The Framework for EPOQ - an Extensible Object-Oriented Query Optimizer

Solveig Bjørnestad

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-04
March 1994

The framework for EPOQ - an Extensible Object-Oriented Query Optimizer

Solveig Bjørnestad[1]

Technical Report No. CS-94-4
March 1994

1. Department of Information Science, University of Bergen, N-5020 BERGEN, Norway.
Email: solveig.bjornestad@ifi.uib.no. This work was done while the author visited the Computer Science Department, Brown University, February-August, 1993.

The framework for EPOQ - an Extensible Object-Oriented Query Optimizer

Solveig Bjørnestad, Department of Information Science,
University of Bergen

ABSTRACT:

This report describes the design and implementation of a framework for EPOQ - the architecture for an extensible, object-oriented query optimizer. Extensibility in this context is related to the types of objects that can be queried, the types of queries that can be optimized, and the optimization strategies that can be applied. For object-oriented data base systems different optimization strategies may be used on different data types, and the architecture makes this possible through the combination of several optimization regions that can apply different strategies.

This work was initiated to implement the overall design described in [Mitch93] where the emphasis is made on the overall architecture, and the description is on a relatively high abstraction level. How this could be realized was not emphasized. In this paper focus is made on the detailed design decisions that were made to ensure that the overall goals were met. The architecture is intended to be extensible to make it possible to build flexible optimizers that can be adaptable to new types of objects and experimentation with new optimization strategies.

A second goal has been to design a reusable framework for building query optimizers. To achieve this, focus has been put on designing the architecture as a set of interacting components that can be specialized whenever a new type of optimization strategy has to be implemented. Until the framework has actually been used to build example optimizers, it is hard to tell whether this goal has been achieved.

The report can also be used as a manual for implementation of optimizers based on this architecture. Currently this work is continued at Brown and hopefully we will gain more insight in how such a framework should be organized.

Acknowledgments

This report is the main result of my work as Visiting Scholar at the Computer Science Department, Brown University from February through August, 1993. First of all I want to thank Professor Stan Zdonik for inviting me to work with him. In addition to this work, and in part due to my participation in Stan's seminar on object-oriented databases, I learned a lot about object-oriented databases and query optimization. Also, this gave me an opportunity to get to know a lot of the graduate students at the department.

I want to thank Gail Mitchell for her patience while trying to explain the mysteries of the EPOQ architecture, and Ted Leung for our long discussions on different design and implementation strategies and encouragements at times when life seemed hopeless. Andrew McKeith, who implemented the query representation, was usually willing to make the modifications that I required, and more than often we had a long discussion over cups of coffee or ice-creams. While writing down this report, Misty Nadine has helped me by questioning my decisions, and I hope that by continuing this work she will be able to make the suggested design more elegant and more reusable.

My interest in Brown originates from reading about Peter Wegner's work on object-orientation, and I credit him for my visit. His encouraging comments and his interest in my progress was a great support. Mary Andrade who organized for me to come, made everything except the weather comfortable, and the rest of the staff were also very friendly. My office mates, Dina Goldin and Jian Xu, deserves thanks for their kind support.

During my stay I made a lot of friends that made the stay pleasant and adventurous by taking me to places that I would probably never have seen without them. Of these I would particularly mention Isabel Cruz, Ann Nicholson, and Roberto Tamassia with whom I went skiing, dining, shopping, and a lot more. A special thank goes to Ashim Garg for teaching me the mysteries of Indian cooking.

Contents

1 Introduction **2**

2 Requirements **2**
2.1 A short Overview of the EPOQ Architecture 2
2.2 Designing a Reusable Framework for EPOQ 3

3 Design **3**
3.1 The Model for the Architecture 3
3.1.1 Structural Relationships between Classes 4
3.1.2 Design Alternatives for Regions 7
3.1.3 Design Alternatives for Goals 8
3.1.4 Transactions 9
3.2 EREQ Query Representation 9
3.3 C++ Class descriptions 10
3.4 USL Class Library 10

4 Specification of the EPOQ Architecture **11**
4.1 Class Definitions 11
4.1.1 Class Region (Header file Region.H) 11
4.1.2 Class Goal (Header file Goal.H) 14
4.1.3 Class QuerySet (Headerfile QuerySet.H)) 16
4.1.4 Class GoalPackage (Header file: Region.H) 18
4.1.5 Class Log (Header file: Stores.H) 19
4.1.6 Class LogEntry (Header file: Stores.H) 21
4.1.7 Class TransList (Header file: Stores.H) 22
4.1.8 Class Trans (Header file: Stores.H) 23
4.1.9 Class Wm - (Header file: WM.H) 24
4.1.10 Class WmElement - (Header file: WM.H) 26
4.2 Location of files. 28

5 Testing the Architecture **29**

6 Conclusions and further work **30**
6.1 Problems 30
6.1.1 Requirements 30
6.1.2 Query Representation 30
6.1.3 Schema Manager 31
6.1.4 Architecture 31
6.2 Further work 31
6.2.1 QueryRep 31
6.2.2 The Architecture 32
6.2.2.1 Transactions 32
6.2.2.2 Class WM 32
6.2.2.3 New Abstract ControlRegion Class 32
6.2.2.4 Cooperation Between a Region and its GoalPackages 33
6.2.2.5 The generality of the Attain function 33
6.3 Concluding remarks 33

References **33**

1 Introduction

This paper describes the implementation of the EPOQ architecture. EPOQ is an extensible, object-oriented query optimizer. The overall design of the architecture is described in [Mitch93] and so is an algebra and a query representation for object-oriented queries. The implementation of this query representation is described in [MacKe93]. The EPOQ architecture communicates with the queries through a single interface, thus hiding the complexity of the query classes. Although this representation is not described here, some comments are made regarding the interaction with the representation.

This work is part of the **E**ncore **R**evelation **E**xodus **Q**uery (**EREQ**) project, a current joint project between Brown University, Oregon Graduate Institute, and University of Wisconsin, Madison. The project aims to produce an Object Oriented Query Language and two query optimizers, one at Brown and one at Oregon Graduate Institute. This report describes the work that has been done on the implementation of EPOQ - the optimizer being built at Brown.

A relational database stores only tables, and the attributes can be of a very limited set of types which can not be extended. An object-oriented database, on the contrary, is very flexible when it comes to the types of objects it can store. Because the objects in the database also contains functions, the queries that can be made are very flexible. These facts make it much more complicated to write optimizers for an object-oriented database system. Also, the heuristics that has been used for relational databases do not necessarily hold, since you can not always tell whether a property of an object is stored as an attribute or if it is actually computed using some other objects in the database. This would have an impact on the efficiency of the retrieval.

EPOQ is a suggestion for an architecture that permits extensions to the optimizer when new types of objects are added, or new types of queries are being generated. It is a goal that it should be as easy as possible to make extensions to the optimizer.

The topics dealt with in this paper are:

- The design goals for EPOQ and the design that is the result of applying these goals. During the work we had to make design decisions and the alternatives we saw will be described whenever appropriate.
- A discussion about trade-offs and what alternative design decisions that could have been made.
- A description of how this framework can be used to implement an optimizer.
- Further work.

2 Requirements

2.1 A short Overview of the EPOQ Architecture

The overall design of EPOQ is described in [Mitch 93]. Here we include a brief description of the architecture. The main goal of the design is extensibility, i.e. extensibility when it comes to the types of objects that can be queried, the types of queries that can be made, and what optimization strategies can be applied. These requirements stem from the fact that object-oriented databases are themselves extensible when it comes to the types and queries that can be used. The last type of extensibility has been added because it is realized that different types of objects may have properties that require different optimization strategies. One of the most important requirements of the architecture was to facilitate such an extensibility. If new object types are introduced in the database, it should be possible to extend the optimizer with new strategies whenever this is required.

To meet this goal, the optimizer is organized as a hierarchy, or rather as a directed acyclic graph (DAG), of regions. Each Region can attain one or several goals. For each of these goals, it has a GoalPackage. Depending on the goal it is asked to attain, the Region chooses a GoalPackage and a query to optimize. In the current implementation this query is the input query to the Region, however, any Region is supposed to select what query to use as the query to optimize. The only requirement is that it be equivalent to the input query or to a subquery. The Region may sequential call several goal packages based on some heuristics for

how the goal may best be attained.

The GoalPackage contains a set of "rules" that can be used to set up a plan for how the goal can be attained. To attain a goal the GoalPackage can use one or several of the Region's subregions. This is done by combining the execution of several subregions. These subregions have themselves one or more goals they can attain, and the GoalPackage may ask subregions if they think they can transform a particular query to attain a specific goal. A Log keeps track of the results when part of the plan is carried out. The plan may be modified based on these results.

A Region is responsible for deciding when its goal is attained, but to be able to make this decision it may be set up to use a predefined success function for a particular goal. This makes it possible to ensure that all regions have the same understanding of what it means to fulfil a specific goal.

All regions have the same interface to their calling Region and their subregions. A calling Region, or one of it's goal packages, determines when control is given to a subordinate Region. A subregion passes control back to the controlling Region when the transformation has been terminated. The Region also returns information as to whether the goal has been achieved or not. If a Region is unable to attain a goal, this may cause the caller to change its further plans.

The regions at the bottom of the Region DAG, the leaf regions, are simple transformations without any planning or control activity. Still the computations taking place within them may be quite complex. Actually, a separate optimizer may be called from within such a leaf Region.

2.2 Designing a Reusable Framework for EPOQ

The architecture for EPOQ should be easy to use to make new optimizers and to add new regions to an optimizer. The task should be reduced to focus on the planning and control system. The use of frameworks are generally considered to be a good way to achieve reuse of object-oriented components. A framework in this context means "a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes."[JoFo88]. That is, one identifies the main components within an application area, and designs the abstract classes of such a framework. Whenever a specific application within this area is developed, one can either use the existing classes or subclass any one of them.

All internal regions will have a planning activity, and because the set of subregions that are available will vary, this activity will probably be different for all regions. Separate classes will probably be needed for all regions. It is therefore a major goal to separate the tasks as much as possible, and reduce the number of methods that will have to be re-implemented. The abstract Region class at the top defines the interface of all regions, even in situations where it is difficult to describe a general behaviour. It is most likely that modifications will be made as one gains experience with the use of this framework. Maybe we will find that the differences between regions may be hidden in the goal packages.

3 Design

In this section we identify the main classes of the framework, and describe their role in the optimizer. A typical scenario is described using these classes. This is taken from [Mitch93]. The description of the individual classes will then be outlined, and the way these classes interact with the query representation classes. In this section I call the query class just Query. There is nothing in the architecture itself that limits the way this is represented or implemented. I only outline some requirements for this Query class.

3.1 The Model for the Architecture

We have already given a short outline of the general behaviour of the optimizer.The descriptions of the architecture and the scenarios in [Mitch93] has been used to identify the classes. From this description we can extract the following set:

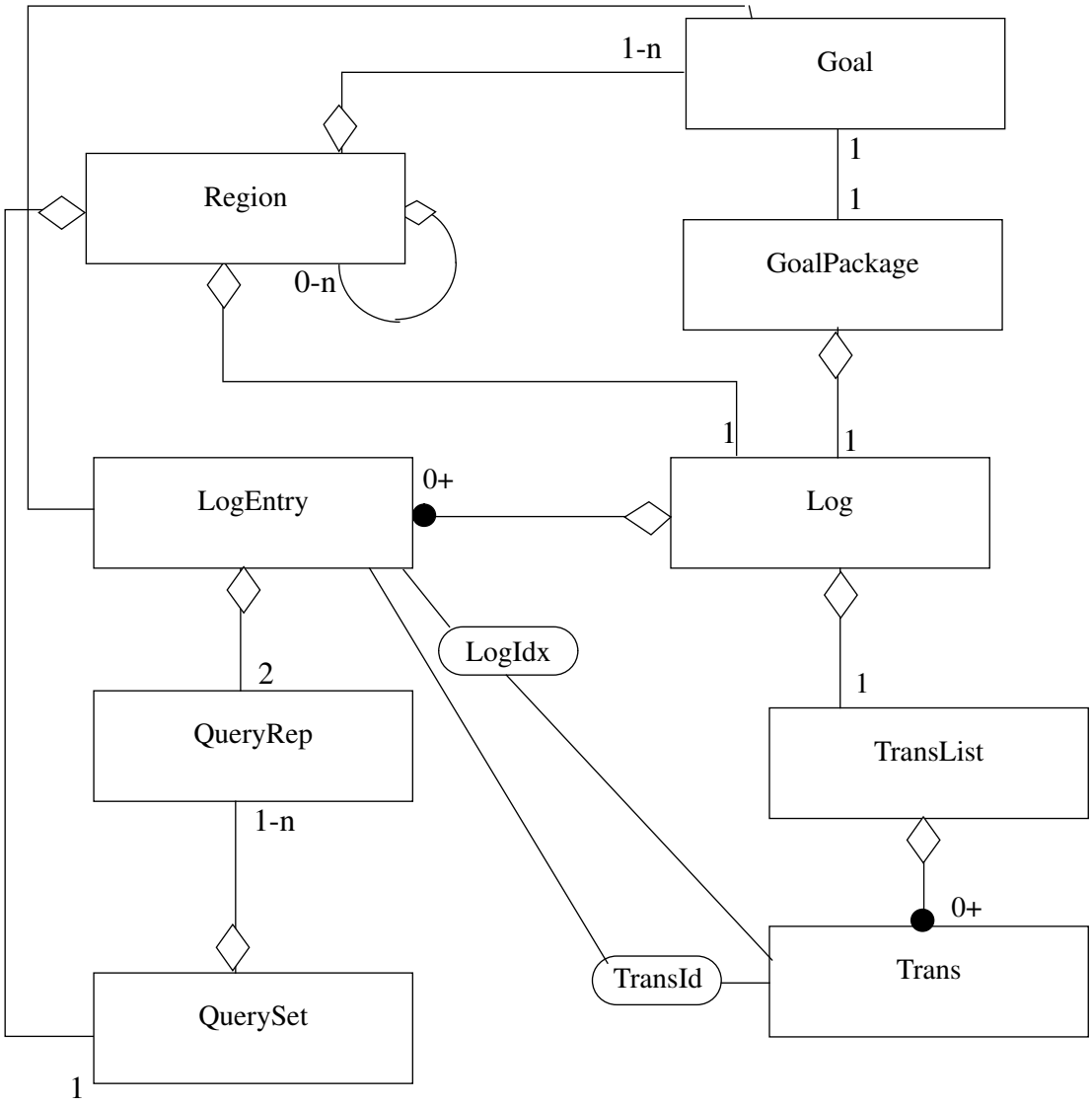
1. **Region** - the optimizer is organized as a DAG of regions. Each Region has one or several goals that it can attain, or try to attain, on a particular query. The regions at the bottom of the DAG - the leaves - are transformations with no controlling or planning activities, while the interior regions use a planning strategy to attain their goals. There is one abstract Region class and all actual regions must be an occurrence of a subclass of Region. The way a Region plans how to attain a particular goal is by the execution of one or more of its goal packages.
2. **GoalPackage** - a Region contains one GoalPackage for each of the goals that it can attain. The GoalPackages can use the Region's subordinate regions to try to attain its goals. The planning activity of a Region is handled within the goal packages.
3. **Log** - both the Regions and the GoalPackages contain a Log where they store intermediate results from transformations. A Log is local to the Region or GoalPackage. This Log is used in the planning activity to see what transformations have been performed, what regions have been called with what result, etc. When the execution of a Region or GoalPackage is terminated, the Log is deleted. The Log contains information about what goals have been tried attained, what subregions that have been called, references to the input and result queries of a transformation, and the result. When a transaction is started or terminated, a message is sent to the Log.
4. **QuerySet** - as a result of a transformation new equivalent queries are generated. These queries are stored in a QuerySet together with the input query. It should be possible to find the best query, and also to prune the QuerySet to keep only a specified number of queries. The default number of queries that are kept is three.
5. **Transaction (Trans)** - the Regions and GoalPackages should be able to keep track of the transactions that have been performed. It should be possible to abort the current or a specific transaction.
6. **Working Memory (WM)** - the Regions and GoalPackages have a working memory where all local information is stored. This is modelled after OPS5 working memory. Examples of such information are queries and flags, but virtually anything that may be needed by the planning system should be stored there.

These are the main classes of the system. In addition to these there are a few that are useful for organizing the rest. The Log is for example a collection of entries, and transformations are collected in a list of transformations. A goal could be viewed as a name, but we suggest that there be made a goal class. This leads to the following additions.

7. **LogEntry** - the Log is composed of LogEntries. These are viewed as a sequence of entries added chronologically.
8. **Transaction List (TransList)** - when a new transaction is started, a transaction entry is added to the list. A transaction must have an identifier for us to be able to abort a specific transaction. It also stores information about what entry in the Log that is the first and last that "belongs" to this transaction.
9. **Goal** - each Region is able to attain one or several goals, and for each goal the Region has a GoalPackage. To attain this goal, a GoalPackage plans the execution of one or several subregions in the hope that the combination of their execution may lead to the main goal being attained. Since the understanding of a particular goal is probably universal across Region boundaries, the goal itself has knowledge about when it is attained. However, any GoalPackage may redefine this behaviour.

3.1.1 Structural Relationships between Classes

In the following we will discuss this architecture and different design alternatives that were considered. For some of the classes this was quite straightforward, while other classes have a much more complex structure. It is these latter that will be discussed the most.



Working memory is used in subregions.

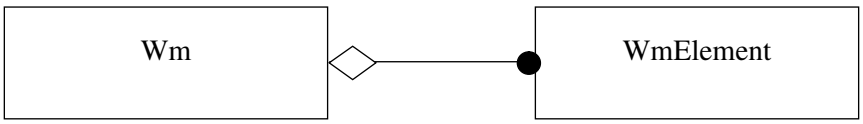


Figure 1 Object Model for EPOQ (notation from Rumb91)

As can be seen from Figure 1 the working memory is not connected to the rest of the model. This is due to the fact that we have not yet used it in any of our regions. When a more complex goalpackage is created it will be necessary to use it in the planning activity. Two attributes have been added in the figure to show the relationships between the transactions and entries in the Log. When a new transaction is started, a LogEntry is added to the Log with this transaction's identifier stored in it. The transaction gets this entry's

index in the Log. This makes it easier to abort a transaction. When an abort is done, all entries in the Log referencing this transaction must also be removed.

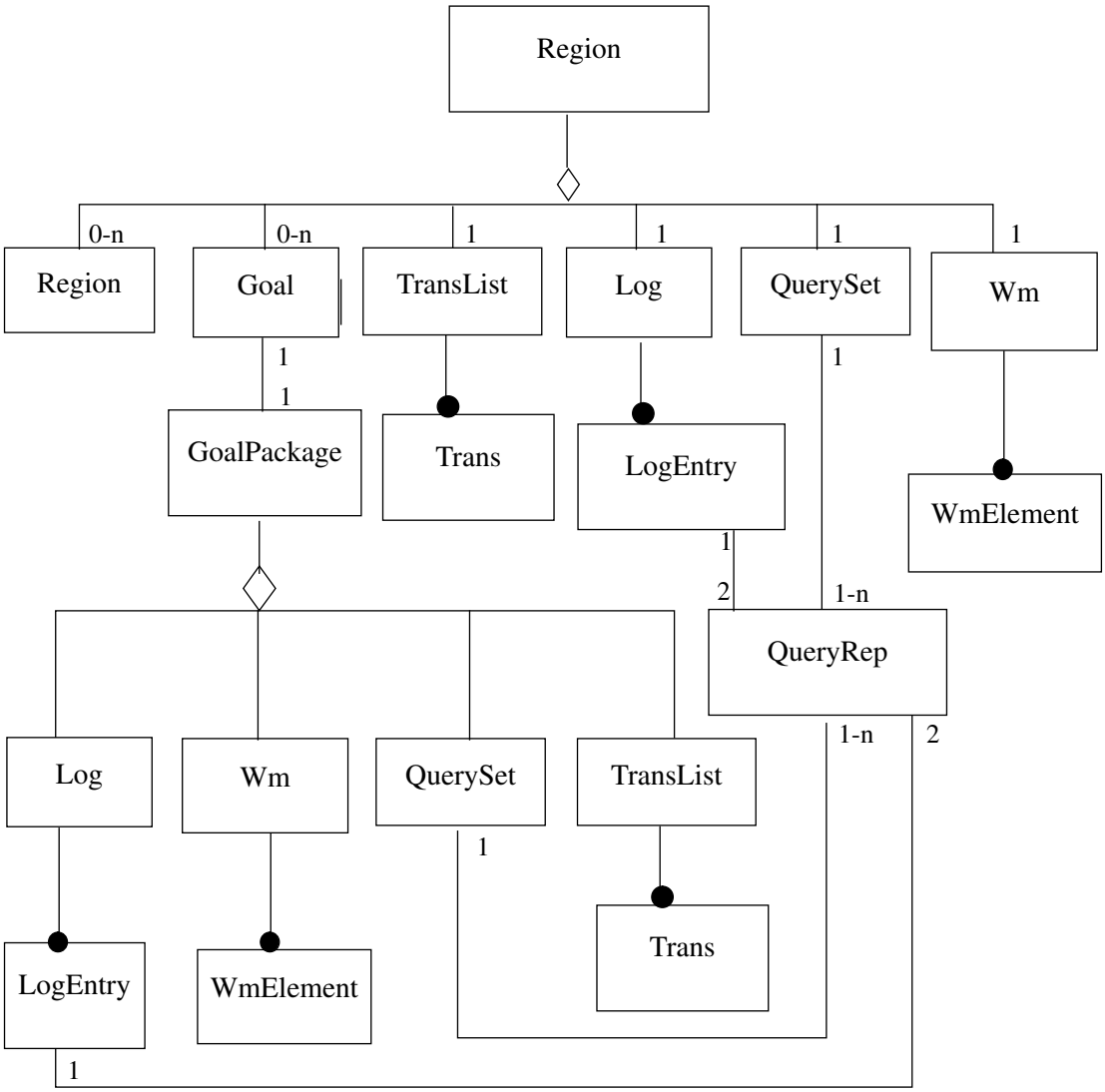


Figure 2 An Object Model with Generalization (notation from Rumb91)

To keep this figure simple inheritance is not included. Regions and GoalPackages will probably be subclassed for every new type of Region. However, some investigation should be done to see how similarities can be revealed to reduce the number of properties that must be redesigned or re-implemented. The working memory class is very simple, and to make it easier to use with flags, which we assume will be the most common types that are stored in the Wm, special functions are added for this purpose. Maybe it would be easier to subclass WM to add extra function or to adapt it for special new types, however, this has not been done.

In Figure 2 you can see that regions and goal packages have a similar structure. This led to the assumption that a generalization of these two could have been designed. However, this was not done, because it was not at all clear that this would have been the best approach. Another alternative would have been to say that leaf Regions are much simpler that interior Regions. They do not need GoalPackages, a Log or a TransList. We considered making class Region very simple containing only the base behaviour and

attributes, but no goal packages, Log and TransList. If this approach had been chosen, Region could not have had a common super-class with GoalPackage. While making an object-oriented design, the natural generalizations will often only occur during the design process. One possible solution is the one shown in figure Figure 3.

3.1.2 Design Alternatives for Regions

The fact that a distinction was made between interior and leaf Regions, where interior Regions control the planning activity and the leaf Regions are merely transformations, made it seem natural to have an abstract class Region with two subclasses LeafRegion and InteriorRegion. LeafRegions are much simper than interior regions that controls the execution of subordinate regions. LeafRegions do not need GoalPackages nor do they perform any planning activity. Any real Region would then be a subclass of one of these two, depending on what category of Region.

However, due to the way we had mapped goals to GoalPackages to make it easy to choose GoalPackage, we wanted to be able to treat all Regions the same regarding this aspect. This was not shown in Figure 3. One might argue that the LeafRegions do not really need a QuerySet nor a working memory. The way the responsibilities have been shared between Regions and GoalPackages, one might also argue that only the GoalPackages need the working memory. If it is used as a tool for the planning system, this might very well be so. The Regions merely use the GoalPackages to do the transformations. The solution to this problem is probably slightly different from what is described in [Mitch93].

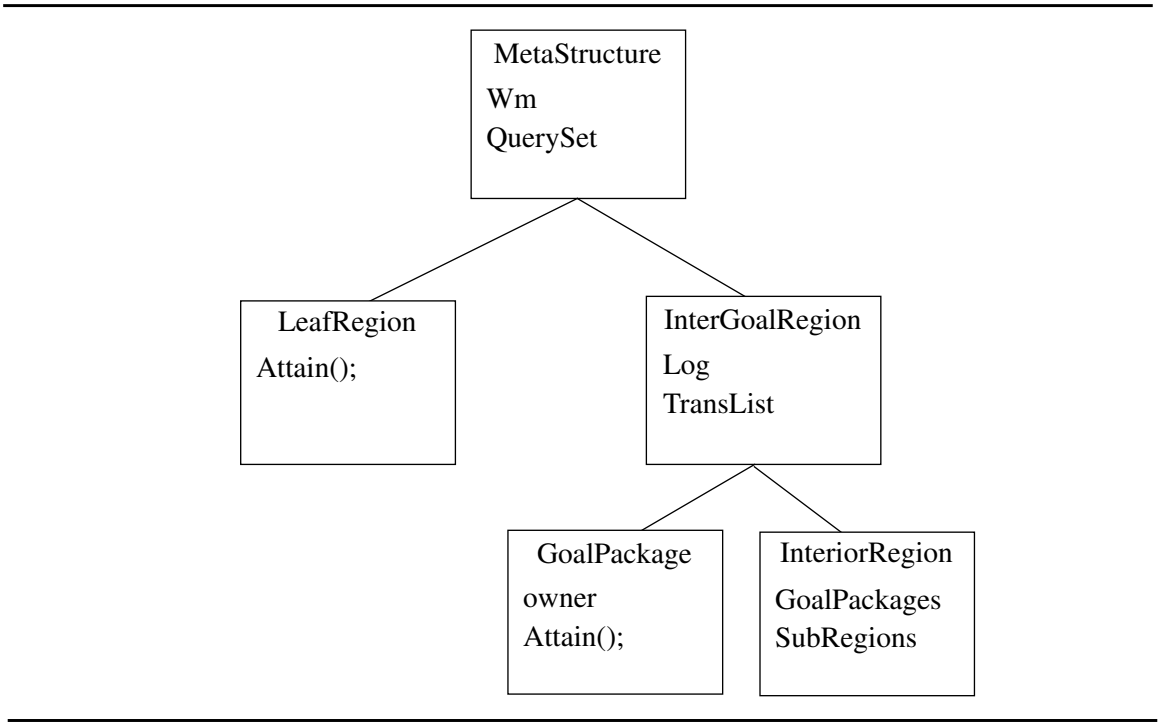


Figure 3 A possible, alternative inheritance structure for Regions and GoalPackages

If the attain function is empty in MetaStructure, it may simply be a call to an optimizer in LeafRegion, and its subregions. In InterGoalRegion, it may be a sequence of calls to some local functions, and they may be redefined in the subclasses whenever necessary. In this way modifications may be constrained in subclasses, and it may be simpler to design new Regions and GoalPackages.

The problem with this solution is that Regions contain a structure which is a mapping between goals

and GoalPackages. For the inheritance structure above to be possible, the binding between a Goal and GoalPackage would have to be redefined. Then a leaf Region which does not have a GoalPackage can still have a goal. This could be an occurrence of class Goal. In the other regions the mapping could be between a pointer to this goal and a pointer to the GoalPackage. The solution that has been chosen has a very effective hashing function, and some redefinition would have to be done. All Regions need a function that returns the name of the goal, and this would have to be redefined for the abstract classes.

The set of subregions is now implemented as a set of Regions. Using the class structure in Figure 3 this would have to be changed to a set of MetaStructure to allow for the LeafRegions to be inserted. However, this would also allow for GoalPackages to be inserted in the set, and this is not legal in accordance to the specification.

In the current implementation there is only one abstract class Region that all other regions are directly inheriting from. I feel quite confident that there should at least be found an intermediate abstract class called InteriorRegion with more knowledge about the planning activities, but this has to be postponed until we achieve more experience about the behaviour of such regions. The interior Region would then redefine the Attain where the behaviour was split on a set of local or private functions. The goal should be that interior Regions that are actually used in an optimizer, should redefine only a few of the local functions.

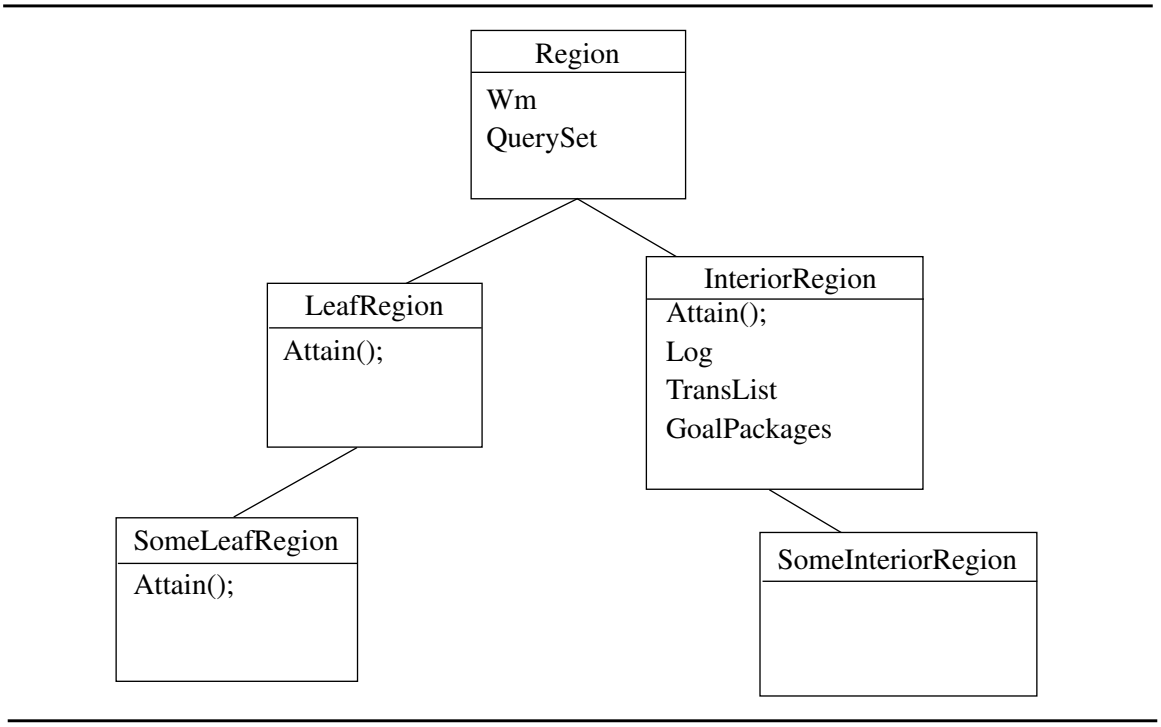


Figure 4 The suggested inheritance structure for Regions

3.1.3 Design Alternatives for Goals

We have considered different possibilities for implementation of goals. These alternatives are the following.

- First we modelled a goal as a name, i.e. a text string. This name is mapped to a goalpackage. The success, prune and bestChoice functions were modelled as functor classes [Copl92]. A functor is a class that has a behaviour like a function and that is used just like you would use a function. Still you can subclass it to change the behaviour.

- Then we realized that these functors were dependent on the goal at hand, and considered implementing Goal as a class where you could make subclasses whenever you wanted the behaviour to change. The class at the root of the hierarchy would have the behaviour that was most likely to occur. This solution was not very attractive, because we would need a separate hierarchy where it might be a problem to figure out how it should be organized - what behaviour is more general than any other when it comes to behaviour for pruning. It might also be the case that the same prune behaviour, for example, would occur in several classes in such a hierarchy.
- The chosen solution was to make one Goal class. This class has a default behaviour defined by a default prune, success, and bestChoice function. If one particular goal needs other functions instead of one of the default onew, new versions of the functions are written, and when the constructor of the goal is called, these functions are given as parameters. It may be difficult to foresee what function is likely to be most generally accepted. When you have default values of any of the parameters, you cannot, in C++, choose to modify only the first one. Then all the parameters must be specified. Therefore it is important that the last parameter is the one that is most likely to change, etc. It may therefore be necessary to modify the sequence of these parameters as experience is gained.

3.1.4 Transactions

The hierarchical Region structure results in a nested calling of Regions. A transaction is a single Region execution. Since Region executions are nested, transactions are also nested. However, within a single region, the transactions are not nested. When a subregion returns to its calling Region, the subregion's transaction is terminated. In the calling Region the transaction is terminated and logged before a new transaction starts. Therefore, there is no need for one Log to be able to register nested transactions. This permits the transactions in one Region or GoalPackage to have a rather simple list structure.

To be able to abort a transaction, the current or a previous one, the transactions must have an identifier. If a child Region aborts, it's Log is cleared, and a failure is returned. Actually a NULL pointer is returned. The calling Region may then choose to Log this transaction as unsuccessful or to remove it. It may be useful to keep it, because it will be possible to tell later that this subregion has actually been tried. If the Region wants to remove the entry in the Log, it can just call the function Abort (Region& unsuccessfulSubregion).

3.2 EREQ Query Representation

The work on a query representation for this project started in 1992 when a parser and tree builder for the ENCORE Query Language was built [Lo92]. The representation was called **qTree** and was implemented in C. MacKeith was making a cost model for the optimizer, and he needed an updated query representation which included cost annotations. To save time the **qTree** representation was used as an intermediate form and transformed into the EREQ query representation, called **QueryRep**. This representation is implemented in C++, but it includes much of the old C code. Thus the parser could be used as it was, a **qTree** was created and transformed into the extended query representation. The previous work provided an API which could also be used. Although most of the details of this representation is hidden from a designer of a EPOQ optimizer, it has been necessary to make calls to this code directly in some test regions we have designed. The reason for this was that the only existing optimizers we could use for testing used the **qTree** representation. In the future all references to the **qTree** representation should be eliminated.

The API functions createQueryTree() and printTree() can be used to display the query tree in graphical form in an X-window. It has proven to be quite useful when testing an optimizer to see whether the queries that are generated are equivalent to the input query. The displayed query tree is of the **qTree** type. This means that to use them we had to convert the **QueryRep** to a **qTree**.

The EREQ Query Representation was implemented as several changes and additions to the previous work. The changes have been done for the following reasons:

1. To avoid duplication of functions in different programs

2. To provide additional functions to access the existing data dictionaries
3. to provide an object oriented version of the query tree (an EAT or Equal Annotated Tree) which can be used in the extensible optimizer
4. to implement the additions in the query representation, particular additional annotations.

The annotations were implemented to allow complete flexibility. The goal was to permit new types of annotations without modifications of the structure. The implementation that was chosen, basically a list of void pointers, makes this possible. Each annotation contains a text string telling what type of annotation it is, e.g. path or cost annotation. This permits flexibility without modification. One of the problems with the void* representation is that the data cannot be copied by a copy constructor, neither can the data be deleted by a destructor.

3.3 C++ Class descriptions

The class and function names are kept as close as possible to the names used in [Mitch93]. A problem has been that different names have been used for the same phenomenon in different parts of the thesis. Another problem has been that something that at first seemed to be an object turned out to be a property of another class and this led to a redesign. Maybe the problem was that we were looking for potential classes in the text while, when writing the thesis, Mitchell did not think of them as objects at all. As a matter of fact, when we thought that the architecture should be a set of inter-operating objects, this was not at all obvious from the outset.

3.4 USL Class Library

In addition to designing the architecture as a framework to increase reuse, it has been important to use existing classes whenever possible. We chose to use the USL class library because it is general and because the major design goal for the library was efficiency. The USL class library provides a set of generally useful classes for commonly used structures. In he lack of template classes in many current C++ compilers, macros are used to give the same effect. The following C++ classes from the USL class library have been used.

The **USLString** class has been used most places where a character string was needed. The only exceptions are in interfaces to other classes, particularly the QueryRep, where ordinary c-strings are used. It may be a solution to substitute all occurrences of c-strings with USLStrings. In the query representation classes, USLStrings are at the present only used as the key for the Map, but it would seem to be a useful class to use generally for strings.

The **Map** class provides an extensible associative array structure which we have used for the linking between goals (strings) and goal packages.

The **List** class is used in several places. e.g., a Region has a list of children, the Log has a list of entries.

The **Array** class is a primitive class that enhances ordinary arrays by permitting the size of the array to be expandible and it can be used as a return value from a function. We have used it for an intermediate structure in the QuerySet.

One problem have been encountered that is probably due to the lack of real template classes in the USL class library. When designing classes it is an advantage to declare functions that do not modify the state of the object as constant functions, e.g. USLString& Goal::getName() const { return goal; } This makes it obvious to the user of a class that this function has no side effect, and the function may also be used on a constant occurrence of this class. Many places where this could have been done, the compiler complained, and this may be due to the fact that there is probably no way to ensure that no modification is taking place during call to a macro, and it is therefore out-ruled by the compiler.

4 Specification of the EPOQ Architecture

This section contains the description of all the classes in the architecture as well as an overview of the location of the source for the architecture.

4.1 Class Definitions

A description of each of the classes of the architecture is given below with their intended use and the semantics of the class methods. We distinguish between the public and protected interface of the classes because this document is also meant to be of help to anybody making subclasses for optimizers.

All the classes have been designed using the Orthodox Canonical Class Form [Copl92]. If classes are designed according to this form, their instances can be assigned, declared, and passed as arguments just like any C++ variable. You need a

- default constructor to be able to declare an instance like you declare an ordinary variable -

```
ClassType instance;
```

  If no default constructor exists, the compiler will create one, and some members may be un-initialized.
- copy constructor - makes a logical copy of its parameter. Externally it should behave as if it is a physical copy
- assignment operator - lets you assign objects just as you would C atomic variables.
- destructor. - cleans up any resources acquired by an object through execution of its constructors.

These operations will not be commented on for the individual classes. In addition to these functions, we also have included an overloaded output operator<< for all classes to make it easier to print the content of objects for testing purposes. The USL classes have such an operator that requires that the output operator is defined for the elements included in the container classes.

4.1.1 Class Region (Header file Region.H)

The optimizer is organized as a directed acyclic graph - DAG - of regions. A top level Region receives a query to optimize together with a goal that it is supposed to attain. The Region may have a set of subregions that can be used to try to attain this goal, and it makes a plan for how it will go about this task. **Region** is an abstract class where the general behaviour of all regions is defined. Any Region must be a subclass of **Region**. All regions of an optimizer will most likely belong to different classes, but the goal is to design a class hierarchy where the modifications and extensions that has to be done are as small as possible.

Leaf regions in the DAG does not have a planning system; they are mere transformations. However, such a Region may itself represent an optimizer.

A Region

- has a name
- knows its original input query
- knows its current query
- has a current goal it is working towards
- has a log containing the transformation history and a current entry
- has a set of subregions
- has a set of goal packages and a current GoalPackage

Public interface

Operators:

`virtual int operator==(const Region&) const;`

Return: TRUE if they are exactly the same Region.

`friend ostream& operator<< (ostream&, const Region&);`

Return: Writes to standard output the information about the current Region and all its subregions. This operator should therefore only be used on the root in a Region tree.

Modifiers:

`virtual QueryRep* Attain (QueryRep &in, const USLString& goal);`

A Region is asked to attain a particular **goal** on a query **in**. It looks for a GoalPackage for **goal** that does the job.

Return: A pointer to the transformed query. This query is an equivalent to **in** if the **goal** was attained. If the **goal** is not attained, NULL is returned, this indicates no success.

`Boolean addGoalPackage(const USLString& goal, GoalPackage* gp);`

The GoalPackage **gp** is added to the Region and bound to **goal**. A Region class may have constructors where goal packages are inserted directly, or this function may be called to add goal packages separately. You cannot bind more than one goalpackage to the same goal.

1. The goal that this GoalPackage is supposed to attain
2. The GoalPackage

Return: TRUE if the goalpackage is added, FALSE otherwise.

`void addSubRegion (Region*);`

A Region that has been created is added as a subregion to the current one. It is added to the list of subregions, and the designer of the Region must decide how this is to be used by the planning system.

Access functions:

`USLString& getName();`

Return: The name of the Region.

`Boolean hasGoal (const USLString& g) const;`

Return: TRUE if the Region has a goalpackage that can attain the specified goal **g**.

`Set(USLString)& getGoals() const;`

Return: The set of names of all goals that can be attained by the Region.

`void listGoals() const;`

Lists the goals that can be attained by this Region on standard output. This function was written for testing purposes, but it has been kept since it may turn out to be useful when testing out new regions.

`Set_of_p(Region)& getChildren();`

Return: A set of pointers to the children (subregions) of the Region.

```
int children() const;
```

Return: The number of children (subregions) this Region has.

```
virtual Boolean
canTransform (const QueryRep& query, const USLString& goal) const;
```

1. input: The query it is asked if it can transform
2. input: The goal for the transformation.

A Region is asked whether it can transform a particular query based on a specific goal.

When planning how to attain a particular goal, a Region or GoalPackage may ask all the Region's subregions whether they think they may be able to attain the specified **goal** on a particular **query**. If a subregion affirms this, it does not necessarily imply that it will actually succeed, since that would require that it actually tried to attain the goal.

The default behavior is that a Region will check whether it has a GoalPackage for this particular goal. If this is not the wanted behavior, this function must be modified.

Return: TRUE if it may be able to transform it, FALSE otherwise.

Protected interface

Some of the following functions are candidates for specialisation when subclasses are designed. Conditions for using them are included here. All functions have been implemented. However, the default behaviour may not be adequate.

Constructors:

```
Region (const char *regionName = "\0");
```

The default constructor has a parameter that represents the name of the Region. This is set to an empty string if not applied. When a subclass i s designed, this constructor should be invoked with the actual regionName: SubRegion (regionName, ...other parameters ....) : Region(regionName) { .... }

```
Region (const Region&);
```

The copy constructor is protected because it not intended to be called. It should probably have been designed private.

```
virtual ~Region();
```

The destructor is protected because the DAG of regions is considered to be a structure that should not be altered during execution. If an optimizer is built that needs to rearrange the structure of the DAG during execution, the destructor would have to be moved to the public interface.

Auxiliary functions:

```
void init();
```

This function is used by constructors. It may be that it should rather be private.

```
void initStore (QueryRep& in, const USLString&);
```

The local store of the Region is initiated before any activity is undertaken. InitStore is called by the Attain function to initialize

1. the original query = current query
2. current goal

Access functions:

```
virtual QueryRep* SelectQuery();
```

Selects which query the GoalPackage is going to use for its transformations.

In this implementation the query that was the result of the last transformation is selected. If no other transformation has been applied, the input query is selected.

In some situations it may be appropriate to select a subquery instead of the whole query. In such situations this function may be redefined.

Result: currentQuery is set to point to the last query

```
virtual QueryRep* ChooseBest();
```

Returns a pointer to the best query. The default version implemented in Region merely returns the result query of the last successful transformation that is stored in the Log. This may not be the wanted behavior in every situation, and this function may therefore be a candidate for modification.

```
virtual Boolean Terminate();
```

Return: TRUE if any subregion has already tried to attain the current goal, else FALSE.

Modifiers:

```
void startTransaction();
```

Marks the start of a transaction

```
void endCurrentTransaction( );
```

Marks the end of a transaction.

4.1.2 Class Goal (Header file Goal.H)

The class Goal represents the goal of a Region or GoalPackage. In addition to the **name** of the goal it contains three function pointers to functions that will vary according to which goal the Region or GoalPackage is supposed to attain.

The functions dependent on the goal are: prune, choose best and success. Default versions of these functions are declared in the file GoalFunctions.H and they are defined in GoalFunctions.C. If the implementor of an optimizer does not supply specialized functions, these will be used. The member functions of Goal will just call the functions that have been supplied when the goal object was created.

Constructors:

```
Goal (USLString n, prune_fp p = &defaultPrune,
        choose_fp c = &defaultChoose, success_fp s = &defaultSuccess );
```

Input:

1. **n** - the name of the goal
2. **p** - a pointer to a Prune function. A default value for this function, defaultPrune, is supplied:
   ```
   List(QueryRep)& defaultPrune(List(QueryRep)&, int number=3);
   ```
   defaultPrune returns: A reference to a list of the QueryRep that has not been removed.
   All queries except the **number** with lowest cost are chosen from the QuerySet, **List(QueryRep)&**. The **defaultPrune** function may be supplied with another **number**, or a special Prune function may be supplied.
3. **c** - a pointer to a **Choose** function. A default value for this function, **defaultChoose**, is supplied where the query with the lowest cost is chosen of the queries q1 and q2.

```
QueryRep& defaultChoose (QueryRep* q1, QueryRep* q2);
```

defaultChoose returns: The QueryRep with the lowest cost of q1 and q2.

4. **s** - a pointer to a Success function. A default value for this function, **defaultSuccess**, is supplied. This function returns true is query q2 has a lower cost than query q1.

```
Boolean defaultSuccess (QueryRep* q2, QueryRep* q1=NULL,
                        int improvement= 0);
```

defaultSuccess returns: True if success, false otherwise.

If the improvement in cost has to be above a specific level for a particular Region, the improvement parameter should be specified.

The function pointers are defined by a typedef.

```
Goal();// Default constructor

Goal (Goal&);                          // Copy constructor

~Goal ( );                             // Destructor
```

Operators:

```
Goal& operator= (const Goal& g);
```

Assignment operator

The goal gets the name of goal **g** as well as the same functions as **g**.

```
int operator== (const Goal& g);
```

Equality operator

Return: TRUE if the name of the goal **g** and all the functions are the same as the current goal.

```
ostream& operator<< (ostream& oo, const Goal& g);
```

The name of goal **g** is written to standard output.

Access functions:

```
USLString& getName();
```

No parameters

Return: a reference to the name of the goal of type USLString&.

```
Boolean success (QueryRep* q2, QueryRep* q1 = NULL,
                 int improvement = 0);
```

Input:

1. The resultQuery q2
2. The original query q2
3. Improvement in terms of %. This is added for the sake of flexibility, but it is not used in the default function.

   This function calls the function that has been bound to sfp (success function pointer) when the goal was created. If now other function was supplied, it is bound to the defaultSuccess function.

```
QueryRep& bestChoice (QueryRep* q1, QueryRep* q2);
```

If a Region or GoalPackage has two queries to choose between, this function decides which query is best in accordance to a specific criteria. In the default version of this function, the best query is the one with lowest cost.

This function calls the function that has been bound to cfp (choice function pointer) when the goal was created. If now other function was supplied, it is bound to the defaultChoice function.

Modifiers:

```
List(QueryRep)& prune (List(QueryRep)&, int number = 3);
```

1. A QuerySet, e.g. a List of QueryRep.
2. The number of queries that are to be kept after the execution of the operation.
   This function calls the function that has been bound to pfp when the goal was created. If now other function was supplied, it is bound to the defaultPrune function.

   Return: If the QuerySet contains more queries than **number**, it returns the **number** best queries, else it returns the whole list.

Requirements for subclasses

1. When the designer of a new Region wants to specify a specific function for any of the goal functions, this may be done by adding it to the GoalFunction files (header and C).
2. When the goal is created by calling the constructor, this new function is specified for the relevant parameter.
3. Whenever the function (success, prune, or bestChoice) is called on this particular goal, the function that is connected to the goal is called.

This means that the designer of the Region or GoalPackage can decide how the goal is defined, and users of the regions or goal packages, does not have to worry about what goal is actually used or how it is attained.

4.1.3 Class QuerySet (Headerfile QuerySet.H))

Every Region and GoalPackage contains at least one QuerySet. When a Region or GoalPackage is first called, the input query is inserted into the QuerySet. Whenever a transformation has been performed and a new, equivalent query has been generated, this query is stored in the QuerySet. From this set, the planning system can choose which query to use in the next transformation, or can choose whether or not a goal has been achieved.

The queries are stored in the QuerySet in chronological order. That is, this is actually not a set, but a list. The reason is that we might sometimes want to know if a query was generated before or after another query. The actual functionality to collect the first added query, etc., has not been implemented yet.

The QuerySet is used in the prune function defined for class Goal. To be able to prune the QuerySet based on a particular Goal, the Goal class is declared as a friend of QuerySet.

Public interface

Constructors:

```
QuerySet (QueryRep& query);
```

Creates a QuerySet containing **query**

The queries in the QuerySet may also be referenced in LogEntries in the Region- or GoalPackage Log.

`QuerySet();`
Creates an empty QuerySet.

`QuerySet (QuerySet& qs);`
Copies QuerySet **qs**

`~QuerySet();`
Destructor - removes the whole QuerySet.

Operators:

`int operator== (const QuerySet& qs);`
Equality operator - the QuerySet is compared to qs.
Return: If equal, TRUE, else FALSE.

`void operator= (const QuerySet& qs);`
Assignment operator. The QuerySet **qs** is assigned to the current.

`ostream& operator<< (ostream& oo, const QuerySet& qs);`
Information about **qs** is written to standard output. This function is dependant of the existence of a similar operator for class QueryRep, and so is the format of the output.

Modifiers:

`Boolean addChoice (QueryRep& qr);`
Adds the new query **qr** to the set of queries. When a deep equality operator for QueryRep has been implemented, it should be used to test whether the query is already in the set of queries.

1. New query
   Return: TRUE if the query is added, FALSE if not.

`Boolean prune (USLString& goal, int number = 3);`
prune removes queries that are no longer considered necessary.

1. The goal is supplied to enable the prune function to be dependent on what goal the Region is trying to attain. The default prune function given for any goal is to keep the queries with lowest cost.
2. The default number of queries that are kept is given by the second parameter.

   Return: TRUE if queries are removed, false if not.

Access functions:

`QueryRep* chooseBest (USLString& goal, QueryRep* q1, QueryRep* q2);`
Dependant on the given goal, the best query of **q1** and **q2** is chosen and returned.

1. Specified goal
2. The queries that are evaluated.
   Return: The best query.

4.1.4 Class GoalPackage (Header file: Region.H)

An interior Region contains one GoalPackage for each main goal that it can attain. The GoalPackage is responsible for selecting subgoals and subregions to execute to try to attain the overall goal. The GoalPackage knows about what Region it belongs to, it's owner, and it is able to get to its subregions.

Several subregions may try to attain the same goal, and a GoalPackage may try to combine the execution of several subregions with different goals to attain its own goal.

The GoalPackage keeps track of the transformations that has been performed in a Log of logentries including what subregions that has been accessed. This is similar to the Log of the regions.

Constructors:

```
GoalPackage (Region* reg, const char* goal);
```

A GoalPackage is created with a particular **goal** and it has a reference to the Region, **reg**, it belongs to.

```
~GoalPackage();
```

Destructor.

Operators:

```
GoalPackage& operator= (const GoalPackage&);
```

Assignment operator.

```
friend ostream& operator<< (ostream&, const GoalPackPtr);
```

Modifiers:

```
virtual QueryRep* Attain (QueryRep& query);
```

User required function to be supplied for all goal packages

Return: The result query. If the goal is not attained, returns NULL.

```
void addSubGoal (const USLString& goal);
```

Inserts the goal in its list of goals.

Access functions:

```
USLString& goal();
```

Return: The main goal of the GoalPackage.

```
virtual Boolean success( );
```

Return: If the goalpackage has had success in achieving its goal, it returns TRUE, else FALSE.

Protected interface

Constructors:

```
GoalPackage();
```

Default constructor

```
GoalPackage (const GoalPackage &);
```

Copy constructor

Access functions:

```
virtual QueryRep* ChooseBest();
```

Auxiliary functions:

```
void initStore (QueryRep& qr);
```

1. The original query.

4.1.5 Class Log (Header file: Stores.H)

The Log is used to keep track of the transformations of a Region or GoalPackage.

The Log contains logentries that are kept in a chronological list to keep track of the sequence in which they were generated. It also contains a list of transactions with references to the Log. The Log knows at what position in the Log the current transaction starts as well as the identifier of the current transaction. The transactions are numbered in the order in which they have been started. After execution of a subregion or GoalPackage, the responsible Region or Log can abort a particular transaction. This is done by sending an abort message to the Log.

Public Interface

Constructors:

```
Log();
```

Default constructor

```
Log (LogEntry& entry);
```

Creates a Log with one LogEntry **entry**.

```
Log (Log&);
```

Copy constructor.

```
~Log(){}
```

Destructor.

Operators:

```
int operator== (const Log&);
```

Equality operator

```
friend ostream& operator<< (ostream&, const Log&);
```

Output operator.

Modifiers:

```
Boolean append (LogEntry& entry);
```

Appends a new LogEntry to the Log

Return: TRUE if **entry** has been added.

`void startTransaction( );`
Mark the start of a transaction

`void endTransaction( );`
Mark the end of a transaction.

`Boolean abort (Region& reg);`
Abort the transaction using Region **reg**
Return: TRUE if such a transaction had been registered.

`Boolean abort( );`
Abort the current transaction if set
Return: TRUE if a transaction has been aborted.

`void removeCurrentTransaction();`
Removes the current transaction if there is one.

`void removeUnsuccessful();`
Remove all unsuccessful transformations from the Log.

`void empty();`
Empties the whole Log.

Access functions:

`LogEntry* getLast();`
Return: The last logentry added to the Log.

`LogEntry* getLastGoalPackage (QueryRep& query);`
Return: The last logentry that contains a reference to query.

`LogEntry* getLastSuccessful( );`
Return: The LogEntry representing the last successful transformation

`Boolean hasBeenTransformed(QueryRep& query);`
Checks whether the current Log has registered that query has been transformed.
Return: TRUE if query has been transformed.

`Boolean hasBeenTried (Region&);`
Checks whether the Region reg has been tried to make a transformation.
Return: TRUE if it has been used whether that transformation was successful or not.

`Boolean hasBeenTried (const USLString& goal);`
Checks whether the coal has been tried attained.
Return: TRUE if it has.

4.1.6 Class LogEntry (Header file: Stores.H)

The LogEntry class describes the elements that can be stored in the Log. Each LogEntry consists of

- a transaction id.
- a pointer to the query that initiates the transaction
- a pointer to the query that results from the execution
- a pointer to the Region that is used to attain the query
- the goal that is tried to be attained
- a success factor

If the goal is not attained the Region returns NULL not the initial query as specified in [Mitch93].

Constructors:

```
LogEntry();
```
Default constructor

```
LogEntry (const LogEntry&);
```
Copy constructor

```
~LogEntry();                          // Destructor
```

Operators:

```
int operator== (const LogEntry&);
```
Equality operator

```
friend ostream& operator<< (ostream& oo, const LogEntry&);
```
Sends information about the LogEntry to standard output **oo** in the format:

LogEntry is :
Transaction id : id#
Original query : <format of a query>
Return query : <format of a query>
Region used : <info about the owner Region>
Goal used : <name of goal>
Success indicator: <TRUE of FALSE>

Modifiers:

```
LogEntry& Init (QueryRep& in, Region& exec, const USLString& goal);
```
Before a transaction starts the information already known is entered into a new LogEntry. This information is:

1. The query that is chosen to be the input query for this transformation
2. The Region that is being chosen
3. The goal to be attained

Return: The partly filled LogEntry.

`void Complete (QueryRep& out, Boolean success);`

After return from the subregion that was called to attain a particular goal, the remaining information is given to the LogEntry. This is:

1. The resulting query. This may be NULL if the goal was not attained or a query that is equivalent to the input query.
2. The success factor. Even if the subregion considers that the goal has been attained and returns a new query, this decision can be overruled by the superior Region, thus entering FALSE here. The normal thing would probably be to accept the subregions decision.

`void setTrans (int trans);`

A transaction is started when a decision has been made to call a particular subregion. The **trans** can be obtained from this transaction by calling the function **Trans::getID**();

`int getTrans();`

Return: The identifier of the transaction represented by the current LogEntry.

`QueryRep* getInQuery();`

Return: A pointer to the query registered as the input query in this LogEntry.

`QueryRep* Result();`

Return: A pointer to the query that was the result of the transformation.

`Boolean Success();`

Return: The success indicator of this LogEntry.

`Boolean triedGoal (const USLString& goal);`

For the planning system it can be useful to find out whether it has previously tried to attain a particular goal. Should this function have been renamed to isGoal(). Alternatively a function that just returns its goal could be implemented and this returned value could be compared to the goal.

Return: TRUE if the goal in the LogEntry is **goal**.

`Boolean usedRegion (Region&);`

Return: TRUE if the specified Region has been used.

4.1.7 Class TransList (Header file: Stores.H)

A list of Trans entries. A Region may abort a transaction. When this is done, all information that relates to this transaction must be removed from the Log and Store of the Region. When a new transaction is started, a new Trans is created and this is set to indicate a start of transaction. The transaction identifier is set, and this same value is added to the LogEntry. That is, each entry in the Log belongs to a particular transaction. When the transaction is done,

Constructors:

`TransList();` **`// Default constructor`**

`TransList (const TransList& tr);`

Copy constructor - each element in **tr** is copied into the transaction list.

```
~TransList(){}                          // Destructor
```

Operators:

```
int operator== (const TransList&);

friend ostream& operator<< (ostream& oo, const TransList& tl);
```

The entries in the transaction list are sent to standard output. This function is dependent on the existence of an output function in class Trans, and so is the format of the output.

Modifiers:

```
void append (Trans&);
```

A new transaction is added to the transaction list.

```
int remove (int* currTransNr);
```

The transaction currTransNr is removed from the transaction list.

Return: If a transaction with the currTransNr exists, the number of the previous transaction is returned, else NotSet is returned.

```
Boolean startsTrans (int idx);
```

Searches the transaction list for a transaction with --- IS THIS FUNCTION WRONG?

Return: TRUE if the entry is the start of a transaction.

```
Boolean endsTrans (int idx);
```

Searches the transaction list for a transaction with id= idx. --- IS THIS FUNCTION WRONG?

Return: TRUE if the entry is the end of a transaction.

```
void getPrevTrans (Trans*);

Boolean backupIncluding (int idx);
```

4.1.8 Class Trans (Header file: Stores.H)

The Trans class is used to keep track of information about transactions, it is not the transaction itself. A Trans is characterized by a transaction ID, a reference to the entry in the Log where this transaction either starts or ends, and whether it actually represents the beginning or end of a transaction. The transaction ID is also kept in the Log entries in the Log.

For each transaction there is two entries in the TransList, one representing the beginning and one representing the end of the transaction. This makes it possible to have nested transactions, although we don't believe that transactions in a Region will be nested. The nesting of transactions are taken care of the organization of the Region hierarchy itself.

Constructors:

```
Trans( int id, int logInx, delimit);
```

Default constructor.

The delimiter can have the values UNDEFINED, START, or END.

When a new occurrence is generated, all values should be supplied.

```
Trans (const Trans&);
```
Copy constructor.

Operators:

```
int operator==(const Trans&);
```
Equality operator

```
friend ostream& operator<< (ostream&, const Trans& tr );
```

Access functions:

```
Boolean start();
```
Check if this Trans entry marks the beginning of a transaction.

```
Boolean end();
```
Check if this Trans entry marks the end of a transaction.

```
int getID();
```
Return: The transaction identifier.

```
int getIndex();
```
Return: The entry in the Log that is related to this transaction.

4.1.9 Class Wm - (Header file: WM.H)

The working memory is modelled after OPS5. To support extensibility, it should be possible to store any type of variable in working memory. The working memory will contain flags and any other type that may be useful. The current implementation excludes queries (QueryRep), since they are already stored in the QuerySet. A couple of specialized functions has been added for entries representing flags, since they are likely to occur often.

No two entries of the same type and with the same name are allowed in WM. Entries can have any number of attributes describing them, but to be able to search the WM for values in any of these attributes, specialized functions may be supplied.

The maximum number of attributes in an entry is set to 100. The wildcard used is * and it can only be used to replace one complete attribute when you search for elements.

Constructors:

```
Wm( USLString& t, USLString& n, USLString& v );
```
The working memory is initialized with one WM element specified as a set of fields of strings.

1. The type of the element
2. The name of the element
3. The value of the element

```
Wm();                                   // Default constructor
```

Operators:

`friend ostream& operator<<( ostream&, const Wm& );`

Sends information about the Wm to standard output on the form of a list of WmElements.

Modifiers:

`Boolean add(WmElement& elem);`

The element **elem** is added to the working memory.

Return: TRUE if element is added.

`Boolean add( char* first ... );`

An element is added to the working memory. Duplicates are not allowed.

1. **first** is the type of the working memory element.
2. The remaining parameters, if any, do not have specific interpretation. However, for flags, we have established the rule that the second parameter is the name of the element and the third parameter is the value. There does not exist any rules for what fields that an element might contain in addition to these. It is recommended that the interpretation of the second and third parameter is adhered to, even though it is not a requirement.
   Return: TRUE if element has been added. If any of the strings in the parameter list is a wildcard, the element is not added, and **add** is FALSE.

`Boolean del (List(USLString)& elem, Boolean wildCard = FALSE);`

Delete element equal to elem if it exists.

1. A list of USLString representing a Working Memory element.
2. Optional. If this parameter is not set or is set to FALSE, the routine will require an exact match for all fields. If the parameter is set to TRUE, fields in **elem** containing a wildcard will be compared.
   If wildCard is defined as TRUE, all entries that matches the described elem, will be deleted, e.g. “FLAG”, “*”, “FALSE” will delete all flags that has a value of FALSE. If wildCard is FALSE, an element is deleted if there is a match.

   This function is called by **del** (char* format ...) if there is a wild card in the parameter list.

   Return: TRUE if one or more elements are deleted, FALSE if not.

`Boolean del( char* format ... );`

This version of the function will accept any sequence of char strings written in quotes and separated with commas. Together they are supposed to represent a working memory element, and the sequence of the fields are significant for the semantics. Any one of the fields may be substituted with a wildcard.

Return: TRUE if an element has been deleted.

`WmElement* match( WmElement& wme );`

Searches through the working memory to find an element that matches the pattern **wme**.

If it is found, a pointer to this element is returned.

`List(WmElement)* match( char* format ... );`

Searches through the working memory to find an element that matches the format specified. Wildcards are permitted as any of the parameters, e.g. match (“FLAG”, “*”, “TRUE”) will return a list of all flags that have the value TRUE.

Return: A pointer to a list of the elements that match the pattern.

Caution: In the documentation to the USL library it is said that this may be dangerous since any modifications done to the list after this pointer has been assigned, and the use of this pointer may have changed the structure of the list, and may cause unpredictable behavior. The intension with this function is to use it in connection with the add and delete, and other match functions. Maybe it could have been made a private or protected member function and thus preventing misbehavior. Used as it is by the other functions in this class, there are no dangers involved.

```
USLString& ReadFlag( USLString& name );
```

Function on elements of type flag. If an element of type flag with name exists, return its value, else return empty USLString.

```
Boolean SetFlag( USLString& name, USLString& val );
```

If element of type flag with name exists, set value to val. return true, else return false

Protected Interface:

```
List(WmElement)* wildMatch (List(USLString)& element, USLString& str );
```

wildMatch is called by a delete or match function when it is discovered that there is a wildCard in the pattern.

wildMatch generates a list of WmElements that matches the pattern in element allowing for the use of wildcards represented by str.

1. **element** represents the WmElement that is the pattern to look for in working memory.
2. **str** is the string representing the wildcard.
   Return: A list of WmElements that matches element where wildcard str is allowed.

4.1.10 Class WmElement - (Header file: WM.H)

Class WmElement is a generalisation of the objects that can be stored in the working memory, and that can be manipulated by the working memory operations ([Mitch93] (thesis) pp. 120-122). Working memory is modelled after OPS5 [OPS5].

A working memory element is composed of a set of fields that together describe the element. There is not a fixed set of fields or attributes describing an element, although the working memory restricts the number of fields to 100. Each field is a string. The only types of elements that was described in [Mitch93] were queries and flags. Queries are not stored in our implementation of working memory. We have not come up with any other types of working memory elements. For simplicity we have decided that the three first fields means type, name and value of element. If any new types of elements are added, new functions written to access them should be supplied.

The main deviation that has been done relative to the thesis, is that the queries are not stored in the Wm since they are already stored in the QuerySet. Also, the Apply operation is not implemented. Instead of Apply() the specialized functions getFlag() and setFlag() have been added.

Constructors:

```
WmElement(const USLString& t, const USLString& n, const USLString& v);
```

The WmElement is initialized with values for the element:

1. The type of the WmElement, e.g. FLAG
2. The name of the WmElement
3. the value of the VmElement

This constructor was designed with the flags in mind, and that is the reason for the three parameters. However, a WmElement may contain any number of fields. The function **add** is used to add additional fields.

```
WmElement( List(USLString)& list);
```

A WmElement is initialized with a **list** of USLStrings. Since a WmElement is not bound to be of a particular length (a predefined number of attributes), it may be useful to generate a list, and then insert it into the Working Memory.

```
WmElement();
```

Default constructor

```
WmElement( const WmElement& element);   // Copy Constructor

virtual ~WmElement();                   // Destructor
```

Operators:

```
int operator==(const WmElement& wme);
```

Equality operator.
Return: 1 if **wme** is equal to the current WmElement.

```
WmElement& operator=(const WmELement&);
```

Assignment operator.

```
friend ostream& operator<<( ostream&, const WmElement&);
```

The WmElement is sent to standard output. Used by the output operator of Wm.

Modifiers:

```
int add (const USLString& str);
```

Adds **str** as the last field in the WmElement

1. The string to be added
   Return: the position of **str** in the WmElement, e.g. 4 if **str** is the 4. field in this WmElement.

```
Boolean update (int pos, const USLString& str);
```

The string at position **pos** is updated with **str**.

1. the position in WmElement of the string to be updated.
2. The new value of the string at position **pos**.
   Return: TRUE if the string at position pos has been updated
   FALSE if there are not **pos** fields in WmElement.

Logical operations:

```
int contains (const USLString& str, int pos = 0);
```

Searches for the string **str** starting at position **pos**.
Return: the position where s**tr** is found, else 0.

`Boolean match (const WmElement&)`

Return: TRUE if complete match, FALSE if not.

`Boolean wildMatch( List(USLString)& list, USLString& wild);`

We want it to be possible to perform a more general search, e.g. find all flags set to true. This function makes it possible to search using wild cards.

1. The list is the search string representing the element we are searching for.
2. The wild card that can represent any value for any particular field in the element.
   Return: TRUE if an element matches either completely or based on the wild card, FALSE otherwise.

`Boolean isFlag();`

A special function to test whether the particular element is a flag.

Return: TRUE if this element is a flag, FALSE if not

`USLString& type();`

Return: The type of the element (The 1. field in the element).

`USLString& name();`

Return: The name of the element (The 2. field in the element.)

`USLString& value();`

Return: The value of the element (The 3. field in the element)

4.2 Location of files.

The files for the EPOQ query optimizer are located in the following directories.

The whole project is stored under directory /pro/oodb/epoq. This is here termed the EPOQHOME directory. All other references below relates to this directory.

Source files for the framework of optimizer architecture are kept in directory **EPOQHOME/architecture**.

This directory should be used for the "include" and "library" directory in a makefile. The files have been compiled with the debugging flag set. This directory includes:

- header files **Store.H, Goal.H**, GoalFunctions.H, **QuerySet.H, WM.H, Region.H**, and implementation of the classes of these files in *.C files respectively**.**
- A library file for these classes - **libEPOQ.a**.

The example regions are kept in the directory **EPOQHOME/regions** and includes classes for a TopLevelRegion and a Join Region and Exodus Region. These are described in section 5. They are all directly subclassed from class Region.

Files for the query representation are stored in directory **EPOQHOME/rep/**. This directory should also be used as the "include" and "library" directory in a makefile. This directory includes:

- header files **OptRepMeta.H. OptRep.H, OptRepCost.H,.OptRepLib.H**.
- the OptRep library **libOptRep.a**;
- the working driver program **makeRep**; and
- the data dictionary files **.data_dict, .type_dict .attr_dict** and **.text_dict**.

- The cost data file **.OptRepTypeCostData**. All these data files contain information about the Altair Travel Agency database.

The compilation for the query representation was done in directory EPOQ**HOME/rep/work/.**

This includes

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

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

Files relating to the "gcl" representation [Lo92] are in directory **EPOQHOME/epoq/rep/RepLib/**. This contains the older version of the library, named **libRep.a**, which is now included in **libOptRep.a**. The header file **OptRepIntRep.h**, which contains enums for the operators etc. is in this directory, and is included in the .H files.

The directory **EPOQHOME/regions** contains two test regions and test drivers in three separate directories.

- The directory **exodus-region** contains header and C-files for a class Exodus that has been subclassed from Region. The library for the optimizer generated from Exodus [Lo92] is also placed here in addition to the c-source files and include files in two separate sub-directories.
- The directory **join-region** contains header and C-files for a class Join-region and necessary header-files and c-source code for the **join-region** described in [Tan 92].
- **Test drivers** for the prototype classes are placed in EPOQHOME/regions/test

Makefiles

There are Makefiles in each of the above mentioned directories. They originally came from one source, but have developed in different directions. Some effort should be put into making them consistent when it comes to what they do and how this is done.

The -DTESTING flag will cause many print statements to be activated.

5 Testing the Architecture

To be able to test the suggested architecture, we designed one main Region controlling two leaf regions.

The controlling Region, called ExTopLevelRegion, has no built-in subregions. We considered it more flexible to add its subregions in the driver program. When called, the optimizer is asked to try to attain the goal "Good_rewrite". A Region contains a map between a particular goal and a GoalPackage. So a match is attempted, and if it has such a GoalPackage, the task is delegated to it. The GoalPackage in turn is set up to try to attain the main goal by a combination of two subordinate goals. In this case the two goals "Best _ Est _ Cost: and "Convert _ Join _ Predicate" are tried in sequence. It searches the set of subregions to see if any of these has s main goal that matches the first of these goal. In that case, the matching subregions are executed. Thereafter the next goal is tried. In this simple case no attempt is made to re-plan while this activity is taking place.

The leaf regions are simple transformations. When called they merely call an existing optimizer, one generated by the Exodus Optimizer Generator [Lo92] and one which tries to identify and insert hidden joins in a query [Tan92]. The two regions were named the Exodus and Join Region, respectively. Both these optimizers where implemented using an old query representation [Lo92], so these regions had to transform a query from the EREQ query representation to this old representation whenever called. When done, the optimizer reports whether the transformation was successful or not, i.e., whether the goal was attained or not.

While testing we discovered a few problems with the query representation and schema manager that are

described below. The testing also showed that it is possible to build an optimizer like this. However, more experience is needed to learn what an extra abstraction level, as indicated in Figure 3, should look like. The way the Attain function can be structured to avoid re-implementation of the Attain function in every class, is particularly important.

The Working Memory and QuerySet were not tested in the above mentioned application. However, they have been tested separately. Several test programs where run to test different sets of classes - separate or in combination.

6 Conclusions and further work

In this section we discuss prpblems that are related to the requirements and motivation of the project, as well as suggesting modifications that need to be done and what direction any new development should take.

6.1 Problems

6.1.1 Requirements

The decision to implement the optimizer in C++ had already been made before the current work started, but there where no other specifications as to how this should be done.

The planning system for EPOQ is described as a rule based system in [Mitch93], and it had been considered using a rule based language to implement at least this part of the system. However, this was abandoned. The best solution might have been to use a multi-paradigm solution, i.e. implement the architecture using an object-oriented approach and the planning system using a rule-based approach.

The description of the working memory was modelled after the OPS5 working memory [OPS5]. The reason was that the OPS5 working memory is very general when it comes to what it can store and how new types can be added on the fly. This matches well with how an object-oriented database must handle new types of objects that may even be created during optimization of a query.

However, this requirement conflicts with the decision to use C++ as an implementation language. It is difficult to implement in C++, because unlike OPS5 C++ is a typed language. We therefore had to make some modifications on the implementation of the working memory. The resulting implementation is not as general, and it is still a bit uncertain as to how useful it may be.

6.1.2 Query Representation

The EREQ query representation [MacKe93] provides all the functionality of the old [Lo92] and some additional facilities. One major problem is the fact that the annotations are not typed. The reason why this solution was chosen, was to keep the representation as flexible as possible, i.e., so that new types of annotations could be added. This is the same discussion as the one used for the Working memory.

This has lead to some unsolved problems. The annotations cannot be copied, deleted, or checked for equality in a general way. To be able to make the annotations extensible, a special list class was defined that contains nodes that in addition to a void pointer contains the size of the data object that the pointer points to. This could have been used to solve the problem with copying and deletion of an Annotation List without knowing anything about the list except the size of the data block in the list, however this has not been implemented. The representation contains functions to manipulate these annotations, which questions the assumption that the annotations should be untyped.

Since the representation needs to know what annotations exist anyway, perhaps the easiest way would have been to choose a completely different approach. This could have been to design a class hierarchy of Annotations. Each node in the query representation could have contained a list of annotations, and this

could have been populated with whatever annotation was needed. The disadvantage of this solution is that a new subclass would have to be designed and implemented each time a new annotation was needed. The advantage is that the program would be much more reliable and clean. The annotations could be manipulated as any other object in the system.

6.1.3 Schema Manager

The schema contains information about the underlying database. The Schema Manager has been written to enable access to type details, and to include type cost parameters. It does these successfully, although the functionality which has been added in comparison to the previous schema manager is only in relation to the cost parameters. However, during the execution of the optimizer, the schema should be extended with intermediate types that are typically tuples created on the basis of a subquery.

The main problem with the Schema Manager is that whenever new types are added to a query during optimization, they are not added to the schema. The next time a transformation is taking place, in the same or another subregion, the system looks in the schema for the types. When they are not found, an exception is reported, and the optimizer crashes.

The reason why this was not discovered when the Schema Manager was first written, was probably that the tests that had been performed were too limited. The problems were only encountered when a query that already contained types generated by the optimizer, was run through the same or another Region.

The Schema Manager should therefore be rewritten. Another reason for doing this is that it is based on the old query representation, and that the quality of that code is not particularly good.

6.1.4 Architecture

The main problem with the architecture is that the suggested framework has not been tested properly. The optimizers we had available did not work properly, and because of the above mentioned problems, we were not able to execute two subregions in sequence. The only way to test a framework is to use it. Subclasses must be generated and used. If the process of subclassing is easy and does not require too much work, one may conclude that the framework is useful. It is therefore important that more optimizer prototypes are made using this framework.

There will obviously be a need for redesign during this process. One of the main areas where a design decision has not been reached, is how the planning activity is to be implemented. Since we have only designed general Region and GoalPackage classes, and they do not have any actual planning activity since they are abstract classes, it is important that when generating concrete classes the planning behaviour is analysed to factor out any common activities. At least the structure of how the planning is performed should be somewhat general. When such behaviour has been identified, this should be factored out and placed in a second level of abstract classes called ControlRegion and ControlPackage. The goal must be that concrete Region and GoalPackage classes should re-implement as little as possible of this behaviour, perhaps only an operation that selects the query to be optimized and the operation that actually does the planning.

6.2 Further work

6.2.1 QueryRep

The query representation is still lacking some functionality, e.g., copy constructors, and operators such as equality and assignment. In [Mitch93] several types of equality operators are described, deep equality, i.e. all the nodes in the query tree are identical in content at any level, or equal_i where the queries are the same to a level i in the tree. Since the queries are referenced through pointers, the only way two queries can be identical is that the pointers are identical. This is of course the easiest to implement, but it is not likely that it will happen often, because copies are made whenever a new transformation is started.

The most serious problem with the QueryRep is that the annotations are untyped. This may lead to serious problems when deleting a QueryRep.

At some point it was decided that equivalent queries should be implemented as an array of queries. This extra dimension was called the Z-dimension. Whenever an equivalent query was created, this should be stored in the Z-dimension. This would also be done to a subquery, that is, on every level in a query there could be an array of equivalent queries. This would result in a multi-list. This dimension was never used.

One problem with this Z-dimension is that the cost information stored in any node represents the cost of the whole subtree - subquery - below that level. If there were several subqueries below this point, it would only be possible to store the cost calculated on the basis of one of these subqueries. Also, if the prune function was to be performed on a QuerySet, it would be difficult to know what parts of the query tree was generated the same time, since possible at the root node there would be only one occurrence in the Z-dimension, while further down there would be two equivalent queries and still further down even more. The heuristic might be that the current query was always the one at front, but of older queries, or subqueries, this might be difficult.

The rationale for storing the queries like this was to keep the size of the query set down. Maybe it would be possible to achieve the same by storing the subqueries in the query set if only a subquery was modified. In such cases it could be stored a flag that this was actually a subquery, and with a pointer to the node in the full query where the subquery had its origin. If this query is to be part in a new transformation, this subquery can replace the previous subquery at that point.

6.2.2 The Architecture

While writing this documentation, a few inconsistencies in the design have been discovered. They are commented on here, but these changes have not been implemented. They should be easy to implement. I will list the changes required class by class.

6.2.2.1 Transactions

As mentioned before transactions are not nested inside a single Region, rather, they are nested by nesting Region executions. The transaction entries contain the transaction identifier, a reference to an entry in the Log and an indication of whether this entry marks the start or end of a transaction. If, as we have stated before, each execution of a subregion called by a GoalPackage represents one transaction, there is no need for the transaction list at all. Instead the suggested functions specified to abort transactions, implemented on the Log, would only have to remove entries in the Log, not the transaction list as well.

As it is now, all information about execution of subregions - input and output queries, region reference, success information, as well as the transaction identifier - is stored in entries in the Log. Thus, it is possible to abort a transaction based on the transaction identifier from the information in the LogEntry alone. As long as the transactions are not nested within one Log, the TransList is not necessary. The nested organization of the optimizer itself might result in components, i.e., regions and GoalPackages, that are simpler.

6.2.2.2 Class WM

The two add functions do not have the same semantics:

- Boolean Wm::add (char* first ...) checks whether there is an identical element already stored in the working memory
- Boolean Wm::add (WmElement& elem) permits duplicates to be stored.

The behavior of the first of these functions are correct, and the second should be modified to also check for an existing identical element before a new one is added. For a flag, the existence of a flag with the same name should not be permitted.

6.2.2.3 New Abstract ControlRegion Class

While designing prototype optimizers, it is important that the planning activities are studied carefully to

reveal any common behaviour and responsibilities in the classes that can be factored out in a new abstract class. This should probably be done by designing two or more interior regions with goal packages and see what they have in common rather that trying to figure out this in advance. The reason that we did not implement such a class, was that we did not want to make any premature decisions about what might be the protocol for such regions. Another reason was that we focused on the overall structure and did not have time to study in detail the parts in [Mitch93] where the planning activities are discussed. A set of scenarios should be used to verify the design and to build prototypes. The information gained must than again be used to improve the current framework.

6.2.2.4 Cooperation Between a Region and its GoalPackages

When a Region is called, it will try to attain the goal it has been given by delegating the task to the GoalPackage for this particular goal. Therefore the planning activity in the Region itself is quite simple. It merely selects a Query and the Goal to be attained and delegates the task to the GoalPackage with this particular Goal. A Region may plan to use several GoalPackages to attain the overall goal. In that case it tries to combine several subordinate goals to attain the overall goal. The planning activities will be a bit more complicated. However, the GoalPackages have the responsibility to plan how to use the specific subregions, and they therefore have a more complex behaviour.

6.2.2.5 The generality of the Attain function

One important topic to investigate is whether it is possible to avoid making each Region and GoalPackage instances of separate classes. If this is possible, the framework would have a much higer reuse value. As mentioned before, one should try to separate the Attain function into subfunctions, to see if this would make it possible to avoid reimplementing the Attain function in each class, and the goal should be to reimplement as few as possible of these sub-functions. However, this is not the only possible solution.

One interesting approach for making the design more general and flexible is to develop a member function, called addRule, in the Region and GoalPackage classes that can be used to add new rules. The rules would be implemented as functions and each Region and GoalPackage would contain a list of pointers to such functions. These functions could then be executed during the planning activities. This could be done in a similar manner as functions are supplied to instances of he Goal class. One requirement for this to be possible, is that the signatures of these functions are identical.

6.3 Concluding remarks

Through this work I have learned a lot both about object-oriented databases and about problems related to query optimization. However, the knowledge gained about designing a framework within an application area where I had little experience, has turned out to be the most valuable for me. If this framework really turns out to be reusable and reused, this effort has been worth while. A framework is never designed in one iteration. All problems will never be encountered, and modifications must be made. Any problems encountered are hartly excepted.

If the problems of reusing this framework turns out to be difficult, or even impossible, an important lesson has still been learned. The experience will be important and will be incorporated in my future work.

References

[Copl92] Advanced C++, Programming Styles and Idioms, James O. Coplien, Addison-Wesley, Reading, Mass., 1992.

[JoFo88] Designing Reusable Classes, Ralph E. Johnson, Brian Foote, Journal of Object-Oriented Programming, Vol. 1, No. 2, pp. 22-31, 1988.

[Lo92] Query Tree Interface, Report on OODB Query Optimizer, by George C. Lo, Brown Univer-

sity, February 1992.

[MacKe93] Query Representation and Cost Model, Andrew MacKeith, June 1993.

[Mitch91] An Architecture for Query Processing in Persistent Object Stores. Gail Mitchell, Stanley B. Zdonik, Umeshwar Dayal, Proceedings of VLDB, 1991.

[Mitch93] Extensible Query Processing in an Object-Oriented Database. Doctoral Thesis by Gail Mitchell, Comp. Sci. Dept., Brown University, April 1993.

[OPS5] Rule-based Programming with OPS5, Thomas A. Cooper & Nancy Wogrin, Morgan Kaufmann, 1988.

[Rumb91] Object-Oriented Modeling and Design, J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Prentice-Hall, 1991.

[Tan92] Identification and Transformation of Queries with Join operations in EQUAL, Peter Tan, Brown University, 1992.

[USL] Introduction to the C++ Standard Components, AT&T Bell Labs (distributed with the library).