Draft Copy: Do not circulate

OPT++User Manual[*†]

Navin Kabra

Computer Sciences Department
University of Wisconsin, Madison

1 Introduction

OPT++[KD95] is a tool for Extensible Database Query Optimization that uses an object-oriented design to simplify the task of implementing, extending, and modifying an optimizer. Building an optimizer using OPT++ makes it easy to extend the query algebra (to add new query algebra operators and physical implementation algorithms to the system), easy to change the search space explored, and also easy to change the search strategy used. Furthermore, OPT++ comes equipped with a number of search strategies that are available for use by an Optimizer-Implementor. OPT++ considerably simplifies both, the task of implementing an optimizer for a new database system, and the task of experimenting with various optimization techniques and strategies to decide what techniques are best suited for that database system.

This document is intended as a user manual for OPT++, as well as for SampleOpt, the sample optimizer that is provided with OPT++. Sample-

[*]This work was supported by the Advanced Research Project Agency, ARPA order number 144-CS84, monitored by the Oregon Graduate Institute of Science and Technology and the Army Research Laboratory.

Opt is a fairly complete optimizer for object-relational query optimization, and in many cases, users of OPT++ should be able to use it directly with a few modifications for their database systems. SampleOpt is designed to be flexible and extensible, so that all of it, or some of its components can be used easily in almost any database system.

In case SampleOpt is completely unsuitable for the purposes of some system, OPT++ can still be used to build an optimizer for that system. In this case, SampleOpt can still be used as an example of how things need to be done, and a new optimizer can be built using "basic" OPT++.

SampleOpt, as it is distributed, contains just a parser and an optimizer for an SQL-like language. It does not contain any actual database execution engine, hence it cannot be used to execute actual queries. Nevertheless, it can still be used to optimize queries and generate optimal execution plans without being connected to a database. We refer to this mode of execution as the "standalone" mode of SampleOpt. In this mode, SampleOpt reads "dummy" catalog information from a file and uses that information to parse and optimize queries. The optimal query plan is then printed out in an ASCII format. This mode is very useful for prototyping and debugging optimizers.

This manual covers three important aspects of using OPT++ and SampleOpt: 1) How to use SampleOpt in "standalone" mode, 2) How to incorporate SampleOpt in some database system, and 3) How to modify the optimizer.

Section 2 describes how to compile and install the code. Section 3 describes how to use SampleOpt in standalone mode. This section includes a description of the query language accepted by the SampleOpt parser. Section 4 gives the details of how to modify SampleOpt for incorporating it into a real database system. This includes a description of the interface to the system catalogs and details of how to interpret and execute the plans produced by SampleOpt. Section 5 gives an overview of the actual code contained in SampleOpt and describes the things required to modify SampleOpt to change the cost-model, the query algebra, the search strategy or the search space. It also describes how to modify the parser to accept new query language constructs.

2 Getting Started

This section deals with installing and compiling the code associated with `OPT++` and `SampleOpt`.

2.1 Installing

To use `OPT++` you will need `g++` version `2.6.2` or better. We have run and tested this code on a SunSPARC Station running SunOS 4.1. `OPT++` does not use any "low-level" system features, hence it should be rather easy to port the code to be used with some other Compiler or Operating System as long as the Compiler has good support for C++ templates, and you have access to the `lex` and `yacc` utilities that work with the OS/architecture you plan to compile it on.

The following are the steps you should follow to compile `SampleOpt` to be used in the "standalone" mode.

- Move the file `opt.tar.gz` to the directory where you would like to install `OPT++`.

- Uncompress `opt.tar.gz`.

  ```
  % gunzip opt.tar.gz
  ```

  This will create the file `opt.tar`.

- Un-tar `opt.tar` by running the command

  ```
  % tar -xvf opt.tar
  ```

  This will create a directory called `optpls` in the current directory.

- Change directory to `optpls`. This directory will be referred to as `OPTDIR` in the rest of this section.

- There should be a file called `Config.dir` in the `OPTDIR` directory. The first line of this file will look something like this:

  ```
  BASE= /local.tom/optpls
  ```

  Change this line so that the variable `BASE` is assigned the full pathname of the `OPTDIR` directory.

- Set up the dependency information that `make` will use by running the following command:

  ```
  % make depend
  ```

- Now run `make`

  ```
  % make
  ```

  This should create an executable called `sampleopt` in the `OPTDIR/bin` directory.

Now `SampleOpt` should be ready to run in the "standalone" mode.

2.2 A trial run

After you have followed the steps in Section 2.1, there should be an executable file called `sampleopt` in the `OPTDIR/bin` directory. This executable runs the sample optimizer in "standalone" mode, *i.e.*, without being associated with any real Database System. This executable reads queries from the standard input, optimizes them and prints the optimal plan on the standard output. Dummy system "catalog" information and "statistics" is read from a file and this information is used to optimize the query.

In addition to optimizing queries read from the standard input, `sampleopt` can also randomly generate queries and optimize them. This can be useful for doing performance studies and benchmarking.

To run `sampleopt` simply run the following command:

```
% sampleopt
```

This should give you the following prompt:

```
Query>
```

At this prompt type the following query:

```
Query> select * from emp where emp.name = "Lee";
```

`SampleOpt` optimizes this query and prints out the following plan on the standard output:

```
The Optimal Plan:
preorder

n1 index idx1 emp name

n2 indexselect n1
== attribute $1 name string "Lee"
empty

end
```

The query language accepted by `sampleopt` is SQL with some extensions (ADTs, methods, references) and some restrictions (no nested-subqueries, no views) and is discussed in detail in Section 3.1. Queries can span multiple lines, and end with a semi-colon. The schema information and the statistics used for parsing the optimizing the queries are read in from the file `.catalogs` in the current directory.

The next section discusses in more detail the query language accepted by the `SampleOpt` parser and how to interpret the plans produced by `SampleOpt`.

3 Using `SampleOpt`

This section describes in detail how to use `SampleOpt` in "standalone" mode. In this mode, `SampleOpt` is not connected to any database system. It just reads "dummy" catalog information from a file, reads queries from the standard input, optimizes the queries using the "dummy" catalog information, and prints the optimal plan on the standard output.

There are a number of reasons why this is useful

- This makes it easy for the Optimizer-Implementor to familiarize himself with `SampleOpt` code before going ahead and incorporating it into a database system.
- This makes it easy to test and debug the optimizer independent of the rest of the database system.
- It offers the Optimizer-Implementor a convenient platform rapidly evaluating new optimizer techniques and strategies. The "standalone" optimizer allows a quick evaluation of new techniques before actually having to implement them in the real database system.

In this section, we mainly concentrate on describing the query language accepted the `SampleOpt`. Details of how to interpret the plans produced by `SampleOpt` and how to incorporate `OPT++` or `SampleOpt` in a real database system is described in Section 4.

3.1 The Query Language

The language accepted by the `SampleOpt` parser is similar to SQL with with some extensions for handling object-relational constructs. In addition to normal select-project-join queries, and SQL aggregates, the query language has syntax to deal with abstract data types (ADTs), functions and methods, reference-valued attributes, and set-valued attributes.

In this section, we give examples of the queries/commands accepted by the parser. One example of each of the different types of commands is given here. Familiarity with SQL is assumed and explanations for commands are included only if the behavior is different from standard SQL.

For reference purposes, the complete syntactic specification of the language (the "yacc" grammar) is included in Appendix A.

Query statements

```
select * from relname1 r1, relname2 r2
where r1.attr1 = r2.attr2
and r1.attr3 < 25
order by r2.attr4
```

A typical SQL select-project-join query. Aggregates in the projection list, and the **group by** clause are supported. The **having** clause is not yet supported. Nested sub-queries are not supported. Relation variable names in the from clause (r1, r2) are optional. In addition to standard relational operators (=, <>, <, >, <=, >=), the **between** and **not between** operators are also accepted. In addition to **and**, the **or** and **not** boolean operators are also accepted. Parentheses can be used to enforce precedence of boolean operators in the where clause.

`SampleOpt` extends SQL a little to incorporate object-relational constructs.

Reference valued attributes: The nested-dot notation can be used for reference valued attributes. Suppose there is a relation "emp" that has an attribute called "dept", and the "dept" attribute is actually a reference to a tuple in the "department" relation, then the following query may be written.

```
select emp.name, emp.dept.name from emp
where emp.dept.budget > 100000
```

ADT methods: As explained before, an attribute of a tuple of a relation may be an instance of some ADT. In this case, methods of that ADT can be invoked on that attribute. Suppose we have a Polygon ADT that that has a method “area” defined on it, and suppose we have a relation called “city” that has an attribute “borders” that is of type Polygon. Then, a query to find the names of all cities with an area less than “50” might be written as follows:

```
select c.name from city c
where c.borders.area () < 50
```

Note, that ADT methods can take an arbitrary number of arguments, and the return value may be another ADT on which other methods can be invoked. Consider this rather complicated (and probably pointless) query:

```
select c1.name from city c1, city c2
where c1.borders.rotate (45).overlaps (c2.borders)
```

Here “c1.borders.rotate (45)” rotates the “borders” polygon by 45 degrees and returns a new polygon. Then we invoke the method “overlaps” on the new polygon that takes another polygon (c2.borders) as its argument and returns TRUE if the two polygons overlap. Thus, this query returns the names of all cities whose borders, when rotated by 45 degrees, overlap with the borders of another city.

Polymorphism of ADT-methods is supported. Thus, two different methods of an ADT can have the same name as long as they have different argument types. Thus for instance, the expression “city.borders.magnify (1.5)” might invoke a polygon method to magnify the polygon by 1.5 times, while the expression “city.borders.magnify (1.2, 1.4)” might invoke a different function that stretches the polygon 1.2 times in the X direction, and 1.4 times in the Y direction.

Arbitrary functions: In addition to ADT-methods, arbitrary function calls (not associated with any ADT) can also be supported (as long as the catalog supports them).

```
select * from emp
where log (emp.salary) > 7
```

This query calls the “log” function on the “salary” attribute of relation “emp” and compares the result to 7. (This is a rather contrived way of finding all employees who are millionaires!)

The following query shows a more realistic example of a function being used. Suppose, we have an ADT called point that represents a point in two-dimensional space. Further, suppose that we have a function called “point” that takes two integers and returns an instance of the Point ADT located at the given co-ordinates (a constructor). Then the following query:

```
select * from city
where city.borders.contains (point (10, 20))
```

finds all citys whose borders contain the point at location (10,20).

As in the case of ADT methods, polymorphism of functions is supported.

Unnesting set-valued attributes: The current implementation of `SampleOpt` implicitly inserts an unnest operation to flatten any set-valued attributes in a query. Thus, for example, if the “person” relation has a set-valued attribute called “children” that returns a set of references to other tuples of the person relation, then the following query retreives the names of all children of person(s) named “Lee”.

```
select person.name, child.name
from person, child in person.children
where person.name = ‘‘Lee’’
```

Utility commands

Creating and Deleting Databases . These are the commands that can be used for creating and deleting databases.

```
create database dbname
```

Creates a database called “dbname”.

```
open database dbname
```

Opens a database called “dbname”.

close database `dbname`

Closes the database called “dbname”.

destroy database `dbname`

Destroys the database called “dbname” and deletes all its contents.

Creating and Loading Relations . These are the commands for creating relation and indices and loading them into the database.

create table `relname (attr1` **integer**`, attr2` **real**`)`

`SampleOpt` currently does not support the **key** clause. In addition to the basic data-types **integer**, **real** and **character**, the data-type of an attribute in the create table command can be the name of any of the ADTs (Abstract Data Type) supported by the database backend. For example, if the database backend is a Geographic Information System (GIS), then it might have ADTs to represent Polygons and Points. In this case, “Polygon” and “Point” can be used as data-types for attributes in the create table command.

load table `relname` **from** `file1, file2, file3, ...`

`SampleOpt` allows a table to be loaded from multiple files. All the tuples in the files file1, file2 and file3 (presumably in some ASCII format that the database backend can understand) are to be loaded into table relname.

drop table `relname`

Delete all the contents of the table given by “relname” and remove it from the database.

create index `idxname` **on** `relname (attrname)`

Create an unclustered index of attribute “attrname” of relation “relname”.

create clustered index `idxname` **on** `relname (attrname)`

Create a clustered index of attribute "attrname" of relation "relname".

drop index `idxname`

Drop the index given by "idxname".

Support for Transactions . Use these commands if a number of statements have to be treated as a single transaction. Otherwise, it should be assumed that each statement is a separate transaction.

begin transaction

end transaction

Update commands

delete from `relname` **where** *predicate*

Here *predicate* stands for a normal SQL-like predicate. Predicates are further discussed in the next sub-section when we describe queries.

insert into `relname values 42, 98.6`

Inserts a single tuple with the given values in relation "relname".

insert into `relname` *query*

Here *query* is any query that returns tuples that have the same schema as the relation "relname".

update `relname` **set** `attr1 = 42, attr2 = 98.6` **where** *predicate*

Updates all tuples of relation "relname" that satisfy the given predicate by setting values of the attributes according to the expressions in the set-clause. The right hand side of the assignment expressions in the set clause can be any valid expression in terms of the attributes of "relname".

3.2 Interpreting the Execution Plans Produced

The format of the execution plans produced by the optimizer is discussed in detail in Section 4.2.

4 Incorporating `SampleOpt` in a Database System

This section describes the work involved in incorporating `SampleOpt` into an existing database system. The "standalone" version of `SampleOpt` takes a query as an input in the form of an ASCII string, parses and optimizes it, and produces an optimal execution plan in an ASCII format. The main things that need to be done to incorporate this into a database are

- Modifying the `SampleOpt` code so that the parser and the optimizer uses information from the actual system catalogs of the database system instead of reading information from a dummy catalog file, and
- Writing code to interpret and execute the execution plans produced by `SampleOpt`.

4.1 The Catalog Interface

The parser and the optimizer of a query processor both require information from the system catalogs for type-checking and for statistics. Since the system catalogs of different database systems have different interfaces, it becomes difficult to integrate an external optimizer into a database system. Specifically, going into the parser and optimizer code and changing all the code that makes catalog calls to conform to the interface of a database can be a time-consuming task.

Hence, we have defined a standard catalog interface and have defined a `Catalog` wrapper class that can be used to insulate the optimizer and parser code from the actual system catalogs. Thus, the optimizer and parser code only makes calls to the methods of this `Catalog` class. And this catalog class, internally makes calls to the system catalogs. Thus, when the optimizer has to be incorporated into some database system, the only changes required for interfacing it with the the system catalogs are all localized to the `Catalog` class.

In addition defining a `Catalog` class, we alos define a number of classes to serve as handles or identifiers for various catalog entities. Thus the `AttributeId` class serves as a handle for information about attributes. Thus when the `GetAttributeId` method of the `Catalog` class is invoked to get information about a particular attribute of a particular relation, it returns an instance of the `AttributeId` class. This `AttributeId` can then be used to get further information about an attribute.

We first describe this catalog interface, including the `Catalog` class and the various "handle" classes. To incorporate `SampleOpt` into a real database system, the methods of the `Catalog` class and the related "handle" classes have to modified to make calls to the actual system catalogs (instead of reading information from a "dummy" catalog file).

Data Types The catalog interface defines a `TypeId` class that serves as a handle or identifier for a particular type (e.g. integer, real, string) associated with attributes and objects in the database. The `GetTypeId` method on the `Catalog` class takes a name of a type and returns an instance of the `TypeId` class that corresponds to the given typename. This class serves as a wrapper around whatever type the real system catalogs use to identify types.

Relations . Similarly the `RelId` class serves as a handle or an identifier for a particular relation or extent. Given the name of a relation, the `GetRelId` method of the `Catalog` class returns an instance of the `RelId` class associated with that relation. An invalid `RelId` is returned if no such relation exists. The `RelId` class has methods that can be invoked on its intances to extract other information about the relation. The `NumAttrs` method returns the number of attributes in the schema of the reation. `NumPages`, `NumTuples`, and `TupleSize` methods return statistical information about the number of pages in the relation, the number of tuples, and the average size of each tuple. The `AttributeId` method can be used to get the information about individual attributes in the relation.

Attributes . The `AttributeId` class serves as a handle or identifier for information about attributes in a relation. The `AttributeId` method of the `Catalog` class returns the `AttributeId` instance associated with an attribute given the relation name and the attribute name. The methods of the `AttributeId` class can then be invoked on that instance to get more information about the attribute. The `TypeId` method is used to indicate determine the type of the attribute. There are methods to indicate whether this is a referenced valued attribute, or a set valued attribute; whether it is a key for the relation, and whether this attribute has an inverse (if it is a referenced valued attribute).

Indices . The `IndexId` class serves as a handle or identifier for information about indexes on a relation. Given a relation name and an attribute name, or an index pathname, the `IndexId` method of the `Catalog` class returns the `IndexId` instance associated with that index. Methods of the `IndexId` class can be used to extract information about the index. Specifically, methods are used to find out whether the index is clustered, the number of distinct values, the minimum and maximum values stored in the index.

Functions and Methods . The `MethodId` class serves as a handle or identifier for information about methods that exist on abstract data types. Given the typename and the method name and a list of argument types, the `MethodId` method of the `Catalog` class returns a `MethodId` instance corresponding to a method on the given type and taking exactly the arguments specified. An invalid `MethodId` is returned if no such method exists. Two methods having the same name but taking different argument types, are considered different due to method overloading.

Storing literal values of Abstract Data Types . `SampleOpt` uses the `Adt` class to represent literals (integers, reals, strings and also literals of abstract data types) that occur in the query to be optimized. This provides a standard and extensible method of manipulating literals of all the types in the system. The most important way in which `SampleOpt` uses this class is by invoking comparison operators (like "<" or ">") on two instances of an `Adt` to compare two values.

Value Distributions . Distributions of attribute values are often stored in system catalogs as "statistics" and are very important in query optimization. Some systems use histograms to store value distributions. Others just store minimum and maximum values and the number of distinct values. In general, a number of different schemes are possible for storing and estimating distributions. `SampleOpt` uses the `ValueDistribution` class to abstract out the concept of a value distribution. A method of this class is used to determine the probability that a given literal value (an instance of the `Adt` class) is less than or equal to a random value picked according to the value distribution. Another method is used to determine the probability that a value randomly picked from this distribution will be less than or equal to a value randomly picked from another given value distribution. `SampleOpt` uses these methods (and the `Adt` class) to estimate selectivities of predicates and result sizes of various operations.

In the current implementation of `SampleOpt` the `ValueDistribution` class just holds the number of distinct values, and the minimum and maximum values of a particular value distribution. This is used with the uniform distribution assumption to compute the required probabilities, as described in [SAC$^+$79].

This standardized catalog interface is used by the parser and the optimizer. To incorporate this optimizer into a database system, the various methods described above have to be changed to invoke the actual functions provided by the real system catalogs. Although this seems like a lot of code to change, the effort required is very low because

- It is very likely that the actual system catalogs directly provide most, if not all, of this functionality. Hence, most of the classes and methods described in this section serve just as wrappers around the actual system catalogs functions
- All the changes are localized to one file, and this makes it much easier.

4.2 Interpreting the Generated Execution Plans

`SampleOpt` generates execution plans in an ASCII plain-text format (as opposed to producing an in-memory tree data structure). Hence, to execute the plans produced by `SampleOpt`, we have to write code that can read the ASCII text, interpret the plan and make calls to the appropriate routines in the "backend" of the database.

An ASCII format has a number of advantages over producing an in-memory data structure. First, this allows the optimizer code which produces these plans and the database "backend" code that executes the plans to be independent of each other as long as the backend understands the format of the plans produced. This also makes it easier for execution plans to be shipped across process boundaries, which might happen if the the optimizer and the backend are in different processes in a client-server architecture. Finally, this also allows the plans to be easily stored in a file on disk for execution at a later time.

In this section, we define the format for the ASCII plans produced by `SampleOpt`. This format has been defined to be unambiguous, and easily machine-readable. *i.e.*, it should be very easy to write code to interpret the plans without having to do any complicated parsing. Consequently, this plan format is not very user friendly (from the point of view of a human reader). Nevertheless, after a little familiarity with the plan format, the Optimizer-Implementor should not have much difficulty interpreting the plans produced by reading the ASCII text produced.

An execution plan is a tree in which each node represents some relational operation or *algorithm*[1] that has to be executed. Figure 1 shows a graphical representation an example of an execution plan. For each algorithm that has to invoked, arguments in the form of predicates to be applied and expressions

[1]In the rest of this document we use the term *operator* to refer to the abstract relational operators and *algorithm* to refer to the actual methods to implement relational operators in the database backend. Thus, *select*, and *join* are relational *operators* while the methods used to implement them, like, *index-select*, *nested-loops-join*, *sort-merge-join* are *algorithms*.

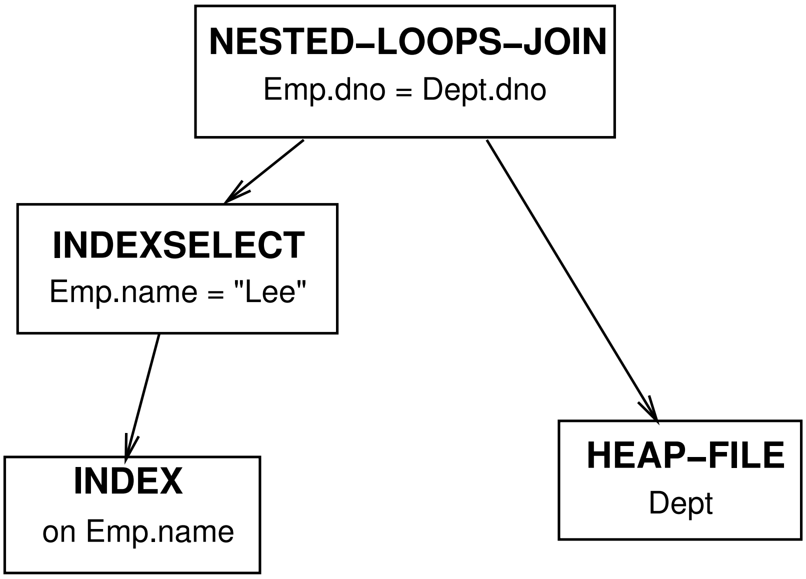

Figure 1: An Execution Plan

to be computed have to be supplied.

In Section 4.2.1 we define the format for encoding predicates and other expressions that are used as arguments to the operators in the Execution Plan Tree. In Section 4.2.2 we define the format used for each of the algorithm instances in the tree. Finally, in Section 4.2.3 we define how these fit together to form complete execution plans.

4.2.1 Interpreting Predicates and other Sub-Expressions

This section defines the format in which the predicates and other expressions are encoded.

The general format for an expression is

expr-type argument1 argument2 ...

Here *expr-type* is a string of characters not containing any spaces and indicates what type of expression this is. For instance, the *expr-type* could be the string "+" indicating that this expression represents an arithmetic addition. *argument1 argument2* ... is a list of expressions representing operands or arguments of the operation specified by *expr-type*. Thus, if *expr-type* is "+" then it will have two arguments indicating the operands that have to be added together. *argument1*, *argument2*, *etc.*, are in turn expressions encoded in exactly the same format. The number The number of arguments depends upon the *expr-type*.

Now we give a list of the various *expr-types* permitted, the number and meaning of their arguments, and we give with examples of expressions of those types.

Referring to attributes . The general format for this is

`attribute` *relation-reference attribute-name*

The `attribute` expression type refers to attributes of relations. The first argument indicates the relation whose attribute this is, while the second argument is the name of the attribute. The first argument can either be a string giving a relation name or it can be a complicated expression representing some relation. (Expressions representing relations are possible if the data-model has reference valued attributes or nested relations. This is discussed later.)

$1 and $2 are special relation names[2]. When this expression is being used as an argument to a relation operator or an algorithm, $1 refers to the relation that serves as the first input of that operator and $2 refers to the second input of that operator (if any). Thus, for instance,

`attribute` $1 salary

refers to the attribute "salary" in the first input of the corresponding operator or algorithm.

Examples where complicated expressions represent the relation are discussed later.

[2] Actually the $ character followed by any integer.

Literals in expression . Basic types are `integer`, `real` and `string`. The format for `integer` is

`integer` *integer*

where *integer* is an actual integer. Thus, for example,

`integer` 10

represents the integer 10[3]. Strings and reals are very similar. Strings must be enclosed in double quotes.

Reference-valued attributes .

`attr-deref` *attribute-reference*

The `attr-deref` expression type indicates that its argument is a reference valued attribute and needs to be dereferenced. The *attribute-reference* is an expression that represents an attribute as discussed earlier in this section.

For example,

`attr-deref attribute` $1 dept

This is an expression in which the "dept" attribute in the first input of the operator is assumed to be a referenced-valued attribute and the reference is dereferenced.

In this example, if we assume that the "dept" attribute is actually a reference to a tuple in the Dept relation, then

`attribute attr-deref` $1 dept budget

In this expression, the "dept" attribute of the first input of the operator is dereferenced and then the "budget" attribute of the resultant Dept tuple is returned as the value of this expression.

Set-valued attributes . The `attr-unnest` expression type represents a set-valued attribute of a tuple being unnested. The general format is,

[3]Although this might seem a rather convoluted way of representing integers, this format makes is much easier for a program to interpret whole expressions.

`attr-unnest` *attribute-reference*

indicates that the attribute represented by the *attribute-reference* expression should be unnested.

Thus,

`attr-unnest` `attribute` $1 colors

indicates that the "colors" attribute of the first input (assumed to be set-valued) should be unnested.

Functions, Aggregate Functions, and Methods . A function call is represented as

`function` *function-name num-args arg1 ...*

This says the the function called `function-name` should be invoked with *num-args* arguments. The expressions representing the arguments of the function follow after the *num-args*.

Thus,

`function` cosine 1 `real` 0.174

represents the function "cosine" being invoked with 1 argument, and that argument is the real number 0.174. In other words, this expression represents the function call "cosine (0.174)".

Aggregate functions are treated just like functions except that the name of the expression type is `aggr-func` instead of `function`. Thus

`aggr-func` max 1 `attribute` $1 salary

represents the expression "max ($1.salary)".

Methods are very similar to functions except that they are invoked on instances of abstract data types (complex types). The general format is

`method` *method-name adt-expression num-args arg1 ...*

The method "method-name" is invoked on the result of the expression represented by *adt-expression* with the given arguments. Thus, for example, `method` clip `attribute` \$1 raster 1 `integer` 4

indicates that the method “clip” should be invoked on the “raster” attribute of the first input with the integer 4 as an argument. In other words, this represent the expression \$1.raster.clip (4).

Arithmetic Operations . General format

arith-op expr1 expr2

Here *arith-op* can be “+” or “-” or “*” or “/”. For example,

`+ integer` 10 `integer` 20

represents the expression “10 + 20”.

Comparison Operations . The format for these is

cmp-op expr1 expr2

Here *cmp-op* can be “<=”, “>=”, “<”, “>”, “!=” or “==”.

< `attribute` \$1 salary `integer` 2000

represents the expression “\$1.salary < 2000”.

Range Comparisons . These represent the *between* and *not between* operators of SQL. The format is

`between` *expr1 expr2 expr3*

which evaluates to TRUE if the value of *expr1* is between the values of *expr2* and *expr3*. Thus,

`between attribute` \$1 salary `integer` 2000 `integer` 3000

represents the expression “\$1.salary between 2000 and 3000”.

The keyword `between` can be replaced by `not-between` to invert the condition.

Boolean Operations .

bool-op expr1 expr2

Here *bool-op* can either be `and`, `or` and represents the logical "and" or "or" of the results of the two expressions.

The unary "not" operator is very similar, but takes only one argument.

`not` *expr1*

represents the logical "not" of the result of the *expr1*.

No-Operation .

`empty`

This represents an empty expression. A NOP. For example if we wanted to invoke a "select" operator without any select predicates, we could give it an `empty` predicate as an argument.

4.2.2 Representing Individual Nodes of the Plan Tree

In this section, we describe how an execution plan tree is represented in the ASCII format by `SampleOpt`. Each node of an execution plan tree represents a database *algorithm* that needs to be invoked in the backend.

The generic format in which each algorithm invocation is represented is:

label algorithm-name listof-input-labels arguments

The *label* is a string that serves as a tag for this particular operation in the execution plan. This label will be used in other parts of the execution plan tree to refer to this particular operation. The *algorithm-name* is the name of the algorithm to be invoked, like `hashjoin` or `mergejoin`. The *listof-input-labels* is a list of one or more strings representing where the inputs of this algorithm are coming from. Specifically, these are labels of other algorithm nodes whose outputs serve as the inputs of this algorithm. The number of labels in *listof-input-labels* is one for unary algorithms and two for binary algorithms. The *arguments* are arguments for the algorithm (like predicates). The number and type of arguments depends upon the individual algorithms.

Now we give the exact format corresponding to the various algorithms.

Dummy algorithms are used to represent relations as they already exist in the database. Specifically, `heapfile` represents a relation stored on disk as a heapfile that can be sequentially be scanned and the `index` represents an index that can be used in an index-scan or in a index-nested-loops join. These serve as leaf-nodes of execution plan trees.

The dummy operations .

> *label* `heapfile` *relation-name*

represents the `heapfile` dummy algorithm.

For example,

> n1 `heapfile` emp

represents a node labeled n1 which represents the relation emp.

Similarly

> *label* `index` *index-name relation-name pathname*

represents the index named *index-name* on the relation *relation-name. pathname* can the name of the indexed attribute in case of a simple single-attribute relational index, or it can be a path expression in case of a path-index.

Thus,

> n2 `index` idx1 emp name

represents the index called idx1 on "emp.name".

Selection using the sequential scan . This represents the relational "select" operator being implemented using a sequential scan. The format is

> *label* `seqselect` *input-label predicate*

Here *label* represents the label by which this operation will be referred in the rest of the plan, and *input-label* is the label of the operation whose output is being fed into this select. The *predicate* is a predicate expression in the format described in Section 4.2.1.

For example,

> n3 `seqselect` n1
> $>$ `attribute` \$1 salary `integer` 2000

If we assume that "n1" is the label for the `heapfile` on "emp" (discussed above), then this represents the "emp" relation being sequentially scanned, and the predicate "emp.salary $>$ 2000" being applied.

AfeatureheadSelection using an Index Scan. This represents the relational "select" operator being implemented using an index scan. The format is

> *label* `indexselect` *input-label idx-predicate aux-predicate*

Here *label* represents the label by which this operation will be referred in the rest of the plan, and *input-label* is the label of the operation whose output is being fed into this select. The *idx-predicate* is the predicate that is to be used to limit the index scan. The *aux-predicate* is an auxilliary predicate that needs to be applied to each tuple after it is brought into memory. This represents a predicate that is not on the indexed attribute, and thus cannot be used to limit the index scan.

For example,

> n4 `indexselect` n2
> $==$ `attribute` \$1 name `string` "Lee" $>$ `attribute` \$1 salary
> `integer` 2000

If we assume that "n2" is the label for the `index` on "emp.name" (discussed previously in this section), then this represents the index on "emp.name" being used scan the "emp" relation using the predicate "emp.name = "Lee"" as the index predicate and the predicate "emp.salary $>$ 2000" as the auxiliary predicate. (Note: a plan like this might be produced while optimizing the query “select * from emp where emp.name = ”Lee” and emp.salary > 2000”.)

The index-only scan is very similar to the index scan, except that the actual tuples of the relation being indexed are never materialized into memory. Only the attribute(s) that are present in the index (as the “key”) are used. This is useful when all attributes of the relation that are not present in the index key are dropped as a result of a projection. The format for the index-only is very similar to the index-scan, except that the formal name of the algorithm is `idxonlyscan` instead of `indexselect`.

Nested Loops Join . The format for the nested-loops join algorithm is,

label `loopsjoin` *input1 input2 predicate*

Here *input1* and *input2* are labels of the algorithm invocations whose outputs serve as the left and right inputs of the join respectively. The *predicate* is a predicate expression that is used as the join predicate.

For instance,

n6 `loopsjoin` n3 n5
== `attribute` $1 dno `attribute` $2 deptno

represents a nested-loops join of two inputs (represented by the lables “n3” and “n5”) in which the “dno” attribute of the left input is equated to the “deptno” attribute of the right input.

Indexed Nested Loops Join . The format for the indexed-nested-loops join algorithm is,

label `nlidxjoin` *input1 input2 join-predicate aux-predicate*

Here *input1* and *input2* are labels of the algorithm invocations whose outputs serve as the left and right inputs of the join respectively. The *join-predicate* is a predicate expression that is used as the join predicate (to scan the index on the inner input). The *aux-predicate* represents an auxilliary predicate that cannot be used for scanning the index on the inner input, but can be applied after the tuples of the inner and outer have been brought into memory. (This is very similar to the auxilliary predicate in case of the indexselect.)

For instance,

> n7 **`nlidxjoin`** n3 n5
> == **`attribute`** \$1 dno **`attribute`** \$2 deptno **`empty`**

represents the same join as in the previous example being implemented using an indexed nested-loops join.

Very similar to the indexed nested-loops join is the index-only nested-loops join. The only difference is that the actual tuples of the inner input are not actually materialized into memory, but only the attributes present in the index (the "key" attributes) are used. The only difference between the format for this join method is that the name of the algorithm is changed to `nlidxonlyjoin`.

Merge Join . The format for the merge join algorithm is,

> *label* `mergejoin` *input1 input2 join-predicate aux-predicate*

Here *input1* and *input2* are labels of the algorithm invocations whose outputs serve as the left and right inputs of the join respectively. It is assumed that the inputs are already sorted on the join attributes. The *join-predicate* is a predicate expression that is used as the join predicate (and it indicates what should the sort-order on the inputs be). The *aux-predicate* represents an auxilliary predicate that cannot be used in the merge join, but can be applied after the tuples of the inner and outer have been brought into memory.

For instance,

> n7 **`mergejoin`** n3sorted n5sorted
> == **`attribute`** \$1 dno **`attribute`** \$2 deptno **`empty`**

represents the same join as in the previous example being implemented using a merge join. It is assumed that "n3sorted" is already sorted on the "dno" attribute and "n5sorted" is already sorted on the "deptno" attribute.

Hash Join . The format for the hash join algorithm is, *label* `hashjoin` *input1 input2 join-predicate aux-predicate*

Here *input1* and *input2* are labels of the algorithm invocations whose outputs serve as the left and right inputs of the join respectively. The *join-predicate* is a predicate expression that is used as the join predicate, and it indicates what attributes are used for partitioning the inputs. The *aux-predicate* represents an auxilliary predicate that cannot be used in the hashing and partitioning, but can be applied after the tuples of the inner and outer have been brought into memory.

For instance,

n7 `hashjoin` n3sorted n5sorted
== `attribute` $1 dno `attribute` $2 deptno `empty`

represents the same join as in the previous example being implemented using a hash join.

Materializing reference valued attributes . The *materialize* operator [BMG93] is used for materializing (or dereferencing) reference valued attributes. The *assembly* algorithm[KGM91] is one method used to implement this operator. The format for this algorithm is

label `assembly` *input-label*
num-assemblies listof-attr-deref-terms

This algorithm is capable of materializing more than one reference at the same time. Hence, the *num-assemblies* term in the above format refers to the number of references that this algorithm is materializing. The *listof-attr-deref-terms* is a list of expressions each representing one dereference. The number of elements in the *listof-attr-deref-terms* is equal to *num-assemblies*.

For example,

n8 `assembly` n1
2
`attr-deref` `attribute` $1 job
`attr-deref` `attribute` $1 manager

We assume that the input of this `assembly` (represented by the label "n7") contains the attributes "job" and "manager", both of which are reference-valued. This algorithm invocation will materialize both these references in one pass of its input. This algorithm invocation might be produced as a part of the optimal plan for the query

```
select emp.manager.name, emp.job.title from emp
```

Pointer Based Join . If a reference-valued attribute contains only references to tuples of another relation or extent, then a pointer-based join can be used instead of assembly to materialize that reference. The format for a pointer-based join algorithm is

label `ptrjoin` *input1 input2 attr-deref-term*

Here *input1* represents the original relation whose attribute is being dereferenced and *input2* represents the relation whose tuples are being referenced. The *attr-deref-term* serves as an implicit join predicate. The implied join attribute is that the OID value contained in the reference-valued attribute of the outer input, should be equated to the OID of the tuples of the inner.

Consider,

n10 `ptrjoin` n1 n1jobref
`attr-deref attribute` $1 job

If we assume that "n1" represents the `heapfile` corresponding to the "emp" relation, and "n1jobref" representes the `heapfile` corresponding to the job extent, then this part of the execution plan represents a pointer-based join being used to materialize the "emp.job" reference-valued attribute.

The unnest operation . Unnesting of set-valued attributes is achieved by the unnest-algo algorithm.

label `unnest-algo` *input-label attr-unnest-term*

The *attr-unnest-term* indicates the attribute that needs to be unnested in the input. For example,

n12 `unnest-algo` n11
`attr-unnest` `attribute` \$1 colors

represents the set-valued attribute "colors" of the input being unnested.

Projections, renaming and function evaluation . In the `SampleOpt` design, the relational *project* operator, the rename operator and evaluation of functions and methods have all been combined into one algorithm called `project-algo`. The format for this is as follows

label `project-algo` *input-label*
num-expressions
new-name1 expression1
new-name2 expression2
...

Here *new-name1* indicates the name of the first attribute of the relation that is output of this algorithm, and *expression1* is an expression (expressed in terms of attributes in the input) that indicates the value of that attribute. Similarly *new-name2* and *expression2* represent the second attribute of the output. The number of such *new-name*, *expression* pairs is given by the *num-expressions* term which is a integer.

Here is how a simple projection is represented using this algorithm,

n20 `proj-algo` n1
2
name `attribute` \$1 name
salary `attribute` \$1 salary

This projection discards everything other than the "name" and "salary" attributes of its input.

Renaming of attributes is handled as follows,

n21 `proj-algo` n1
2

emp_name **attribute** $1 name
emp_salary **attribute** $1 salary

Here the attribute "name" of the input is renamed to "emp_name" and the attribute "salary" of the input is changed to "emp_salary". (Note that in addition to renaming of attribute, plan algorithm invocation will also discard all other attributes. Hence it is doing the same projection as the previous example in addition to renaming.)

Functions, methods and arithmetic expressions are allowed in the expressions in the project-algo algorithm. Hence, this algorithm can be used for projection, renaming and expression evaluation as follows,

proj2 **proj-algo** n1
2
emp_name **attribute** $1 name
emp_salary **attribute** $1 salary
emp_taxes **function** compute_taxes 1 **attribute** $1 salary

This is very similar to the previous example, except that there is one extra attribute in the output called "emp_taxes" whose value is the result of the function "compute_taxes ($1.salary)".

Aggregates and Grouping .

label **aggr-algo** *input-label*
num-group-by-exprs listof-group-by-exprs
num-expressions
new-name1 expression1
new-name2 expression2
...

This format indicates that the input indicated by *input-label* should be grouped using the *listof-group-by-exprs* for grouping. The *listof-group-by-exprs* is a list expressions containing *num-group-by-exprs* terms. The input should be grouped into partitions containing the same value for each expression in the *listof-group-by-exprs*. The rest of the agruments of the `aggr-algo` algorithm are very similar to the arguments of the `proj-algo` algorithm and indicate what aggregate functions to apply, and what names to give the results.

Thus for example,

aggr1 `aggr-algo` n1
1 `attribute` $1 dno
2
dept_no `attribute` $1 dno
max_salary `aggr-func` max 1 `attribute` $1 salary

This indicates that the input should be grouped on the "dno" attribute and the output should contain the "dno" and the "max (salary)". If we assume that n1 represents the `heapfile` for the "emp" relation, then the result this `aggr-algo` invocation generates the answer to the query "select emp.eno, max (emp.salary) from emp".

4.2.3 Format of the Execution Plan

So far we have seen how individual algorithm instances corresponding to single nodes in an Execution Plan are encoded in the `SampleOpt` ASCII plan format. Each of these "nodes" has a label associated with it. This label is used when referring to that node to indicate that it is generating the input of another node.

An Execution Plan is just a sequence of specifications of such "nodes". By reading the node specifications, and then using the labels to decide which nodes serve as inputs for each node, it is easy to construct a whole Execution Plan tree from such a sequence of nodes.

Consider the plan that is graphically shown in Figure 2. This is a plan corresponding to the SQL query

```
select * from emp where emp.name = "Lee" order by emp.dno
```

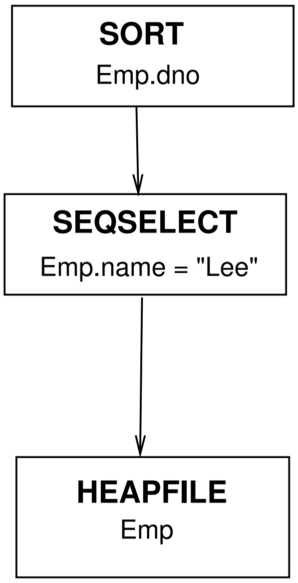

Figure 2: An Example Plan

The ASCII format plan corresponding to it is given below:

```
preorder

n1 heapfile emp

n2 seqselect n1
> attribute $1 name string "Lee"

n3 sort n2
1 attribute $1 dno

end
```

The first word of the plan is a hint that indicates the order of the nodes in the execution plan tree. The word "preorder" indicates that the sequence of the nodes correspond to a pre-order traversal of the plan tree. Another possibility is "depthfirst" indicating a depth-first traversal of the plan tree.

This is followed by a sequence of nodes. Node "n1" represents the `heapfile` corresponding to the "emp" relation. Node "n2" represents a sequential select being applied to the "emp" relation using predicate "name > "Lee"". Finally node "n3" indicates that the result of the select should be sorted on the "dno" attribute. The command "end" indicates the end of the plan.

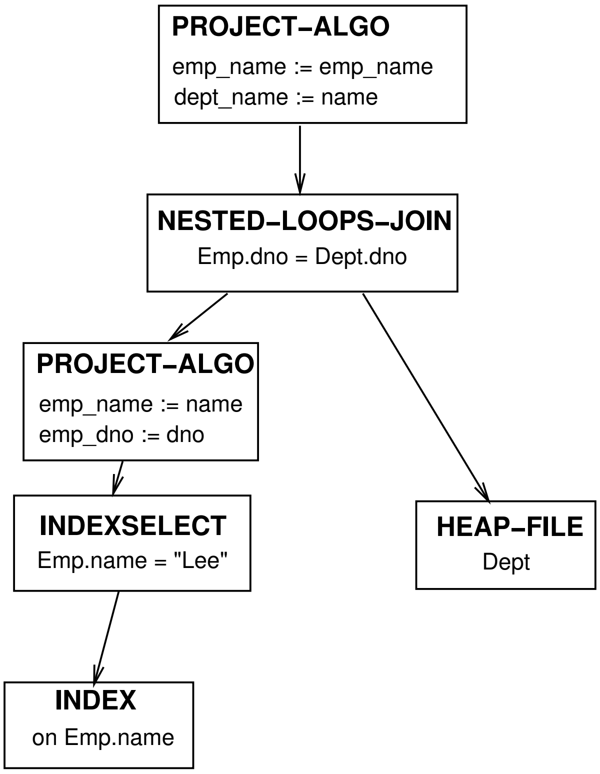

Figure 3: Another Example

As another example consider Figure 3. This is a plan that corresponds to the SQL query

```
select emp.name, dept.name from emp, dept where emp.name
= "Lee" and emp.dno = dept.dno
```

This is what the plan looks like:

```
preorder

n1 index idx1 emp name

n2 indexselect n1
== attribute $1 name string "Lee"
empty

n3 project-algo n2
2
emp_name attribute $1 name
emp_dno  attribute $1 dno

n4 heapfile dept

n5 loopsjoin n3 n4
== attribute $1 dno attribute $2 deptno

n6 project n5
2
emp_name attribute $1 emp_name
dept_name attribute $1 name

end
```

5 Modifying SampleOpt

This section describes how to modify the `SampleOpt` to tailor it to the needs of a specific database system. First, we give an overview of the code organization and its working. Then we indicate what changes are required to the code for modifying different aspects of the optimizer.

This section assumes familiarity with the `OPT++` architecture as described in [KD95]. We start with a *very* brief overview of of the `OPT++` design and then move on to description of the `SampleOpt` code.

5.1 Brief Overview of the OPT++ design

The Optimizaton Process . An optimizer finds the optimal access plan for a particular query based on a given cost-model. The input query is parsed and converted to some internal representation. Then various different access plans that are equivalent to the input query are generated. The execution cost of each of them is estimated and the cheapest plan is output as the optimal plan.

`OPT++` Design Philosophy . `OPT++` defines a few abstract classes and corresponding virtual methods that abstract out all the features that are common to various different query algebras and search spaces. The search strategy can then be implemented entirely in terms of the virtual methods of these abstract classes. Then, classes corresponding to a specific query algebra or search space can be derived from these abstract classes to implement an optimizer for that specific system without having to modify the search strategy at all.

Representing Operator Trees and Access Plans . Instances of the classes derived from the `Operator` class are used to represent operator trees, and their virtual methods are invoked to derive properties of the different operator trees. The classes derived from the `Algorithm` class are used to represent access plans, and their virtual methods are used to derive properties of the access plans, including the estimated cost of execution of the plan.

Thus the various operator and algorithm classes and their methods determine the cost-model of the optimizer.

Generating different Operator Trees and Access Plans . Instances of the classes derived from the `TreeToTreeGenerator` class are used to generate various operator trees equivalent to the input query. Instances of the classes derived from the `TreeToPlanGenerator` class are used to generate various access plans corresponding to the operator trees generated. `PlanTo-PlanGenerator` classes are used to further generate new access plans from existing access plans.

Thus the various tree and plan generator classes and their methods determine which operator trees and access plans get generated. Hence, they control the search space that gets explored.

The Search Strategy . `OPT++` has a number of search strategies that are implemented in terms of the abstract operator, algorithm and the generator classes and their virtual methods. Due to this, at runtime, there is a choice of various search strategies that can be used to optimize the same query algebra.

We first discuss the cost-model of `SampleOpt` as implemented in terms of the various operator and algorithm classes and their methods, and how to modify it. Then we discuss how the various tree and plan generators are used to give a choice of search spaces to be explored during optimization. Finally, we discuss how to switch from one search strategy to another.

We end this section with an overview of the parser code and how to modify the parser to accept new language constructs.

5.2 The Cost Model

The `DeriveTreeDescriptor` virtual method of the `Operator` class and the `DerivePlanDescriptor` virtual method of the `Algorithm` class are used by the search strategy of `OPT++` to estimate the properties of operator trees and access plans. Specifically, the execution cost of an access plan is estimated using these methods.

One important thing that is required for estimating the cost of execution of a particular algorithm with given inputs is knowledge of the different predicates that are being applied by that algorithm and the different expressions or functions that are being computed during execution of that algorithm.

Instances of the `SubExpr` class are used to represent different predicates that need to be applied and expressions that need to be computed for evaluating a given query. Methods of the `SubExpr` class are used to determine properties like selectivity of a predicate and distribution of the values of an expression.

In this section, we first describe how the `SubExpr` class is used to represent predicates and other sub-expressions Then we describe how these classes are used by the `DerivePlanDescriptor` and the `DeriveTreeDescriptor` virtual methods of the `Operator` and `Algorithm` classes to estimate the execution costs.

5.2.1 The `SubExpr` class

The `SubExpr` class and classes derived from it[4] are used to represent select predicates, join predicates, method invocations, expressions in projection list of a query, and other "sub-expressions" that can be used as arguments to relational operators and algorithms. For instance, a select operator has to be supplied a select predicate that it uses to apply to its input. The project relational operator has to be provided with a list of attributes to be projected out. All these, and other, are represented in `SampleOpt` using the `SubExpr` class hierarchy.

The `SubExpr` class hierarchy is very closely related to the ASCII format for representing Predicates and Expressions described in Section 4.2.1. In fact, in most cases, the ASCII format is just a textual representation of the data encoded in the corresponding `SubExpr` class. Hence, it is recommended that you read Section 4.2.1 before reading this section.

In general, a `SubExpr` instance has a data-member indicating the type of the expression and an array of pointers to zero or more other `SubExpr` instances representing the operands of this expression. For instance, an expression representing addition of two expressions would have a type of `SubExpr::plus`[5] and it would have pointers to two `SubExpr` instances indi-

[4]We will refer to this as the `SubExpr` class hierarchy in the rest of this section.

[5]The there is an enumerated type called `SubExpr::Type` that is used to represent the types of the expressions. The type can indicate whether an expression is an attribute of a relation, a method invocation, an arithmetic operation *etc.*.

cating the operands of the addition.

Classes are derived from the `SubExpr` class to represent specific expression types, and the derived class can have methods or data-members corresponding to that specific type. Thus, the `BooleanExpr` derived class is used to represent boolean predicates that can be used in select or join operations, while the `LiteralExpr` derived class is used to represent literals (integers, strings, *etc.*).

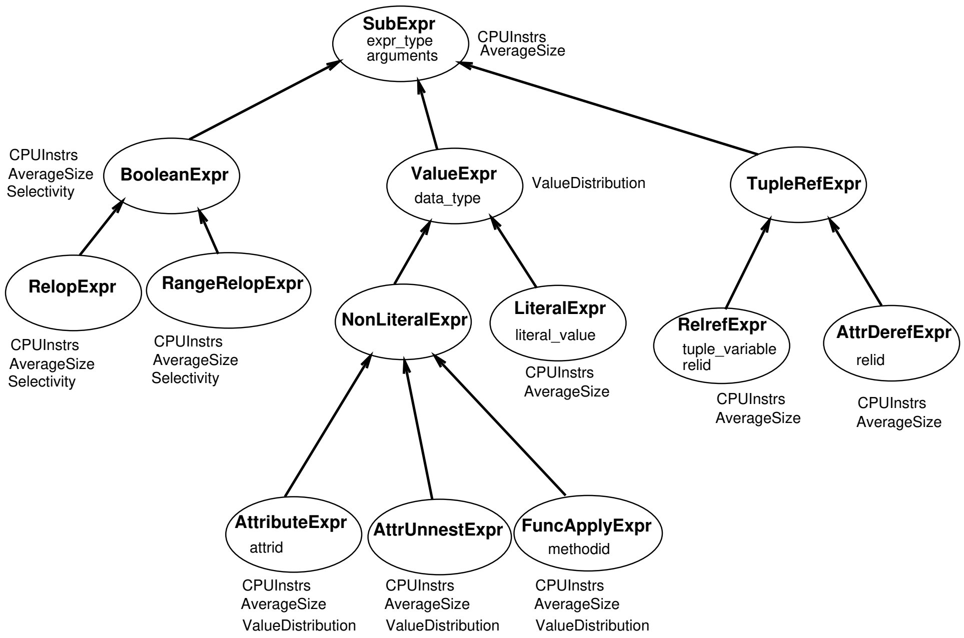

Figure 4: The `SubExpr` class hierarchy

Figure 4 shows the `SubExpr` class hierarchy in the current `SampleOpt` code. For each class we have shown some important data-members and virtual methods. Now we describe each of these classes and the methods in some detail. In the next section we will see how the virtual methods of these classes are used to estimate the costs of execution of various database algorithms.

The `SubExpr` class . As described earlier the `SubExpr` class contains one data member indicating the type of the expression and an array of zero or more pointers representing operands of this expression. The various types of expressions that are possible will become clearer as we describe the classes derived from the `SubExpr` class.

An important virtual method of the `SubExpr` class is the `CPUInstrs` method. This method is expected to return an estimate of the number of CPU instructions executed for evaluating the corresponding expression once. All the classes that are described in the rest of this section over-ride this method to return an estimate of the number of CPU instructions required to evaluate the corresponding type of expression. Since the calculations used for these estimates are pretty straightforward, we will not discuss this further.

Another important method of the `SubExpr` class is the `AverageSize-` method. This method returns an estimate of the average size of the objects that result when this expression is applied to each tuple of a give relation. Statistical information from the catalogs will typically be used to estimate this. All the classes derived from `SubExpr` class over-ride this method to return some estimate of the size of the result.

The `BooleanExpr` class . This class represents boolean predicates. These are expressions that can be used as join predicates or select predicates. A `BooleanExpr` can represent the `and`, `or` or `not` operation being applied to other `BooleanExprs`, or it can be a `RelopExpr` as described next.

The `BooleanExpr` class has a virtual method called `Selectivity` that estimates the selectivity of the corresponding predicate. The selectivity of a predicate cannot be estimated unless we have information about the input(s) to which the predicate is going to be applied. Hence, the `Select-ivity` method is given an operator tree indicating the input(s) to which that predicate is being applied. Specifically, the predicate is expected to be applied during the evaluation of the top operator of the operator tree. The operator tree is passed in as an argument, and a number between 0 and 1 is returned. If necessary, the `TreeDescriptor` of the corresponding `Operator` instance can be used to get information about which predicates have already been applied, which joins have already been computed, *etc.*.

The `RelopExpr` class . This is derived from the `BooleanExpr` class and represents the comparisonoperations "==", "!=", "<", "<=", ">", and ">=". Each of these has two operands that represent expressions that are being compared. Both the operands are expected to be instances of the `ValueExpr` class that is described later in this section. The `Selectivity` virtual method of the `BooleanExpr` class is over-ridden to estimate the selectivity of the comparison predicates using the distribution of values of the operands of the comparison operator. Computing the distribution of values of an expression is achieved using the `ValueDistribution` method of the `ValueExpr` class. This is described further when we discuss the `ValueExpr` class.

The `RangeRelopExpr` class . This class is derived from the `BooleanExpr` class and is used to represent the "between" and "not between" predicates of SQL. A "between" expression takes three operands and is true if the value of the first operand is between the values of the second and third operands. A "not between" expression is the logical complement of the corresponding "between" expression.

The `RangeRelopExpr` class over-rides the `Selectivity` method to compute the proper selectivity of the range predicate using the `ValueDistribution` of the three operands.

The `TuplerefExpr` class . This class is used for an expression that represents a tuple. This can either be a reference to a relation listed in the from clause or it could be a more complicated expression that refers to a tuple.

For example, in the query

```
select e.dept.name from emp e where e.age < 20
```

"e.dept" is an expression that refers to a tuple (and "e.dept.name" refers to one attribute of a tuple). Similarly, the expression "e" refers to a tuple of the emp relation (and "e.age" refers to one attribute of that tuple). These two cases are represented using the `AttrDerefExpr` class and the `RelrefExpr` class respectively.

The `RelrefExpr` class . This class is used to represent an expression that refers to a relation listed in the from clause. There is one data member that contains the name of the corresponding tuple variable in the from clause. Another data member contains the `RelId` of the corresponding relation from the system catalogs. Methods of the `RelId` can be used for accessing information about the relation from the catalogs.

The `AttrDerefExpr` class . This class is used to represent a reference valued attribute being dereferenced. Assuming that the attribute references some other tuple, the attribute dereference gives rise to a `TuplerefExpr`. For instance, in the query "select e.dept.name from emp" the projection list contains an implicit dereference of the "e.dept" attribute to give rise to a "dept" tuple. "e.dept.name" refers to the "name" attribute of that tuple.

This expression has one operand. The operand is a pointer to an `AttributeExpr` instance representing the attribute being dereferenced.

The `ValueExpr` class . This class is derived from the `SubExpr` class and is used to represent expressions that produce some value (as opposed to predicates that are used as filters). Literals, attributes, functions, methods, arithmetic expressions are all considered values. This class has a data member that indicates that data-type of the expression.

The `ValueExpr` class also has a method that is used to estimate the distribution of values when that expression is applied to every tuple of a given relation. Just as in the case of selectivity, the distribution of values cannot be computed unless we know the point in a query operator tree that the expression is being computed. Hence, the operator tree is passed in as a parameter to this method, and an instance of the `ValueDistribution` class is returned. (The `ValueDistribution` class has been described in Section 4.1.)

The `LiteralExpr` class . This is a sub-class of the `ValueExpr` class that is used to represent literal values (*i.e.*, integers, strings, reals, and other ADT literals). This class has one data member that has a pointer to an `Adt` instance that stores the actual value of the literal.

The `NonLiteralExpr` class . This represents `ValueExprs` that are not literals. Currently, there are three main types of non-literal expressions: attributes expressions, attribute unnest expressions, and function application expressions. For each of these, we have derive a class from the `NonLiteralExpr` class. These classes are described next.

The `AttributeExpr` class . This class represents an expression that refers to an attribute. For example, "emp.name", and "emp.dept.name" are both expressions that refer to attributes.

An instance of this class has one data member containing the name of the attribute and another data member containing an `AttributeId` of the corresponding attribute in the system catalogs.

An instance of this class has one operand. The operand is a pointer to the `TuplerefExpr` instance that represents the tuple whose attribute this is.

The `AttrUnnestExpr` class . This class represents a set valued attribute being unnested. This class has one operand. The operand is an `AttributeExpr` instance representing the attribute that is being unnested.

The `FuncApplyExpr` class . This class represents function applications and method invocations. There is one data member that stores the `MethodId` of the function from the system catalogs.

If an instance of this class represents a function then all the operands of that instance represent arguments of the function. If, on the other hand, an instance represents a method of an ADT, then the first operand of the expression refers to the ADT on which the method is being invoked, and the rest of the operands represent the arguments of the method.

5.2.2 Selectivity and Cost Estimation

In the previous section, we discussed how predicates and expressions are represented in `SampleOpt` using the `SubExpr` class hierarchy. We also described the various methods of these classes. In this section, we will see how these are used by `SampleOpt` to estimate the cost of an access plan in particular, and the TreeDescriptor and PlanDescriptor in general.

Is SampleOpt, every Operator and Algorithm instance has data-members indicating what predicates it is applying and what expressions and functions it is computing. These are stored as a set of pointers to SubExpr instances. These pointers are used by the DeriveTreeDescriptor and the DerivePlanDescriptor virtual methods of the Operator and Algorithm classes to do the necessary calculations.

Computing the TreeDescriptor

We recall that OPT++ requires the Optimizer-Implementor to implement the TreeDescriptor class to be a representation of the whole operator tree rooted at a particular operator instance. In SampleOpt the TreeDescriptor class corresponding to a particular operator tree stores the set of all the predicates applied and all the expressions computed in the whole operator tree. An operator tree is considered complete when the corresponding set of predicates and expressions contains all the expressions in the query, and all the predicates in the where clause.

The TreeDescriptor also contains estimates of the size of the result of the operator tree. Specifically, it contains estimates of the number of tuples in the result, the size of the tuples, and the number of pages required to store it.

OPT++ invokes the DeriveTreeDescriptor method on an Operator instance to derive the TreeDescriptor instance corresponding to the operator tree rooted at the given operator instance. The TreeDescriptor instances of the inputs of the operator and the data members of the operator instance can be used by the DeriveTreeDescriptor method to accomplish this.

Deriving the TreeDescriptor for an given Operator instance consists mainly of determining the set of predicates and expressions computed in the whole operator tree, and of estimating the size of the result. The set of predicates and expressions computed in the whole operator tree is computed simply by taking a union of all the predicates and expressions computed by the inputs of the operator with the predicates and expressions computed by the operator itself. The size of the result can usually be computed using simple analytical formulae in terms of the sizes of the inputs, and the selectivities of the predicates being applied by this operator. Thus, for example, the number of tuples in the result of a select operator is simply the product of the selectivity of the select predicates and the number of tuples in the input of the select. As described in the previous section, the selectivity of a predicate can be computed by invoking the `Selectivity` method on it. (Note: The `Selectivity` method in turn uses the `ValueDistribution` of its operands, and the `ValueDistribution` is estimated from information found in the system catalogs.)

Computing the `PlanDescriptor`

`OPT++` invokes the `DerivePlanDescriptor` method on an `Algorithm`-instance to derive the `PlanDescriptor` instance corresponding to the access plan rooted at the given algorithm instance. The `PlanDescriptor` instances of the inputs of the algorithm, the data members of the algorithm instance, and the `TreeDescriptor` and other data members of the corresponding operator instance can be used by the `DerivePlanDescriptor` method to accomplish this.

The most important data member of the `PlanDescriptor` class is the estimated execution cost of the corresponding access plan. (The calculation of "interesting" properties like interesting sort-order is rather straightforward and hence we will not discuss it in this document.)

In general, the following information is used in the estimation of the cost of execution of a database algorithm

- Estimates of the sizes of the inputs and the output. This information is available from the `TreeDescriptor` instances associated with to the operator instances corresponding to this algorithm and its inputs.

- Selectivities of the predicates applied by the algorithm. This information is computed using the `Selectivity` method of the predicate.

- Estimate of the number of CPU instructions required to evaluate the predicates and expressions evaluated by the algorithm. The `CPU-Instrs` methods on the predicates and expressions is used for this.

- Miscellaneous information like size of memory available, page-size *etc.*. This information is available from the catalogs and the runtime system.

The `ComputeCost` methods of the `Algorithm` classes use this information along with analytical cost formulae to estimate the cost of executing the corresponding algorithm.

5.2.3 Modifying the Cost-Model

In the previous sections, we described how the cost-model has been implemented in `SampleOpt`. Now we give a brief description of the changes necessary for modifying the cost-model.

Incorporating Histograms . Currently, `SampleOpt` uses the uniform-distribution assumption, coupled with minimum and maximum values to determine the selectivities of predicates. If the system catalogs of the database store histograms of value distributions as statistics, then `SampleOpt` can be easily modified to incorporate this information. Only the `ValueDistribution` class (described in Section 4.1 and it methods have to be changed to use the histograms and compute probabilities using them. Since the rest of the code just uses the methods of the `ValueDistribution` class, nothing else has to be changed.

Modifying the Selectivity Estimation . All selectivity estimation is encapsulated in the `Selectivity` and `ValueDistribution` methods associated with the `SubExpr` class. Hence, changing the selectivity estimation model can be achieved by simply rewriting these methods. Note: the changes have to be made for the `Selectivity` and `ValueDistribution` methods of all the different classes derived from the `SubExpr` class.

Modifying the Algorithm Execution Costs . The `ComputeCost` virtual method of an algorithm class is used to estimate the cost of execution of that algorithm. A number of things are modelled in the formulae contained in these methods. These include estimating the costs of sequential and random disk I/O, estimating the available memory, the buffer hit rate, the number of passes over the inputs required for the various algorithms, *etc.*. All of this code is collected together in the `OPTDIR/algebra/Acost.c` file. `SampleOpt` uses some fairly standard formulae for making all these estimates and they can be easily modified by changing the code in the `Acost.c` file.

Adding a new kind of predicate . Suppose the query language is modified to allow some new kind of predicate as a select or join predicate. For instance, consider a geographic information system that has spatial ADTs like polygons and points. Such a system might have an "overlaps" predicate that returns true if two polygons overlap each other. Adding a new kind of predicate usually involved adding a new class to the `SubExpr` class hierarchy.

Since the "overlaps" predicate is very similar to the comparison operators (like "<" and ">"), we can derive a new `OverlapExpr` class from the `RelopExpr` class. Then, the `Selectivity` and the `CPUInstrs` methods have to be implemented to estimate the selectivity of the predicate and the number of CPU instructions required to evaluate it. These methods can use spatial information from the catalogs for their estimations. In some cases, the `ValueDistribution` class might have to be modified so that it can store distributions of spatial values.

Note: the changes required to modify the parser to accept new predicates like these are described in Section 5.5.

5.3 The Search Space

The various classes derived from the `TreeToTreeGenerator`, `TreeToPlanGenerator`, and `PlanToPlanGenerator` classes together determine the search space that will be explored by the search strategy during the optimization. In this section we describe how the various generators are implemented in `SampleOpt` and what they do.

Classes derived from the `TreeToTreeGenerator` class fall into two broad categories. The "bottom-up" generators are used for building System-R style bottom-up optimizers. These generators take an operator tree and create new operator trees from it by applying new operators to it. The "transformative" generators are used for building transformative (Volcano-like) and randomized optimizers. These generators take an operator tree and create new operator trees from it by tranforming it into an equivalent operator tree using algebraic transformation rules.

Classes derived from the `TreeToPlanGenerator` class are used to create access plans corresponding to given operator trees. In general, a `TreeToPlanGenerator` class indicates how an algorithm may be used to implement a particular operator.

Classes derived from the `PlanToPlanGenerator` class are used for modifying access plans. One important use is to enforce physical properties (like sort-order) of the output of access plans.

5.3.1 Bottom-Up `TreeToTreeGenerators`

In general, `SampleOpt` derives one bottom-up `TreeToTreeGenerator` corresponding to each operator in the query algebra. The `Apply` method of a `TreeToTreeGenerator` class associated with a particular operator creates new operator instances corresponding to the different ways in which that operator can be applied to the given operator tree. Thus when the `SelectGenerator` is applied to a given operator tree, it creates new instances of the `Select` operator corresponding to the different ways in which a select operator can be applied to the given operator tree.

In general, when a `TreeToTreeGenerator` class creates an instance of an `Operator` class, it has to determine what predicates should be applied by that operator and what other expressions should be computed. Thus, when the `SelectGenerator` class generates an instance of the `Select` operator, it has to determine which predicates are applied by that `Select` operator.

We now describe the general method by which the `Apply` method of a `TreeToTreeGenerator` class determines which predicates and expression should be applied by the corresponding operator. The `Query` class contains a set of instances of the `SubExpr` class representing all the predicates and other expressions that need to be computed for evaluating the query. The `TreeDescriptor` instance associated with each operator tree contains a set of instances of the `SubExpr` class representing all the predicates and expressions already computed by that operator tree. This information can be used to determine which predicates and expressions haven't yet been computed.

Two criteria are used to further restrict the set of predicates and expression that can be computed.

First, only certain types of predicates or expressions can be computed by certain operators. Thus, a join predicate can only be applied by a `Join` operator. Similarly, an `AttrUnnest` operation can only be computed by an `Unnest` operator.

Second, a predicate or expression cannot be computed by an operator unless all the operands of the predicate are already present in the inputs of the operator or can be directly computed from them. Thus, a predicate of the form "emp.age $<$ 20" can only be applied by a `Select` operator if the "emp.age" attribute is present in the input.

Finally, once the set of predicate and expressions that can be computed has been determined, the `Apply` method can make decisions about the different ways in which they can be applied. For instance, if more than one select predicates are applicable to a given operator tree, then the `SelectGenerator` might create just one `Select` instance that applies all the predicates at once, or it can create one different instance for each of the different predicates that can be applied.

This general strategy is used in all the "bottom-up" generators derived from the `TreeToTreeGenerator` class to implement the corresponding `Apply` methods. Now we list all the "bottom-up" generator classes derived from the `TreeToTreeGenerator` class in `SampleOpt` and briefly describe the specific details of each.

The `DBRelationGenerator` class . This class generates instances `DBRelation` of the `DBRelation` operator corresponding to each of the relations in the from clause. These instances represent relations that exist in the database and they form the leaf nodes of the various operator trees built during the optimization.

A method on the `Query` class is invoked to get a set of all the relations in the from clause of the query. Then, one instance of the `DBRelation` operator is created corresponding to each relation.

The `SelectGenerator` class . This class generates instances of the `Select` operator. It gets the set of all the select predicates from the `Query` class, and determines which of them can be applied to the given input. Then it creates one instance of the `Select` operator that applies all the applicable select predicates.

The `LeftDeepJoinGenerator` class . This class generates instances of the `Join` operator class in such a way that all the resulting operator trees are left-deep trees. It gets a list of join predicates from the `Query` class and uses it to determine which joins can be applied to the given input operator tree.

5.4 Changing the Query Algebra

During optimization, a query is represented as a set of predicates that need to be applied to the relations in the query and expressions that need to be computed for evaluating the query. For example, for a simple select-project-join query, the whole query can be represented by the set of predicates in the where clause and the expressions in the target list (the projection list).

Every operator in the query algebra is capable of evaluating/applying one or more of these expression. Thus, for example, a select predicate can be applied by a select operator, a join predicate can be applied by a join operator, and an aggregate expression in the target list can be computed by an aggregation operator.

Thus each node in an operator tree represents a query algebra operator applying one or more of these expressions. An operator tree cannot be complete unless each of the expressions/predicates in the query is evaluated by the operators in the operator tree.

5.5 Modifying the Parser

This section describes the code in the parser. The code is designed in an object-oriented and extensible fashion, hence it is easy to modify the parser to accept new constructs.

As described earlier, the parser takes an ASCII query string as an input and converts it into an operator tree specification. Code generated using Lex and Yacc is used to scan the input query string and convert it into a parse tree. The `Astnode` class is used to represent nodes in the parse tree. The `Astnode` class is actually an abstract class with virtual methods. Depending upon the different types of constructs in the query language, different classes are derived from the `Astnode` class. Each node in the parse tree is actually an instance of one of the classes derived from the `Astnode` class. For example the `FromClauseAstnode` is used to represent the from clause of the query, and the `WhereClauseAstnode` is used to represent the where clause.

Each `Astnode` instance has pointers to other `Astnode` instances which represent sub-parts of the query-expression represented by that `Astnode`. Thus for example, the `FromClauseAstnode` has pointers to `RelNameAstnode` each of which represents a relation name represented in the from clause of the query.

Typically each of the production rules in yacc grammar for the language represents one different class derived from the `Astnode` class. The execution code associated with the production rule, allocates a new instance of the appropriate type. Thus, although the parse tree just consists of pointers to `Astnode`, the actual instances in the tree that the pointers point to are of the various types derived from the `Astnode` class.

The `Astnode` abstract type has a virtual method called `TypeCheck` to do the type-checking of the parse-tree. The classes derived from the `Astnode` class override this method to do type-checking specific to that particular language construct. Now, by invoking the `TypeCheck` method on each `Astnode` node in the parse tree, the whole parse tree can be easily type-checked. Due to the virtual function mechanism, the correct `TypeCheck` method is invoked on each node and the proper type-checking is achieved.

For example, the `TypeCheck` method of the `RelNameAstnode` (instances of this class represent relations in the from clause) ensures that the relation mentioned exists in the system-catalogs, and the `TypeCheck` method of the `FromClauseAstnode` ensures that no two `RelNameAstnode` instances under it can be identical. (Relation names can be repeated in a from clause only if they have different tuple-variable names.)

The interface defined by the `Catalog` class described in Section 4.1 is used by the type-check functions for looking up type information in the catalogs.

Similarly, there is a `GenerateOperatorTree` virtual method of the `Astnode` class that is used to generate an operator tree from a parse tree. Various classes derived from the `Astnode` class override this method. This method is expected to create part of the operator tree corresponding to the query expression represented by that node of the parse tree. Typically, this method invokes the `GenerateOperatorTree` method on each of its sub-trees to generate parts of the operator tree. It then uses the information from those various sub-trees to create the operator tree corresponding to that whole query expression.

5.5.1 Modifying the parser

Modifying the parser is very easy due to the extensible design of the parse tree. Adding a new construct to the query language requires the following steps

- Derive a new class from the `Astnode` class corresponding to the new construct.

- Implement the `TypeCheck` method for this class so that expressions of this new type can be type-checked properly using the `Catalog` interface. This `TypeCheck` method will be automatically invoked by the type-checker when the parse-tree is constructed.

- Implement the `GenerateOperatorTree` method for this class so that the correct operators are inserted at the right place in the operator tree corresponding to the query when it is generated by making calls to the `GenerateOperatorTree` method of the various `Astnode` instances.

- Add the production rule corresponding to the new construct to the Yacc source file. In the executon code associated with that production rule, allocate a new instance of the new type just derived from the `Astnode` class. This ensures that an instance of the new class is inserted into the parse tree at the right place. Hence, the correct `TypeCheck` and `GenerateOperatorTree` methods are invoked when the parse tree is being type-checked and when the operator tree is being generated.

Appendix A: Yacc Grammar for the Query Language

This appendix shows the grammar that is currently accepted by the parser in BNF format.

```
StartState      :       Command
                ;

Command         :       Utility
                |       Update
                |       Query
                |       /* empty */
                ;

Utility         :       t_create OptClustered t_index Ident t_on SimpleRelName
                        t_lparen IdxAttrList t_rparen
                |       t_drop t_index Ident
```

```
                |       t_create t_table Ident t_lparen ColTypenameList t_rparen
                |       t_load t_table SimpleRelName t_from FilenameList
                |       t_partition t_table SimpleRelName t_on Ident
                |       t_drop t_table SimpleRelName

                |       t_create t_db Ident
                |       t_drop t_db Ident
                |       t_open t_db Ident
                |       t_close t_db Ident
                |       t_begin t_trans
                |       t_end t_trans
                ;

OptClustered    :       t_clustered
                | /* empty */
                ;

Update          :       t_delete t_from SimpleRelName WhereClause
                |       t_insert t_into SimpleRelName Query
                |       t_insert t_into SimpleRelName
                        t_lparen ColList t_rparen Query
                |       t_insert t_into SimpleRelName t_values ValueList
                |       t_insert t_into SimpleRelName ColList
                        t_values ValueList
                |       t_update SimpleRelName t_set AttrUpdateList WhereClause
                ;

AttrUpdateList  :       AttrUpdate
                |       AttrUpdateList t_comma AttrUpdate
                ;

AttrUpdate      :       Col t_equal Expr
```

```
                |       Col t_equal t_null
                ;

Query           :       SelectClause FromClause WhereClause
                        GroupBy OrderBy
                |       t_lparen Query t_rparen SetOp t_lparen Query
                        t_rparen SetOrderBy
                ;


SetOp           :       t_intersect
                |       t_minus
                |       t_union
                ;

SelectClause    :       t_select t_star
                |       t_select AllDist ExprList
                ;

AllDist         : /* empty */
                |       t_all
                |       t_distinct
                ;

ExprList        :       Expr
                |       ExprList t_comma Expr
                ;

FromClause      :       t_from RelNameList
                ;

RelNameList     :       RelName
```

```
                |       RelNameList t_comma RelName
                ;

RelName         :       SimpleRelName
                |       RelNameRelVar
                ;

SimpleRelName   :       Ident
                ;

RelNameRelVar   :       Ident Ident
                ;

WhereClause     :       /* empty */
                |       t_where Predicate
                ;

Predicate       :       Expr
                |       Expr Relop Expr
                |       Expr t_between ConstExpr t_and ConstExpr
                |       Expr t_not t_between ConstExpr t_and ConstExpr
                |       t_not Predicate
                |       Predicate t_and Predicate
                |       Predicate t_or Predicate
                |       t_lparen Predicate t_rparen
                ;

Relop           :       t_equal
                |       t_not_equal
                |       t_less
                |       t_less_equal
                |       t_greater
```

```
                |       t_greater_equal
                |       t_overlaps
                |       t_not t_overlaps
                ;

Expr            :       t_plus Expr  %prec t_unary_minus
                |       t_minus Expr %prec t_unary_minus
                |       Expr t_plus Expr
                |       Expr t_minus Expr
                |       Expr t_star Expr
                |       Expr t_div Expr
                |       Literal
                |       Ident
                |       Expr t_dot Ident
                |       Expr t_dot Ident t_lparen OptArgList t_rparen
                |       Ident t_lparen OptArgList t_rparen
                ;

OptArgList      :       t_star
                |       t_distinct Expr
                |       ArgList
                |       /* empty */
                ;

ArgList         :       Expr
                |       ArgList t_comma Expr
                ;


Literal         :       Integer
                |       Real
                |       String
```

```
                ;

ConstExpr       :       Expr
                ;

GroupBy         :       /* empty */
                |       t_group t_by ExprList
                ;

OrderBy         :       /* empty */
                |       t_order t_by SortOrderList
                ;

SortOrderList   :       SortOrder
                |       SortOrderList t_comma SortOrder
                ;

SortOrder       :       Expr OptAscDsc
                ;

OptAscDsc       :       /* empty */
                |       t_asc
                |       t_dsc
                ;

SetOrderBy      :       /* empty */
                |       t_order t_by IntegerList
                ;

IntegerList     :       Integer
                |       IntegerList Integer
                ;
```

```
ColList         :       Col
                |       ColList t_comma Col
                ;

Col             :       Ident
                |       Integer
                ;

FilenameList    :       Filename
                |       FilenameList t_comma Filename
                ;

Filename        :       Ident
                |       String
                ;

Ident           :       t_identifier
                ;

Integer         :       t_integer
                ;

Real            :       t_real
                ;

String          :       t_string
                ;

IdxAttrList     :       IdxAttref
                |       IdxAttrList t_comma IdxAttref
                ;
```

```
IdxAttref       :       Col OptAscDsc
                ;

ColTypenameList :       ColTypename
                |       ColTypenameList t_comma ColTypename
                ;

ColTypename     :       Ident Typename
                ;

Typename        :       Ident
                |       Ident t_lbracket Integer t_rbracket
                ;

ValueList       :       ConstExpr
                |       ValueList t_comma ConstExpr
                ;
```

6 Acknowledgements

References

[BMG93] José A. Blakeley, William J. McKenna, and Goetz Graefe. "Experiences Building the Open OODB Query Optimizer". In *Proceedings of the 1993 ACM-SIGMOD Conference*, Washington, DC, May 1993.

[KD95] Navin Kabra and David J. DeWitt. "Opt++: An Object Oriented Implementation for Extensible Database Query Optimization". Unpublished manuscript. Available at http://www.cs.wisc.edu/ navin/research/opt++.ps, 1995.

[KGM91] Tom Keller, Goetz Graefe, and David Maier. "Efficient Assembly of Complex Objects". In *Proceedings of the 1991 ACM-SIGMOD Conference*, Denver, Colorado, May 1991.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. "Access Path Selection in a Relational Database Management System". In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1979.