node record and non-leaf page for B^+tree implementation of identity index respectively.

2.3 Attribute Index

It indexes over the OIDs present in the Identity index. These are B^+ tree indices built over OIDs for different attributes of the given class or it can even be for a complex method which is expensive to compute. We can add as many indices as required over the Identity index depending on the amount of space constraints.

Chapter 3

Implementation Details of PDIndex

In this chapter we will see the issues in implementing the basic data structures of PDIndex and the traversal algorithms which should be provided by the PD-Server as services to the rest of the system.

3.1 Basic Data Structures

PDIndex is implemented as a separate *VAS* server serving the requests of different *VAS* clients. The client and server interface with each other through rpc interface. The Identity index is a non-unique B^+tree. The B^+tree interface of *Shore* is used for this purpose. Each Identity record has the OID and the corresponding s_expression id in which it is present. S-Expressions are implemented as records of a single file which is also provided by the *Shore* in its *VAS* interface.

3.2 Creation

Creation of *PDIndex* is done by giving all the objects along the path to the *PDIndex* client. The client will do the creation by making appropriate requests to the server. After all objects are inserted into the pd-index the details such as statistics, name, path are noted in the statistics module of *BODHI*. These are used later when a query is given to

INPUT : Parent object O_1,Path $C_1, C_2, ..C_n$
OUTPUT: set of children objects O_i
Algo: GetChildren

1: if O_1 is in identity index then
2: get s-expr s_i in which it is present.
3: else return error;
4: get the positions of O_1 and O_1.next() in s_i
5: Fetch all objects of class C_n between these two positions
6: if O_1 is in s_i after O_1.next() then
repeat Step2 and Step3.
7: else if O_1 has another entry in identity index then
8: get s-expr s_j containing O_1 and goto step2;
9: else return finished;

Figure 3.1: Algorithm to Fetch all children of a parent

find whether a *PDIndex* exists for a given path. A directory of all the available indices names are maintained. Initial code base for s-expressions, pdindex structure is available in *BODHI*. This is part of previous work [9] in *BODHI*.

Here the statistics maintained are the average size of collection attributes along the path and the number of such sets, number of objects present in the Path Dictionary for the attributes which are not of type collection. For collection attributes a fan-out of disk space is also maintained. This is extracted from the disk page id portion of OID.

3.3 Path Traversal

As mentioned earlier there are two types of traversals the PDIndex has to support are forward and backward. For example if the path is *Species.FlowerChar.Stigma* then we may have queries to the PDIndex asking get the Stigma object of the given Species object, or get the Species object of the given Stigma object. The associated objects of the given object can be a set of objects rather than just a single object. The basic algorithm for forward traversal is shown in Figure 3.1.

Similar algorithm is written for the backward traversal. Input for the backward traversal is a child object and output is a set of parent objects which are having this object as one of their children objects. As we can only send limited number of objects through rpc client-server interface though set of objects are associated with a given object along the path we can only return single OID or fixed number of OIDs for each rpc call. So the state of traversal must be maintained on the server side so that the server returns the next OID of the result only when the client asks for it. So these traversals are implemented as iterators storing the state of the traversal. This will also reduce lot of function call overhead. If the path is producing only a single object then they should not suffer the overhead of starting iterators and closing. So the same routines are implemented both as stateless and state-full.

Chapter 4

Integration of PDIndex with BODHI

In this chapter we will see the issues involved in integrating PDIndex with BODHI system. In the next section details of query flow through the BODHI system are given.

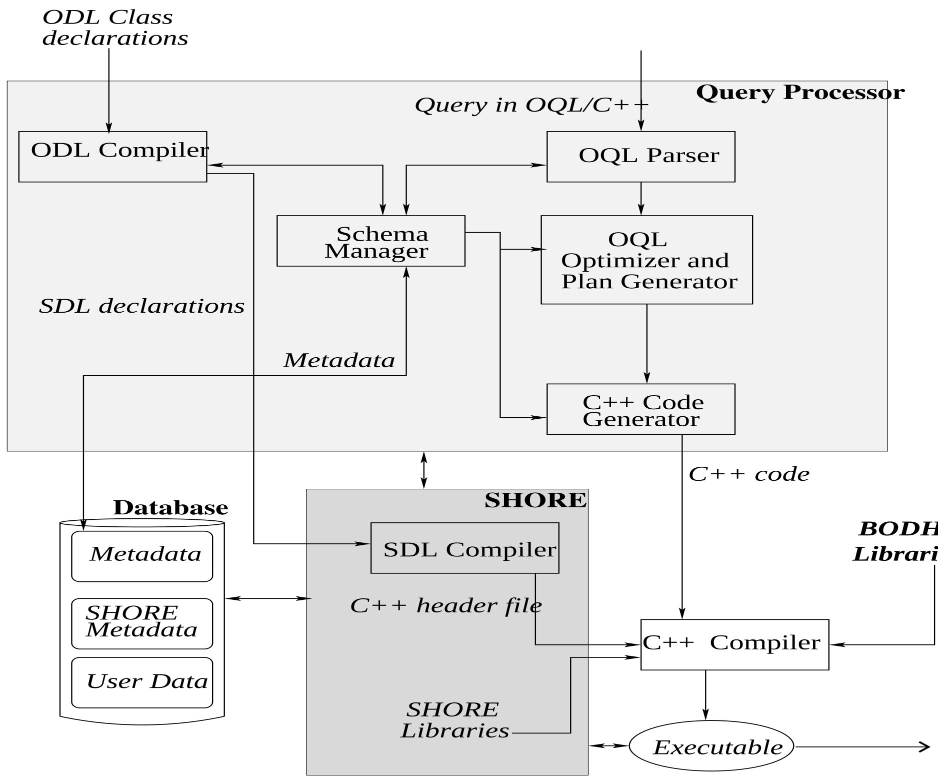


Figure 4.1: Architecture of Query Processor in *BODHI*

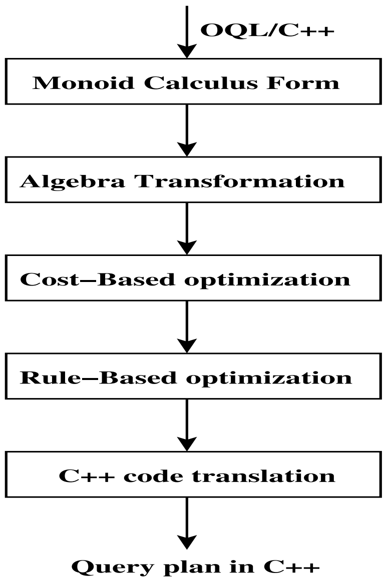


Figure 4.2: Query flow in *BODHI*

4.1 Query Flow in BODHI

As mentioned before BODHI uses a rule-based query optimizer called *lambda-DB* [3]. Figure 4.1 gives a proper view of architecture of query processor. *Lambda-DB* takes the queries in OQL form and generates the query plans in C++. This C++ code should be converted into object form for getting executables of the query. The schema is defined using Object Definition Language(**ODL**)[2]. The ODL declarations are converted into SDL (*Shore* Data Language), a data definition language provided by shore storage manager. The *Schema Manager* of the query processor extracts the meta information and maintains it in the database. All further optimizations and type checking of queries depend on this.

Figure 4.2 shows the flow of query. The query processor optimizes the given query by converting it into several intermediate forms. First it converts the given query into a monoid comprehension calculus form. This calculus framework allows various collection types to be intermixed with each other. It normalizes the query into a form that opens up opportunities for more optimizations such as whether to use sequential scan or indexed access or sorting of the intermediate results. This normalized calculus expression representing the given query is then transformed into algebraic form. During this transformation all path-expressions in the query are transformed into joins along the path as the default mechanism followed is nested-loops evaluation. The inter-mediate algebra is similar to relational algebra. The algebra expression is composed of a set of algebra operators. At each operator the input stream under goes some transformation and an output stream comes. The semantics of the transformation are dependent on the type of operator. They are

- **join** X $\bowtie_p$ Y, joins the two streams X and Y using the join predicate p.
- **selection** σ_p(X), selects all elements of X that satisfy the predicate p.
- **unnest** μ_p^{path}(X), returns all collections of pairs(x,y) foreach $x \in X$ and foreach $y \in$ *x.path* that satisfy the predicate *p(x,y)*.
- **reduce** $\Delta_p^{\oplus/e}$(X) collects the values *e(x)* for all x $\in$ X that satisfy the *p(x)* using the accumulator $\oplus$.

This algebra expression is treated as a set of joins. The order in which these joins should be performed is decided by a greedy join-ordering algorithm called GOO[11]. The algebra expression is converted into physical plan by a rule-based optimizer. This physical plan is converted into equivalent C++ code which when executed fetches the desired tuples from the storage manager. For complete details of query optimizer see LambdaDB[3].

4.2 Integration of PDIndex Creation with *BODHI*

In OQL there are constructs for creation of index. In the previous work done [9] these were extended to hierarchical indices also. Current work includes rest of changes needed to be done for optimal storage of the path. Given an arbitrary path expression using these constructs the path expressions must be analysed and they may need to be sub-divided into small paths for the storage to be efficient. These paths might be one of 1:1 or 1:N or N:1 or N:M. As mentioned before the s-expression schema naturally allows 1:1 or 1:N. So it may be needed to switch the order of objects coming in order to store them in Path-Dictionary efficiently. Such a translation is very much essential as storing in the normal way not only increases the storage cost but also the access cost as it results in very thin s-expressions and during access it results access of lots of disk pages due to fragmentation. To create a PDIndex on the path Species.FlowerChar.StigmaChar the OQL query is the following.

- *% create pdindex "SPECSTIG" Species.FlowerChar.StigmaChar*

As the relationship of the path Species.FlowerChar.StigmaChar is N:1 the order of Classes is flipped while storing the OIDs in the s-expressions. The order would be StigmaChar.FlowerChar.Specie

Code is added into C++ code of query for interfacing with the PDIndex for Creation and insertion of data into the Path dictionary. For this path to be useful in the subsequent queries this pathname and details of it are stored in the schema manager. For query optimization the statistics about the current path are to be retrieved and stored in the statistics module along with the other details about this path.

4.3 Integration of Path Traversals with BODHI

As mentioned previously the given OQL query is transformed into a calculus intermediate form. During the normalization, the calculus form of the query is divided into a set of basic blocks. At the end of normalization all the path expressions are converted into joins along the path. For example the query:

Query1:

- *% foreach v in select * from s in Species where s.FlowerChar.StigmaChar.StigmaForm = LINEAR_SIMP*

is translated into an algebra expression whose physical plan is shown in Figure 4.3. Along each line there will be a stream of objects and if two streams join then outcome is a single stream formed by nested loop of one stream over another subjected to the join predicate. So just before such a translation happens, all the paths in the query are checked for the usability of PDIndex instead of the path-join. If such PDIndex exists then it must be used instead of the join, since in the case of PDIndex the traversal is over OIDS. In the

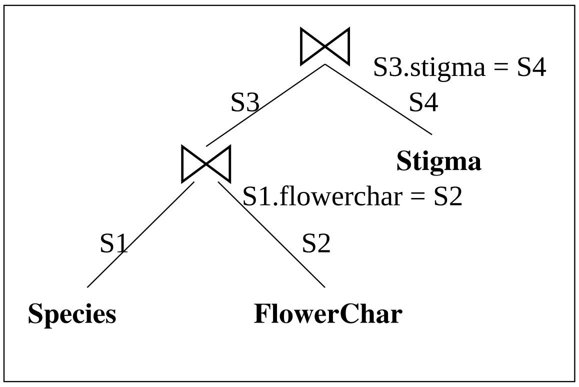


Figure 4.3: Physical Plan for pathjoin

case of path-join it is actual traversal of the path along the objects rather than their IDs. So even if the path is of length 2 it makes a tremendous improvement in the performance of the query if we use PDIndex as it acts as a short circuit between the first object and last object of the path.

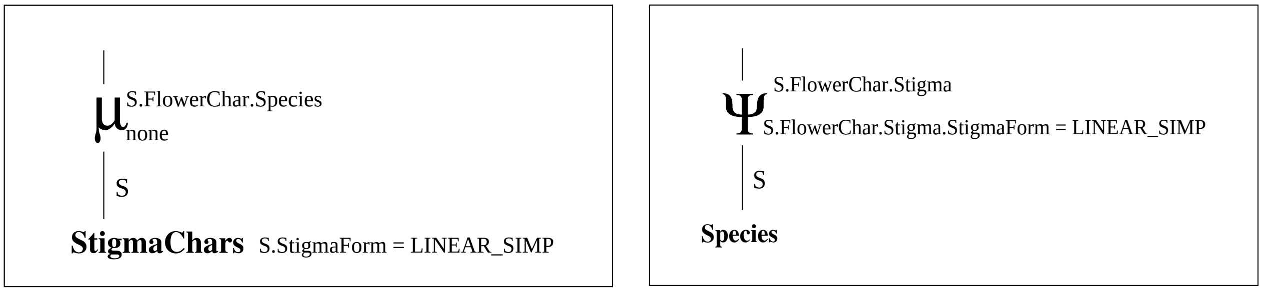
If a path is found to be accessible through PDIndex then all its incarnations in the query are replaced by a single variable representing the last class objects along the path. A new operator pditerator is introduced at the calculus level. This is used as carrier of information about a particular path such as indexname, input stream, output stream.

This path might appear in any of the *from clause* or *project clause* or in the *where clause* (predicates) of the query. If it is in the *from-clause* then each object coming out of the path should be joined with it's parent object and the result should be all parent objects joined with all their children along the path. For example a query which fetches all the Stigma characteristics and the associated Species which are sharing this Stigma characteristic is as follows:

Query2:

- *% foreach v in select * from s in StigmaChars, s1 in s.FlowerChar.Species;*

Here each object of StigmaChar is joined with the objects of Species coming out of the path. If it is in the other parts (*where clause or project clause*) of the query then it suffices to fetch the children objects and the extra join operation with the parent object is not required. For example in Query1 the path Species.FlowerChar.StigmaChar appears in the



(a) Query1 (b) Query2

Figure 4.4: Algebra Expression

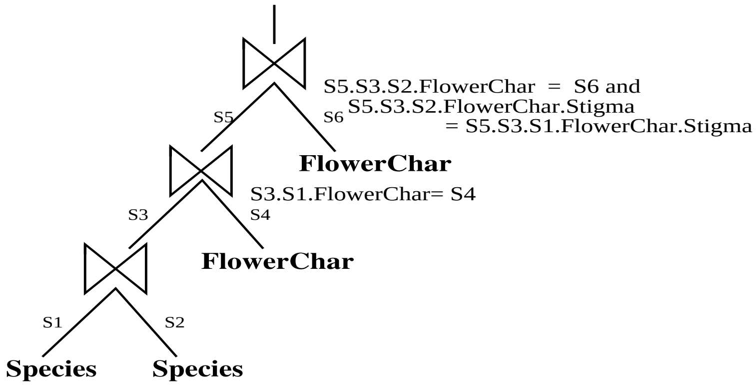
where clause. These are the two cases which needs to be distinguished while converting the calculus into inter-mediate algebra.

While converting calculus expression into inter-mediate algebra, an algebra operation is chosen for each calculus operator. The set of operators available at the algebra level are unnest, nest, reduce, join, select. The operators that are useful for path-expressions are unnest(μ) and map(Ψ). They are defined as:

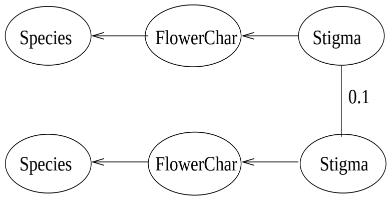
- μ_p^{path}(X) returns the collection of all pairs of tuples(x,y) foreach x $\in$ X and y $\in$ x.path, that satisfy the predicate p(x,y).
- Ψ_p^{path}(X) returns the collection of all pairs of tuples(x,x.path) foreach x $\in$ X and satisfying the predicate p(x,x.path).

Here X is input stream of tuples and x is a variable which iterates over that stream. x.path produces a collection. If the path is present in *from clause* it is transformed into unnest. The algebra expression equivalent to the Query 2 is shown in Figure 4.4(a). If the path is present in the *where clause* or *project clause* the semantics of the operator are different. So instead of unnest a new operator called map(Ψ) is used. Map is introduced at the algebra level of query optimizer specifically for PDIndex. For a detailed discussion of the semantics of these operators see lambda-DB [3]. The algebra expression equivalent to Query1 is shown in Figure 4.4(b).

In the next phase of query optimization, a graph is constructed for the algebra expression. The nodes of this graph are unary operators of algebra and undirected edges



(a) Algebra Expression



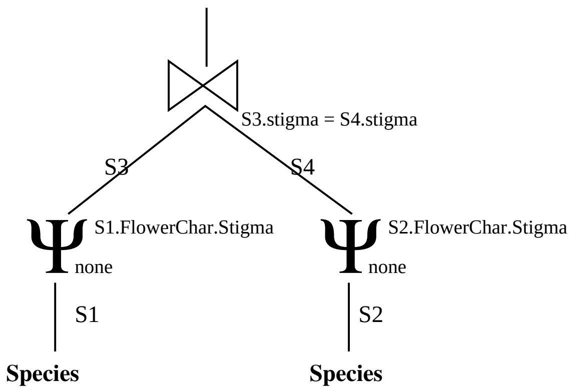
(b) graph

Figure 4.5: Nested-Loops evaluation

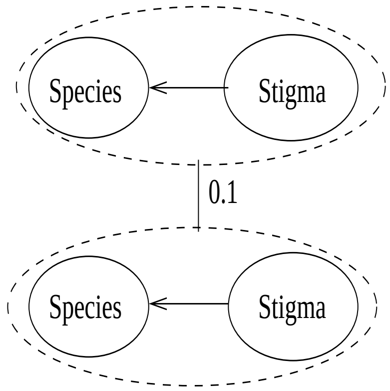
represent joins between two different streams and they are labelled with the selectivity of the join. Directed edges represent dependency of one stream over the other stream. For example in Figure 4.5(b) the stream FlowerChars depends on Species. The order in which these joins are to be performed is decided by a cost-model. A join ordering algorithm which is called GOO[11] is used for ordering the joins. For example for the following query if nested-loops evaluation is followed then the graph will be as shown in Figure 4.5(b).

- *% foreach v in select * from s1 in Species,s2 in Species*
  where s1.FlowerChar.Stigma = s2.FlowerChar.Stigma

Here the order of the joins is decided as shown in Figure 4.5(a). Since the GOO algorithm is based on the heuristic which does optization locally (i.e it does not guarantee a globally optimal plan), the join of species is made first which is not the best way to do this. Here an optimal plan should fetch the stigma objects from each species object and then make the join. If the PDIndex is used for traversing the path-expression then path-expression will be converted into map operator or unnest operator. As map operator acts just like projection of an attribute from the given object, it is not considered as a seperate node while building the graph. The join-graph is as shown in Figure 4.6(b). Hence in the query plan shown in Figure 4.6(a) it is possible to push the the map operators corresponding to path-expressions into the join. The optimized algebra is later converted into physical



(a) Algebra Expression



(b) graph

Figure 4.6: Using PDIndex

plan by a rule-based optimizer. Finally if the operator pditerator is seen in the final query plan then appropriate code for interfacing PDIndex is added into the C++ code.

Chapter 5

Optimization of Queries

The basic idea behind any optimization of queries over path dictionary is to traverse the path in the back-ward direction. For this the query must be rewritten. If the query is as follows:

Query1

- *% foreach v in select * from s in Species where s.FlowerChar.Stigma.StigmaForm = LINEAR_SIMP*

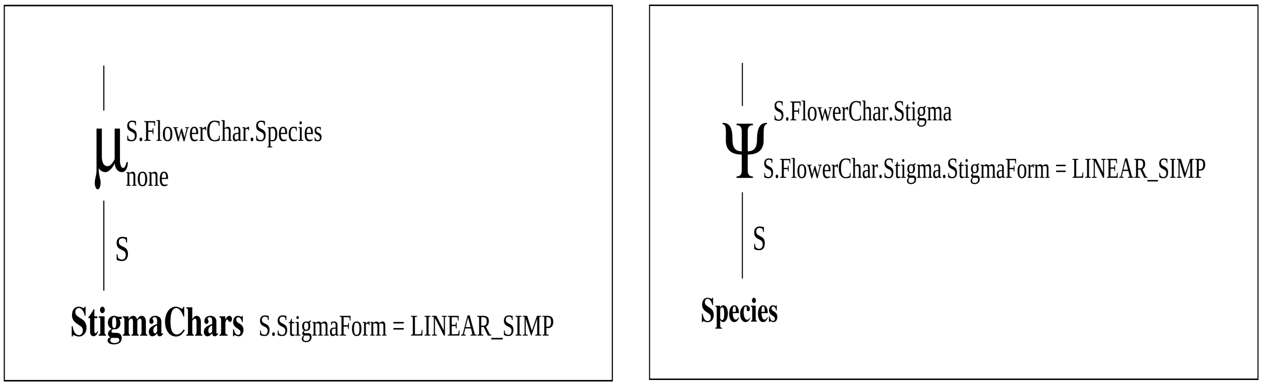
then it is transformed into algebra form in Figure 5.1(a). This query can be rewritten as in Figure 5.1(b). Such a rewrite is very effective when the predicate has the least selectivity among all the other predicates.

Among Bio-diversity community the queries asked usually involve joins of few paths such as get all the pairs of plant species which are having the same stigma characteristics. This query is expressed in OQL as:

Query2

- *% foreach v in select * from s1 in Species,s2 in Species where s1.FlowerChar.Stigma = s2.FlowerChar.Stigma*

Query2 is translated into Algebra as shown in Figure 5.2(a). An alternative way of doing the same is by fetching objects of Stigma characteristics for the first stream of



(a) normal (b) optimal

Figure 5.1: Algebra Expression

plantspecies and getting the species objects along the path which have the same Stigma characteristic. The algebra expression equivalent of this rewritten query is 5.2(b).

To test this out the given query is rewritten at the algebra level. It is rewritten if and only if the the cost of optimization is less than the cost of usual plan. The cost of optimized plan is

- *cost = fan-out * avgcollsize * card(parent) * selectivity(pred) * cost(path_access)*

Here,

fan-out is the average number of pages over which a single set of children spreads.

avgcollsize is the average size of set of children over the path.

card(parent) is the cardinality of the parent extent of the path.

pred is a portion of predicate which depends only on variables iterating over parent extent.

cost(path_access) is network overhead for each path-dictionary rpc call if path dictionary is remotely served.

In Query1 *Stigma.FlowerChar.Species* is the back-ward traversal of path Species.FlowerChar.Stigma. The original path is N:1 and hence the back-ward traversal of the path is 1:N i.e a set of Species objects are associated with each Stigma object.

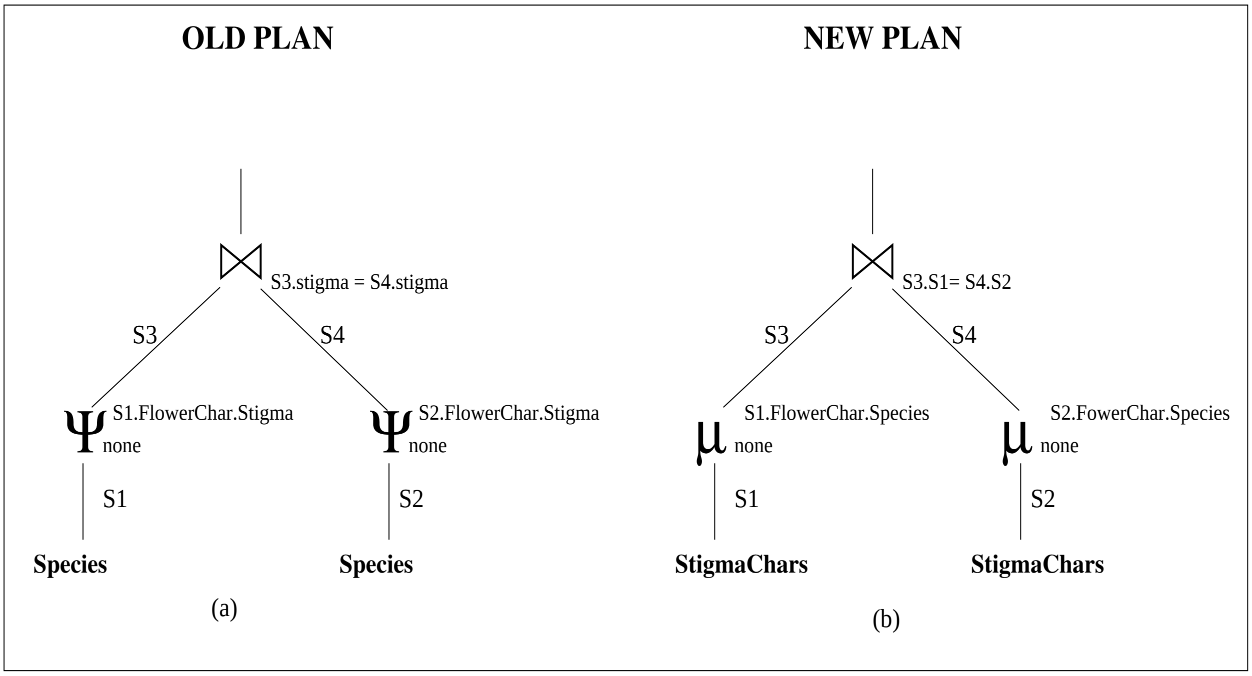


Figure 5.2: Algebra Expression

The cost of optimized plan for Query1 is fan-out * avgspecies(Stigma) * card(Stigma) * selectivity(s1.stigmaform = LINEAR_SIMP) The cost of original plan is

- *cost = card(child) * cost(path_access)*

For Query1 it is card(Species) * cost(path_access).

Chapter 6

Performance Results

There is no standard benchmark for testing path expressions in Object Oriented databases. OO7 benchmark which is the standard benchmark for Object-Oriented databases does not have queries that access lengthy paths. The queries in OO7 are primarily intended for testing the performance of database engine performance such as how efficiently it is scanning a set of objects. So PDIndex has been tested using the taxonomy data model in BODHI. This model has paths of length 3 and 4. The path tested is Species.FlowerChar.StigmaChar. There are two different sizes of datasets available. In the first dataset there 255 species objects and 5 different Stigma characteristics are shared among these 255 Species objects. Second dataset consists of 5000 species and 5 different Stigma characteristics shared among them. In this path there are two N:1 relationships. So the total path is N:1. This path is stored in the PDIndex starting from StigmaChar objects at the root of s-expression,FlowerChar and Stigma Char objects are the second and third level objects in the tree structure respectively. This is the most optimal possible way of storing the s-expression.

6.1 Performance of PDIndex

Query 1: Create the path index for path Species.FlowerChar.StigmaChar

- % create path index SPECSTIGMA on Species.FlowerChar.StigmaChar;

Query	Without PD	With PD	Optimized
Query 1	NA	5 sec	NA
Query 2	12 sec	1.9 sec	1.5 sec
Query 3	720 sec.	13.8 sec	13.3 sec

Table 6.1: Comparative time taken for queries with dataset1

Query	Without PD	With PD	Optimized
Query 2	5084 sec	3.8 sec	3.3 sec
Query 3	3 days	5714.9 sec	5712.8 sec

Table 6.2: Comparative time taken for queries with dataset2

Query 2: Select all the Species which have the LINEAR_SIMP stigma characteristics.

- *% StigmaForm s = (StigmaForm) LINEAR_SIMP;*
- *% foreach v in select * from s in Species where s.FlowerChar.Stigma = s*

Query 3: Select all pairs of Species which have identical Stigma characteristics

- *% foreach v in select * from s1 in Species,s2 in Species*
  where s1.FlowerChar.Stigma = s2.FlowerChar.Stigma

For the same queries Performance of Optimized Query plan is as follows. The optimized plan does not have much difference when compared to the normal query plan because the PDIndex server is run on the same machine as that of shore server. So the network overhead of rpc calls is zero. But it is not the case usually. Hence the Optimized Query plan has profound effect when the PDIndex server and Shore server are running on two different machines. This is reflected in the above tables. The column With PD indicates the time taken for normal query plan and Optimized is for optimized query plan.

Chapter 7

Conclusions and Future Work

In the current work Path Dictionary index is implemented and integrated completely with the BODHI system. Measures are taken to ensure that the access cost and storage cost of paths in PDIndex is optimal. In the process some of the bugs in lambda-DB are fixed as well. Different optimizations by rewriting queries are explored and tested out.

Lambda-DB uses fixed constants for the estimation of selectivity. This should be changed to use the statistics that are available in BODHI. Statistics framework for Spatial component should be developed for the query optimizer to make more sensible query optimization. Currently the query optimizer does not consider the option of materializing sub-expressions which are common for different portions of the query. This puts stringent restrictions on query optimization process. So the query optimizer should make use of materializing partial results during query optimization. Currently the query optimizer allows only one disk volume tobe used for the entire data. But the storage manager allows data to reside on multiple disk volumes. So the query optimizer should make use of this functionality of storage manager.

Bibliography

[1] M. J. Carey et al. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, May 1994.

[2] R. Cattel, editor. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann Publishers, 1994.

[3] L. Fegaras. An experimental optimizer for oql. Technical Report TR-CSE-97-007, University of Texas at Arlington, 1997.

[4] W. Lee and D. Lee. Path dictionary: A new access method for query processing in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(3), May 1998.

[5] E. Bertino and W. Kim. Indexing techniques for querying on nested objects *IEEE Transactions on Knowledge and Data Engineering*, Vol.1,No.2, June 1989, 196-214.

[6] D. Mair and J. Stein. Indexing in an object-oriented DBMS. *Proceedings of International Workshop on Object-Oriented Database Systems,1986,171-182.*

[7] E.J. Shekita and M.J. Carey Performance enhancement through replication in an Object-Oriented DBMS. *Proceedings of 1989 SIGMOD Conference Eugene, Oregon, May 1989, 325-336.*

[8] B.J. Srikanta and J. Haritsa. Design and implementation of a biodiversity information management system. In *Proc. of the 10th International Conference on Management of Data*, 2000.

[9] R. Tadimeti. Database habitats for biological species: Handling taxonomy data in Oshadhi. Master's thesis, Department of Computer Science & Automation, Indian Institute of Science, Jan. 2000.

[10] R. Vidya and J. Haritsa. Oshadhi : A biodiversity information system. In *Proc. of the 7th International Conference on Management of Data*, 1995.

[11] L. Fegaras. A New Heuristic for Optimizing Large Queries. In *International Database and Expert Systems Applications Conference*, pages 726–735, Vienna, Austria, August 1998.