Exercise 7: Query optimization in AMOS2 (PC lab)

When an AMOSQL query is entered to AMOS2 there are several alternatives to find the cheapest query plan for executing the query:

- Heuristics. The search space is limited using heuristic rules [3].
- Random optimization. The optimizer chooses randomly different points in the search space and uses an extended version of 'Hill climbing' to explore the surrounding points [1][4].
- Exhaustive search [6]. The complete search space is searched. Guarantees the cheapest query plan (if the cost model is correct).

In this exercise, you will use the internal LISP in AMOS2, callsed ALisp [5], to write an exhaustive query optimizer, using the “dynamic programming” algorithm.

7.1 Goals

This exercise has the goal to give an understanding of the query optimizer in AMOS2. The purpose of this exercise is to give insight to how an exhaustive search can be implemented effectively, and how to different parameters, cost and fanout can be used to rank different query plans.

7.2 Time

Six hours of scheduled lab time (excluding home preparations) have been allocated for this exercise. (Note that this lab is done on PC and not on the Sun system!)

7.3 The actual exercise

Read the rest of this description. Download AMOS2 as in Exercise 6. In the file `~dbtek/kursbibliotek/amos/kodskelett.lsp` there is a code skeleton for the LISP function `dynprogsort`. It is the function that should perform exhaustive search (it is optional to use the skeleton). Below follows a description of query optimization in AMOS2, of the algorithm you should use, and some usable LISP functions.

Create a copy of the code skeleton from:

```
~dbtek/kursbibliotek/amos/kodskelett.lsp
```

Save it in a file, e.g. `lab7.lsp`, edit your changes, and load it into AMOS2 (see below).

As example database you should use the soccer database from Exercise 6.

Create the World Cup database in the file `wc.dmp` and start AMOS2 with it by:

```
amos2 wc.dmp
```

As default AMOS2 uses a 'greedy' heuristic for query optimization. To change to exhaustive search do

the AMOSQL command:

```
optmethod('exhaustive');
```

When you have given the above command the regular query optimizer will call `dynprogsort`. Don't forget this before testing!! The exhaustive optimization is not supported in the version of AMOS2 you are running and it is the task of this exercise to implement it.

Another important issue is, to make your task simple, you should assume that the pre-optimized query predicates are always transformed to *disjunctive normal form* (DNF). You must instruct the optimizer to transform the predicates to DNF by evaluation the following Lisp expression before running your program:

```
(setq _use_dnf_ t)
```

Hand in all your code to the lab assistant along with examples of optimized queries (from Exercise 6). Also give a search path so the assistant can test your algorithm.

Query optimization in AMOS2

The internal representation of query plans in AMOS2 is based on the logical language ObjectLog [3]. ObjectLog can be seen as an object-oriented variation of Datalog. Datalog [7] is a subset of Prolog developed for relational databases. Predicates in Datalog cannot have function symbols (functors) as arguments.

In fig. 1 a simplified description is given of the steps involved in query optimization in AMOS2:

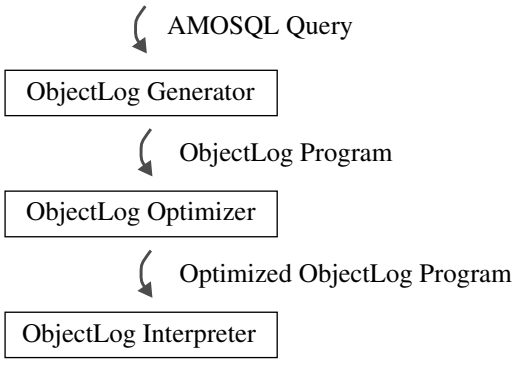

Figure 1: Query processing in AMOS2

Consider the following AMOSQL query:

What matches in 1950 had more than 100000 spectators?

```
select m
from match m
where spectators(m)>100000
   and year(played_in(m))=1950;
```

This query will be translated to an unoptimized ObjectLog program similar to this:

```
answer(M)          <- spectators_match->integer(M,_V1) &
                      played_in_match->tournament(M,_V2)
                      year_tournament->integer(_V2,1950) &
                      >(_V1,100000)
```

ObjectLog predicates are annotated with the signatures of the corresponding AMOSQL functions. They are overloaded on the signatures, i.e. defining the same function for different types.

The optimized version of the above ObjectLog program looks like:

```
answer^f(M)        <- year^fb_tournament->integer(_V2,1950) &
                      played_in^fb_match->tournament(M,_V2) &
                      spectators^bf_match->integer(M,_V1) &
                      >^bb(_V1,100000)
```

Optimized ObjectLog predicates are not only overloaded on the argument types, but also on binding patterns. When the execution order of the ObjectLog predicates is decided it is also known which of the arguments will be bound at the time of execution. The binding pattern 'fb' for the year predicate above means that the first argument will be 'free' and the second will be 'bound'. The binding pattern 'bb' for the '>' predicate means that both of the arguments will be 'bound' at the time of execution (the predicate will thus function as a boolean test).

The interpreter executes the optimized ObjectLog program using the 'nested loop' method. I the above program the interpreter would start with fetching the tournament (_V2) that was had in 1950, then search for a match (M) that was played in that tournament. When such a match is found the interpreter continues with fetching the number of spectators for that match and finally test if this number exceeds 100000. If it does then the match M will be returned as the answer to the query. AMOS2 then 'back-tracks' and tries to find more matches that were played in the tournament _V2 and continues until all tournaments (in 1950) with corresponding matches have been processed.

Internally ObjectLog predicates are represented as LISP lists. The body in the unoptimized ObjectLog program above would be represented by the following list (l1):

```
(AND (P_MATCH.SPECTATORS->INTEGER M _V_NIL_1)
     (P_MATCH.PLAYED_IN->TOURNAMENT M _V_NIL_2)
     (P_TOURNAMENT.YEAR->INTEGER _V_NIL_2 1950)
     (OBJECT.OBJECT.>->BOOLEAN _V_NIL_1 100000)
```

The body in the optimized ObjectLog program above would be represented by the following list (l2):

```
(AND (P_TOURNAMENT.YEAR->INTEGER _V_NIL_2 1950)
     (P_MATCH.PLAYED_IN->TOURNAMENT M _V_NIL_2)
     (P_MATCH.SPECTATORS->INTEGER M _V_NIL_1)
     (CALL GT-- #[OID 91 OBJECT.OBJECT.>->BOOLEAN] _V_NIL_1 100000))
```

! In this exercise you are to write the LISP function `dynprogsort` that takes an unoptimized list of ObjectLog predicates and returns the permutation that equals the cheapest query plan[1]. Thus if the list l1 is sent to the function it should return l2.

Physical storage

Stored functions in AMOSQL have the relationship between the function arguments and the result directly stored in the database.The relationship is stored internally as tables. For example, for the function *year(tournament)->integer* there exist a table *P_tournament.year->integer* with two attributes (of the types *tournament* and *integer*) and 14 tuples (one for each tournament).

The cost model

The AMOS2 optimizer uses two types of estimated values for finding the cheapest query plan: (1) the cost for executing a given predicate given a certain binding pattern and (2) the size of the result of an execution (fanout). The cost is defined as 2 * the number of visited tuples. Fanout is the number of tuples in the result.

Assuming there is no index on the year for the function *year(tournament)->integer* then the cost for executing the predicate:

`year`$_{tournament\text{->}integer}$`(_G2,1950)`

with the binding pattern 'fb' will be 2*14 = 28 since all the 14 tuples have to be scanned. Fanout will be 1 since the result relation contains *one* tuple (1950 the was *one* tournament organized).

Dynamic programming

Assume that an ObjectLog program consists of three predicates (subgoals):

```
answer(X)          <- X & Y & Z
```

There are 6 different permutations of the predicates (XYZ, XZY, YXZ, YZX, ZXY and ZYX). A search tree to find the cheapest permutation could look like the tree in figure 2.

Consider the query plan XYZ, i.e. the path from the root to the leaf furthest to the left in figure 2. The cost for executing X first is estimated to 100 and the expected number of results (fanout) is 50. The cost for executing Y afterwards is estimated to 200 and fanout is estimated to 1. Since the previous fanout was 50 the total cost so far is 100+200*50=10100. Fanout after executing XY is 50*1=50. The cost for finally executing Z is estimated to 8. The total cost for executing the query plan XYZ is thus 10100+8*50=10500.

1. ObjectLog programs are transformed to disjunctive normal form if the LISP variable `_USE_DNF_` is set to T. The task of the LISP-function `dynprogsort` is to optimize a conjunction of ObjectLog predicates.

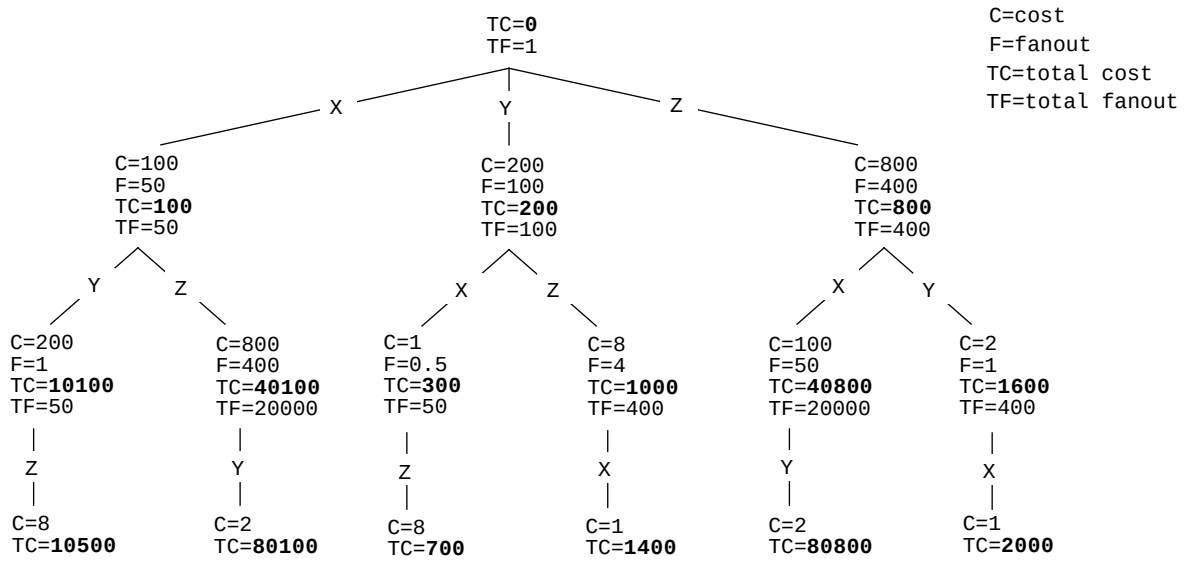

Figure 2: Search-tree for the example query

As can be seen in figure 2 the cheapest query plan is YXZ (with an estimated cost of 700). The algorithm that we shall use here is based on search downward in the branch that is currently the cheapest. If we come to a leaf-node and there is no alternative (sub-) plan with a lower cost then we can stop the search. Since the cost grows monotonically we are guaranteed to have found the cheapest query plan.

In the example above we start with estimating the cost and fanout for executing each of the three predicates X, Y, or Z, first (figure 3a). The cheapest branch in the tree is X with a cost of 100, and then fol-

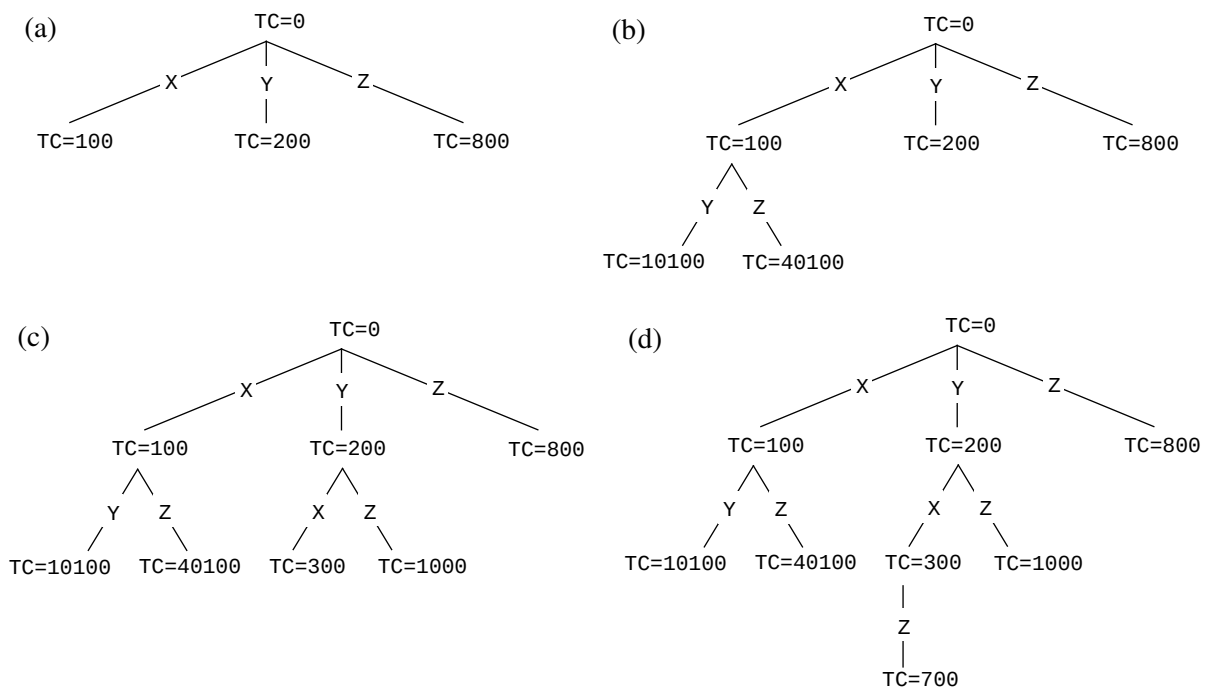

Figure 3: Algorith for finding the cheapest query plan

lows Y (cost 200) and Z (cost 800). We create a queue (X, Y, Z) with these incomplete query plans. The

queue must be kept sorted on the total cost all the time. The next step is to estimate the cost of every continuation after X, i.e. XY and XZ (figure 3b). We put these plans in the queue and remove X: (Y, Z, XY, XZ). The cheapest branch in the tree is now Y with a cost of 200. The cost for all possible continuations after Y, i.e. YX and YZ are estimated (figure 3c). Y is now removed and the new plans are put in the queue: (YX, Z, YZ, XY, XZ). The cheapest branch in the tree is now YX with a cost of 300. After YX there is only one possibility, YXZ, and the cost for this plan is estimated (figure 3d). The cheapest branch is now YXZ (with a cost of 700) since this is a leaf-node i the complete search tree (figure 2) the search for the cheapest query plan is now finished.

ALisp

In AMOS2 there is a complete built-in LISP-interpreter called ALisp. The LISP-dialect is a large subset of CommonLisp. ALisp can be started inside AMOS2 by the AMOSQL command:

```
lisp;
```

To get back to the AMOSQL-prompt one can give the following expression to the ALisp interpreter:

```
:osql
```

ALisp differs from CommonLisp in some ways. Ask the assistant if you run into problems.

Debugging

The easiest way to debug your LISP functions is to add trace printouts yourselves in the code (with prin1, princ, print, ...). In ALisp there are also some debugging tools:

Break points can be put into LISP-functions with the macro:

```
(break fn1 fn2 ...)
```

For example:

```
(break dynprogsort)
```

The execution will be interrupted every time a 'broken' function is started. The. ALisp will then enter into a 'break-loop' inside the following commands can be given:

`?=`

Prints the values of the actual arguments to the broken function.

`:eval`

Evaluates the body of the function and enters the 'break-loop' again.

`!value`

Is a special variable that contains the value of the function that has just been evaluated with *:eval*. Can only be accessed directly after *:eval*.

`:c`

Continues the execution, i.e. exist the break-loop.

`:r`

Resets the ALisp interpreter.

`:ub`

Removes the breakpoint from the current function.

Break points can be removed by:

```
(unbreak fn1 fn2 ...)
```

For example:

```
(unbreak dynprogsort)
```

Traces of function calls can be printed out by:

```
(trace fn1 fn2 ...)
```

For example:

```
(trace dynprogsort)
```

Trace printouts can be cancelled by:

```
(untrace fn1 fn2 ...)
```

For example:

```
(untrace dynprogsort)
```

Useful ALisp-functions

`(objlog query)`
Prints out the internal representation of the ObjectLog program that the *query* is compiled to. First the unoptimized program, and then the optimized. Don't forget the semicolon (;) after the query.

For example:

```
(objlog "select m from match m where spectators(m)>100000 and
year(played_in(m))=1950;")
```

`(load file-name)`
Loads and evaluates a file of LISP code (S-expressions).

For example:

```
(load "lab7.lsp")
```

`(pp fn)`
Prints (pretty prints) the definition of the LISP-function *fn*.

For example:

```
(pp dynprogsort)
```

`(amosql query)`
Executes the AMOSQL *query.*

For example:

```
(amosql "select m from match m where spectators(m)>100000 and
year(played_in(m))=1950;")
```

`(timer S-expression)`
Executes *S-expression* and prints out used CPU-time.

For example:

```
(timer (amosql "...;"))
```

Various optimization functions

The following LISP functions can be very useful when you write the function *dynprogsort*.

`(simple-pred-cost pred bpat)`
Returns (lc . fo) where *lc* is the estimated cost and *fo* is the estimated fanout for executing the predicate *pred* with the binding pattern *bpat*. Returns NIL if the predicate with that binding pattern is not possible to execute.

For example:

```
(setq pred (list (getfunctionnamed 'p_tournament.year->integer) '_v2 1950))
=> (#[OID 356 P_TOURNAMENT.YEAR->INTEGER] _V2 1950)
(simple-pred-cost pred '(+ -))
=> (50 . 1.78571)
```

(**bindadornpat** pred bound)

Returns the binding pattern for *pred*, i.e. a list with + (denoting a free variable) and - (denoting a bound variable). If, for example, variable 2 and 4 (of 4) in pred are bound the pattern '(+ - + -) is returned.

For example:

```
(setq pred (list (getfunctionnamed 'p_tournament.year->integer) '_v2 '_v3))
=> (#[OID 356 P_TOURNAMENT.YEAR->INTEGER] _V2 _V3)
(bindadornpat pred '(_v3))
=> (+ -)
```

(**substbindadorned** pred bpat)

Chooses the correct function (predicate) `pred` for a particular binding pattern `bpat` (found by `bindadornpat`).
For example:

```
(substbindadorned
        (list(getfunctionnamed 'object.object.>->boolean)'_v1 10000)
        '(- -))
=> (CALL GT-- #[OID 91 OBJECT.OBJECT.>->BOOLEAN] _V1 10000)
```

(**pred_binds** pred vars)

Returns the union of *vars* and the variables in *pred*.

For example:

```
(setq pred (list (getfunctionnamed 'number.number.plus->number) 17 42 '_v1))
=> ([#OID 75 NUMBER.NUMBER.PLUS->NUMBER] 17 42 _v1)
(pred_binds pred '(X Y Z))
=> (_V1 X Y Z)
```

PLANINFO-structures

(defstruct planinfo *plan bound rem cost fanout*)

Can be used for storing information about (incomplete) execution plans. Contains the fields:

plan	The (incomplete) plan that has been built so far
bound	The variables that are bound when PLAN has been executed
rem	The predicates that remain to order (the predicates that are not yet in PLAN)
cost	The cost for executing PLAN
fanout	The fanout when PLAN has been executed

The function print-planinfo is useful when debugging:

```
(defun print-planinfo (p)
  (formatl t
           " plan:" (planinfo-plan p) t
           " bound:" (planinfo-bound p) t
           " rem:" (planinfo-rem p) t
           " cost:" (planinfo-cost p) t
```

```
" fanout:" (planinfo-fanout p) t))
```

Storage structure for the queue of PLANINFO-structures

In a real implementation it is important that searches and updates of a queue are very efficient. Because of this some efficient storage structure should be used such as an indexed tree. In this exercise it is OK to use ordinary LISP-lists to represent a queue. Talk to the lab assistant if you want to use some more efficient data structure that already exist in AMOS2.

References

[1] Y.E.Ioannidis, Y.C.Kang: Randomized Algorithms for Optimizing Large Join Queries, *Proc. ACM SIGMOD Conf.*, Atlantic City, 1990.

[2] S.Flodin, T.Risch, M.Sköld: *AMOS2 User's Guide*, **http://www.ida.liu.se/~dbtek/amos2/amosdoc.html**.

[3] W.Litwin, T.Risch: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992.

[4] J.Näs: *Randomized Optimization of Object Oriented Queries in a Main Memory Database Management System*, Master's Thesis, LiTH-IDA-Ex-9325, Linköping University 1993.

[5] T.Risch: *ALisp User's Guide,* **http://www.ida.liu.se/~dbtek/amos2/alisp.fm.pdf**

[6] P.G.Selinger et al.: Access Path Selection in a Relational Database Management System, *Proc. ACM SIGMOD '79*, 1979.

[7] J.D.Ullman: *Database and Knowledge-Base Systems*, Volume I, Computer Science Press, 1988.