# 1-Phase Optimizer for the SimputerDB
# Final Report
# CS - 631 DBMS

Kriti Puniyani - 04305012 - kriti@cse.iitb.ac.in
Neela Sawant - 04305811 - neela@cse.iitb.ac.in

October 20th, 2004

# Contents

# 1 Introduction

The Simputer is a low cost portable alternative to PCs, by which the benefits of IT can reach the common man.

The Simputer is meant to be a mass-access device,and hence to handle a large volume of data and queries which may be simple or complex e.g. select,project,aggregate and join. Most of the queries need to be executed on the Simputer itself to reduce communication cost.Also the need to execute complex queries is increasing by the day. Since the Simputer inherently has limited memory and computational resources, the query plans should be generated such that they make optimal use of available memory and resources i.e. query optimization is a must.

Typical optimizers assume all the memory to be available to each of the operators in the query tree. This assumption obviously does not work for a memory constrained device like Simputer. Memory will get divided amongst all the operators running simultaneously in the pipeline.

This report describes the implementation of the 1-phase query optimizer for Simputer DB that is memory cognizant. The query optimizer considers all possible alternatives of evaluating a query, finding the cost difference between a good and a bad alternative and generating an optimal solution. For example: performing a (r X s) followed by a selection (r.A = s.B) is much slower than performing a join on the same condition. So in our implementation we have tried to implement an efficient query optimizer that will generate a solution which

- Minimize writes to secondary storage
- Uses main memory as write buffer (No materialization of intermediate results)
- Optimize query processing (Cheaper cost of the final plan than that of 2-phase optimizer)

# 2 Description of the System

## 2.1 Overall perspective

## 2.2 Functional Requirements of the System

The 1-phase optimizer has to input a query, examine all possible execution plans for the execution of that query and select the overall cheapest one to be used to generate the answer of the original query.

It must implement the following functions:

- Determine the complete query DAG for the input query. Each path in the DAG will produce the same output but at different cost.
- Determine the optimal division of available memory between the various operators in the query.
- Determine the optimal implementations choices such as the join order for evaluating each subexpression in the query. This is related to the physical schema of the database.
- Find the cost associated with the step 1 and step 3. A formula is used to find the cost of each join method,access type and in general every distinct step that can be found in the execution plan.
- Find the execution order and algorithm which minimizes the cost - this is the final optimized query.

## 2.3 Algorithm Used

- Expand the DAG to find all possible permutations of the operator order by applying all possible transformations on every node of the initial query DAG to create a fully expanded logical AND-OR DAG.
- Call a search procedure on the root of this Logical AND-OR DAG, which is nothing but an equivalence node, using the modified Volcano search algorithm. Effectively, what we are doing is computing for each possible permutation, the cost of that permutation, by optimizing memory allocation- to each operator in the permutation. This is achieved by:

  - Evaluate cost function for all execution schemes (NLJ, INLJ, HJ etc.) and merge them into a single cost function for the operator using MinMerge algorithm.
  - Use the OptMerge algorithm to find the optimal memory allocated to each operator, based on its cost. This is a recursive procedure.
  - Dynamic Programming is used to store results of lower levels in the DAG so that they are not recomputed and the algorithm can be computed in polynomial time.

- Find the permutation that gives the optimal i.e. least cost and this is our optimal join order. The optimal join order is also associated with the memory allocated to each operator since that was used as the evaluation parameter.

- Depending upon the memory allocated to an operator, find the optimal scheme to be used in implementing that operator. This can be be done by referencing the cost function evaluated in step 2(1).

- Return the left-deep tree join order, the memory allocated to each operator and the scheme used for each operator, as the final optimized query.

## 2.4 Issues in the 1-Phase Query Optimizer

### 2.4.1 Evaluation of Cost of Operators

Suppose there are n schemes to implement operator o. All these schemes have a cost and memory associated with them. The cost function of an operator is a collection of (memory, cost) points of its schemes. It is a piecewise linear function. For the join algorithm, the cost and memory for the various schemes is calculated as follows:

**NLJ(Nested Join Algorithm)** Cost: $(m * n) * (2K)$
Memory: 0

**INLJ(Indexed Nested Join Algorithm)** Cost: $m * (2K + K * logm) + N * (K * logm + k)$
Memory: $(p + L) * m$

**HJ (Hashed Join Algorithm)** Cost: $m*(2K+hash)+N*(hash+(m/b)*(3k))$
Memory: $(2p*m)*f$

where
m - no of tuples in input R
n - no of tuples in input S
K - time to read / write / compare the join keys
p - size of pointer
L - avg length of join attribute
f - hash table fudge factor
hash - time to hash a key

Using these formulae, the cost function of each scheme is computed. All schemes of a given operator are then combined to find the total function using MinMerge algorithm, which compares the input cost functions for the entire memory range and for each memory point, picks up the lowest cost value. The operator cost function is thus a piecewise linear function.

### 2.4.2 Size Estimation of Resultant relations

Size estimation of intermediate results is an important part of a query optimizer since cost estimation depends on the sizes of the relations. We have modified the heuristic used in the size estimation of the 2PO since for very large relations, size of resultant is usually not a multiple of the 2 sizes, but instead usually a small multiple of the larger relation. Our Heuristic is:

1. If the join attribute is a primary key of left relation, then $size = (sizeofrightrelation)$.
2. If the join attribute is a primary key of right relation, then $size = (sizeofleftrelation)$.
3. Else if the size of both relations is less than 500, then $size = (left_rel_size * right_rel_size)/2$
4. Else if the size of one of the relations is greater than 500, then $size = (left_rel_size * right_rel_size)/5$
5. Else if the size of both the relations is greater than 500, then $size = (left_rel_size * right_rel_size)/10$

### 2.4.3 Computing the optimal memory allocation

We are given n operators and their piecewise linear functions, along with available memory and we need to compute the optimal memory for each operator. The OptMerge memory allocation algorithm is used for the same. It processes 2 operators at a time and tries to find their optimal memory division. If the operators are o1 and o2, with the cost functions c1 and c2 then the OptMerge procedure simply allocates maximum possible memory to the operator which gains more per unit memory allocation i.e. ha lower slope in that range of memory.Since the cost functions are all piecewise linear, cost function testing is done at change-over points.
The algorithm is discussed in detail in the design document.

### 2.4.4 Execution Order

Due to small amount of main memory, and expensive flash memory writes, we want to minimize materialization of intermediate results in secondary storage. This implies that a left deep tree should be used. Left-deep tree is most suitable for pipelined execution since the the right operand is a stored relation and only the left operand is pipelined. Hence, we have emphasized on the left-deep tree execution order.

### 2.4.5 Implementation choices of algorithms for operators

Every operator has multiple algorithms by which it may be executed. For example, join may be implemented using nested loops (fully pipelineable schedule with minimum memory usage, but multiple scans over the tuples in the inner relation), hash join,merge join (faster but needs more storage). Similarly a number of algorithms exist for GroupBy and other such operators. A selection needs to be made between these options. There is usually a tradeoff between memory and performance, and for the Simputer DB, memory being limited, choices reflect this constraint. However the amount of memory is increasing with every handheld, and we don't want to restrict memory usage to the bare minimum at the cost of performance. Also the choice of algorithm is usually made beforehand (static choice). However we want the choice of algorithm to be dynamic. Hence a dynamic strategy that ensures the best query execution plan for every device depending on the available memory is used by the optimizer to maximize performance.

### 2.4.6 Support for Aggregate Operators

If we want to include support for aggregate operators, then the changes required will include:

- The parser and executer should be able to handle aggregate operators.
- Additional data structures will have to be defined to store the type of operator (min, max, avg, sum, count), the attributes on which aggregation is to be performed, and the group-by attributes.
- Since selection must occur before aggregation, and only 1 selection can be pushed down the QEP tree, hence the aggregation also takes place at the root of the QEP.
- Hence the complexity of the Query optimizer will not increase drastically due to the inclusion of support for aggregate operators.
- The major challenge would be to keep track of the interesting sort orders of the joins, and be able to give more weightage to such operator orders.

# 3 Implementation Details

## 3.1 Input

- Logical Expression of the query in the form of a query DAG (Directed Acyclic Graph)
- The amount of memory available

## 3.2 Output

- The optimal memory allocated to each operator
- The optimal scheme used for each operator
- The optimized left deep tree that defines the operator order

## 3.3 Assumptions

- Currently, we are defining the algorithm to be able to deal only with join. Support for determining left deep tree for aggregation is not provided.
- Also we assume that a query executer is defined which is able to execute the query based on the instructions generated by the optimizer.

## 3.4 Important Data Structures

The data structures used in the above algorithm include:

**PlanNode** Store the relationId and no of tuples in the plan

**PlanJoinInfo** Store the 2 PlanNodes participating in the join, joining attribute, selection ratio etc.

**SchemeInfo** Keep track of the information about the various schemes of the operator including cost, memory, size, and schemeId. The cost and memory will be defined using an array of structures to specify the various change-over points.

**OperatorInfo** Keep track of an operator by defining its opId and the list of schemes stored in SchemeInfo.

**RelInfo** Keep Track of estimated Number of tuples in the resultant relation.

**HashPlanNode** Keeps the hashed intermediate results for use in Dynamic Programming.

**JoinAttrNode** Stores the information about the attributes on which 2 relations are joined.

**SelRelInfo** Stores the selection criteria of each relation in the query

**DispInfo** Stores the attributes of the resultant relation which need to be displayed in the final result.

## 3.5 Implementation level Algorithm

The parser in src/backend/parser/parser.y parses the given query, and passes the various parameters to the test( ) function of optimizer defined in src/backend/optimizer/bestplan.c.

- The test function sets up its own structures and passes the information to FindBestPlan( )
- FindBestPlan( ) now generates all n! possible query execution plans using recursive calls i.e. expands the DAG, and calls memalloc( ) function in memalloc.c for each query plan to find cost of the plan.
- memalloc( ) uses dynamic programming to store intermediate results, sets up size information of the relations, and again calls SetUpOperatorInfo( ) which sets up the info about each possible scheme for the operators, and then call MemAllocate( ).
- MemAllocate( ) finds best scheme for each operator and optimal memory allocation using GetBestScheme( ) and returns the minimum cost and scheme info which is returned to FindBestPlan( ).
- FindBestPlan( ) stores the bestplan found till now, and if the returned plan is better, it stores it as best plan.

- When all possible QEPs have been generated and evaluated, control returns to test, and bestplan stores the best QEP possible, which is passed to the executer by calling the execute_init( ) function defined in src/backend/executer/iterator.c.

- The executer evaluates the query plan given to it and displays the results as expected.

All interfaces have been defined in detail in section 3.6

## 3.6 Integration of code

The code has been integrated with presently existing code. Hence to use the 1-Phase Optimizer inplace of the 2-Phase Optimizer, the files modified (listed in section 4.2) of the present code must be replaced by the files submitted by us.

The interface is as follows:

- The 1-Phase Optimizer is called by the src/backend/parser/parser.y as:
  test(rellist,relnames,displaylist,selectionlist,joinattrlist,keyinfo,planhashlist)
  where rellist - relation ids that are involved in the join.
  relnames - names of the relations in order
  displaylist - list of attributes to be displayed
  selectionlist - selection criteria on each relation
  joinattrlist - list of attributes on which each relation may be joined with other relations
  keyinfo - primary keys for each relation
  planhashlist - used for hashing intermediate plans in dynamic programming

- The output of the 1-Phase Query Optimizer is given to the src/backend/executer/iterator.c by calling its function:
  execute_init(plantemp,joinattrlist,selectionlist,relnames,keyinfo,dispinfolist,Scheme);
  where plantemp - Best Plan achieved after Optimization
  Scheme - Scheme used for each Operator

# 4 Compilation and Execution of the code:

## 4.1 System Requirements

1. Development environment:
   Language : C
   Compiler : gcc version 2.96
   Tools used : gdb for debugging
   Packages : stdio.h, fcntl.h, assert.h, math.h, stdlib.h, sys/time.h, io.h, errno.h, string.h, sys/stat.h

2. The Simputer DB has the following specifications.

   - Hardware
     CPU 32-bit Strong Arm SA-1100 RISC CPU running at 200MHz
     32 MB of DRAM
     24 MB Flash for Permanent Storage (DOC)
     Display I/F 320x240 Monochrome LCD Display Panel

   - System Software
     Operating System: GNU/Linux
     Soft-Modem Algorithms V.34/V.17 Data/Fax Modem Technology
     Perl/Tk scripting environment

However, we have not run our code on the SimputerDB but instead have tested only on PCs.

## 4.2 Directory Structure:

Since the code is integrated with the currently existing code, hence the files coded are included in the src/backend/optimizer/ and /src/include/optimizer directories of the simpdb directory ( /mtpji/simpdb)

1. Files Modified:

   - src/backend/optimizer/bestplan.c

- src/backend/optimizer/memalloc.c
- src/include/optimizer/optimizer.h
- src/include/optimizer/optimizerfunc.h
- src/include/optimizer/optimizerexecuter.h

2. Major Functions written:
   - memalloc()
   - FindBestPlan()

3. Helper Functions written:
   - InsertPlan()
   - DeletePlan()
   - printPlan()
   - traverseJoinAttr()

4. Important Modifications Done in: (including changing the interfaces )
   - test()
   - SetUpOperatorInfo()
   - MemAllocate()
   - GetBestScheme()

## 4.3 Scripts

We have provided shell scripts for :

1. ./m - Compiles all files, and sets up the data files
2. ./i - Creates and inserts data into the required relations
3. ./q - Runs the q1.sql example
4. All other queries can be run by the command:

./bin/sqlplus ¡Query-Name¿ eg. ./bin/sqlplus ./examples/query/q1.sql

Note that all shell scripts must be present in the simpdb directory and all other files have their path-names in reference to the simpdb directory.

A generalized script for time-estimation has been provided as time_estimate.sh which estimates all time parameters, i.e. time comparison of 1PO and 2PO.

## 4.4 Test Cases

The queries for the testing has been placed in the test_cases directory, and can be run by
./bin/sqlplus ./test_cases/query-name

The queries involving customer has the relations placed in the cust directory. These can be set up by:
./customer_i
which will set up the relations. The code also needs to be changed in the src/backend/parser/parser.y by changing the key information (by simply commenting out current info and replacing it by the commented code of the customer relations info) and then the queries can be run as usual from the test_cases directory.

# 5 Performance Measurement

## 5.1 Test Cases And Results for 1-Phase and 2-Phase Optimizers

A major limitation of the parser is that it is unable to set up the information about the primary keys for flat storage, and hence hard-codes these parameters. Hence we were unable to test the optimizer with different relations. Hence our test cases were run by changing the key-info parameters in the src/backend/parser/parser.y file

The results were as follows:
1. The Relations used were:
CREATE TABLE DOCTOR(DOCID INTEGER,NAME CHAR(20));
CREATE TABLE PRES(VISITID INTEGER,DRUGID INTEGER);
CREATE TABLE VISIT(VISITID INTEGER,DOCID INTEGER,DATE INTEGER);
CREATE TABLE DRUG(DRUGID INTEGER,TYPE CHAR(20));

5 queries were run with these relations and the results obtained were as follows:

- **q1.sql:**
  Query : SELECT DOCTOR.NAME FROM DOCTOR,PRES,DRUG,VISIT
  WHERE VISIT.DATE=1998 AND DOCTOR.DOCID=VISIT.DOCID
  AND DRUG.DRUGID=PRES.DRUGID AND PRES.VISITID=VISIT.VISITID
  AND DRUG.TYPE='Antibiotic' ;

  Number of selected rows : 2155

  1. **1-Phase Optimizer:**
     The Final Plan is :
     3 VISIT(270) 0 DOCTOR(270) 1 PRES(2155) 2 DRUG(2155)
     VISIT scheme 3 DOCTOR scheme 1 PRES scheme 3 DRUG
     setting inlj join setting nlj join setting inlj join

Final Minimum Cost = 1182797

2. **2-Phase Optimizer:**
   The Final Plan is :
   3 VISIT(270) 0 DOCTOR(270) 1 PRES(2155) 2 DRUG(2155)
   VISIT scheme 3 DOCTOR scheme 1 PRES scheme 3 DRUG
   setting inlj join setting nlj join setting inlj join

   Final Minimum Cost = 1545540

- **q2.sql:**
  Query : SELECT * FROM DOCTOR,PRES,DRUG,VISIT WHERE VISIT.DATE=1998 AND DOCTOR.DOCID=VISIT.DOCID AND DRUG.DRUGID=PRES.DRUGID AND PRES.VISITID=VISIT.VISITID AND DRUG.TYPE='Vitamins';

  Number of selected rows : 50

  1. **1-Phase Optimizer:**
     The Final Plan is :
     3 VISIT(270) 0 DOCTOR(270) 1 PRES(2155) 2 DRUG(2155)
     VISIT scheme 3 DOCTOR scheme 1 PRES scheme 3 DRUG
     setting inlj join setting nlj join setting inlj join

     Final Minimum Cost = 1182797

  2. **2-Phase Optimizer:**
     The Final Plan is :
     3 VISIT(270) 0 DOCTOR(270) 1 PRES(2155) 2 DRUG(2155)
     VISIT scheme 3 DOCTOR scheme 1 PRES scheme 3 DRUG
     setting inlj join setting nlj join setting inlj join

     Final Minimum Cost = 1545540

- **q3.sql:**
  Query : SELECT DRUG.TYPE FROM DRUG,PRES
  WHERE DRUG.DRUGID=PRES.DRUGID;

  Number of selected rows : 2155

  1. **1-Phase Optimizer:**
     The Final Plan is :
     1 PRES(2155) 0 DRUG(16593)
     PRES scheme 3 DRUG
     setting nlj join

     Final Minimum Cost = 16296

  2. **2-Phase Optimizer:**
     The Final Plan is :
     1 PRES(0) 0 DRUG(82967)
     PRES scheme 0 DRUG
     setting nlj join

     Final Minimum Cost = 331870

- **q4.sql:**
  Query : SELECT * FROM DOCTOR,PRES,VISIT WHERE DOCTOR.DOCID=VISIT.DOCID AND VISIT.VISITID=PRES.VISITID AND DOCTOR.DOCID=65;

  Number of selected rows : 71

  1. **1-Phase Optimizer:**
     The Final Plan is :
     0 DOCTOR(1) 2 VISIT(830) 1 PRES(178865)
     DOCTOR scheme 1 VISIT scheme 3 PRES
     setting nlj join setting inlj join

Final Minimum Cost = 3578960

2. **2-Phase Optimizer:**
   The Final Plan is :
   0 DOCTOR(1) 2 VISIT(830) 1 PRES(894325)
   DOCTOR scheme 1 VISIT scheme 1 PRES
   setting nlj join setting nlj join

   Final Minimum Cost = 3579051

- **q5.sql:**
  Query : SELECT * FROM DOCTOR,PRES,DRUG,VISIT WHERE VISIT.DATE=1998 AND DOCTOR.DOCID=VISIT.DOCID AND DRUG.DRUGID=PRES.DRUGID AND PRES.VISITID=VISIT.VISITID AND DRUG.TYPE='Dupont';

  Number of selected rows : 41

  1. **1-Phase Optimizer:**
     The Final Plan is :
     3 VISIT(270) 0 DOCTOR(270) 1 PRES(2155) 2 DRUG(2155)
     VISIT scheme 3 DOCTOR scheme 1 PRES scheme 3 DRUG
     setting inlj join setting nlj join setting inlj join

     Final Minimum Cost = 1182797

  2. **2-Phase Optimizer:**
     The Final Plan is :
     3 VISIT(270) 0 DOCTOR(270) 1 PRES(2155) 2 DRUG(2155)
     VISIT scheme 3 DOCTOR scheme 1 PRES scheme 3 DRUG
     setting inlj join setting nlj join setting inlj join

     Final Minimum Cost = 1545540

2. The Relations used were:
CREATE TABLE CUSTOMER(CUSTOMERID INTEGER,NAME CHAR(20));
CREATE TABLE BILL(VISITID INTEGER,ITEMID INTEGER);
CREATE TABLE VISIT(VISITID INTEGER,CUSTOMERID INTEGER,DATE INTEGER);
CREATE TABLE ITEM(ITEMID INTEGER,TYPE CHAR(20));

- **q6.sql:**
  Query : SELECT * FROM BILL, ITEM WHERE BILL.ITEMID=ITEM.ITEMID AND ITEM.TYPE='Hockey' ;

  Number of selected rows : 38

  1. **1-Phase Optimizer:**
     The Final Plan is :
     1 ITEM(1) 0 BILL(215)
     ITEM scheme 1 BILL
     setting nlj join

     Final Minimum Cost = 4310

  2. **2-Phase Optimizer:**
     The Final Plan is :
     1 ITEM(1) 0 BILL(1077)
     ITEM scheme 1 BILL
     setting nlj join

     Final Minimum Cost = 4320

- **q7.sql:**
  Query : SELECT * FROM CUSTOMER, ITEM, VISIT, BILL WHERE CUSTOMER.CUSTOMERID=VISIT.CUSTOMERID AND ITEM.ITEMID=BILL.ITEMID AND BILL.VISITID=VISIT.VISITID AND ITEM.TYPE='Carrom' ;

Number of selected rows : 11

1. **1-Phase Optimizer:**
   The Final Plan is :
   1 ITEM(1) 3 BILL(215) 2 VISIT(17845) 0 CUSTOMER(69595)
   ITEM scheme 1 BILL scheme 3 VISIT scheme 3 CUSTOMER
   setting nlj join setting inlj join setting inlj join

   Final Minimum Cost = 473657

2. **2-Phase Optimizer:**
   The Final Plan is :
   3 BILL(0) 2 VISIT(830) 0 CUSTOMER(16185) 1 ITEM(80925)
   BILL scheme 1 VISIT scheme 3 CUSTOMER scheme 3 ITEM
   setting nlj join setting inlj join setting inlj join

   Final Minimum Cost = 3965740

- **q8.sql:**
  Query : SELECT * FROM CUSTOMER,BILL,VISIT WHERE CUSTOMER.CUSTOMERID=VISIT.CUSTOMERID AND VISIT.VISITID =BILL.VISITID AND CUSTOMER.CUSTOMERID=20;

  Number of selected rows : 102

  1. **1-Phase Optimizer:**
     The Final Plan is :
     0 CUSTOMER(1) 2 VISIT(83) 1 BILL(17886)
     CUSTOMER scheme 1 VISIT scheme 3 BILL
     setting nlj join setting inlj join

     Final Minimum Cost = 359390

  2. **2-Phase Optimizer:**
     The Final Plan is :

0 CUSTOMER(1) 2 VISIT(415) 1 BILL(447162)
CUSTOMER scheme 1 VISIT scheme 1 BILL
setting nlj join setting nlj join

Final Minimum Cost = 1790349

## 5.2 Time Performance Analysis

| q1.sql | | |
|---|---|---|
| | 1PO | 2PO |
| **Final Sum** | 1299.556459969 | 1196.115409155 |
| **Final Count** | 1000 | 1000 |
| **Average Time** | 1.299556459969 | 1.196115409155 |
| **q2.sql** | | |
| | 1PO | 2PO |
| **Final Sum** | 1167.142231869 | 1152.696781472 |
| **Final Count** | 1000 | 1000 |
| **Average Time** | 1.167142231869 | 1.152696781472 |
| **q3.sql** | | |
| | 1PO | 2PO |
| **Final Sum** | 352.502123875 | 258.281433075 |
| **Final Count** | 1000 | 1000 |
| **Average Time** | 0.352502123875 | 0.258281433075 |
| **q4.sql** | | |
| | 1PO | 2PO |
| **Final Sum** | 482.015082933 | 347.00873464 |
| **Final Count** | 1000 | 1000 |
| **Average Time** | 0.482015082933 | 0.34700873464 |
| **q5.sql** | | |
| | 1PO | 2PO |
| **Final Sum** | 1284.116834549 | 1197.498557546 |
| **Final Count** | 1000 | 1000 |
| **Average Time** | 1.284116834549 | 1.197498557546 |

As seen in the above table, the time estimation was done by running a number of queries 1000 times each and finding the average time for each query execution, for both 1-Phase and 2-Phase Query Optimizer.

However, it must be noted that since the executer currently available does not utilize the memory allocation results, hence the time estimates do not accurately reflect the performance of the 2 Optimizers. Hence we must rely on the theoretical Analysis of the Optimizers to draw our conclusions.

It is found that although the 1-Phase Query Optimizer is able to give more accurate cost estimations, the following important points must be noted:

- An improvement in terms of a more optimized query is observed very rarely.
- Time required for cost estimation increases only by a marginal amount.
- Space required for query optimization increases due to use of dynamic programming.

In a Simputer DB, if the queries are not very complicated, then the improvement in the query optimization will not be observed, while if the query is very complicated, then due to large amounts of storage required, the optimization cost in terms of memory and time required will be too large, and hence the benefits of the more optimal query plan generated will be canceled out. Hence we see no real benefit in using a 1-Phase query Optimizer as opposed to a 2-Phase query Optimizer. The results that we show after running a few tests also show that using a 1PO may not be beneficial, and instead a 2PO will be simpler and faster.

# 6 Conclusions and Future Work:

Thus we have implemented a 1-Phase Query Optimizer for the SimputerDB by taking into consideration its memory limitations and integrated it with the existing code. Heuristics were redefined for better performance, and Dynamic programming was used to reduce time complexity. The optimizer was designed to work with select, project and joins. It generates left deep tree plans in which all operations are pipelined, and assigns best scheme for each operator.
However, results show that the 1PO does not give any significant performance improvements over the 2PO, but adds to the time and space complexity of the optimizer - a significant disadvantage for the resource constrained SimputerDB. Hence we see no real benefit in using a 1-Phase query Optimizer as opposed to a 2-Phase query Optimizer.

However a major limitation of the implementation was that the parser is unable to define the primary keys of the relations ( which are currently hard-coded into the system ), and the executer does not take the memory division generated by the optimizer into account while executing the queries. These would be the proposed improvements that could be undertaken in the future to tune the performance of the query processing, on the whole, irrespective of the Optimizer used.

# 7 References

Hulgeri, Seshadri, Sudarshan : Memory Cognizant Query Optimization
Rajkumar Sen, Krithi Ramamritham : Efficient Data management on Lightweight Computing Devices.
Rajkumar Sen : An Open Source DBMS for Handheld devices.
Yannis E. Ioannidis : Query Optimization.
Nicolas Anciaux, Phileppe Pucheral : Memory requirements for Query Execution in Highly Constrained Devices
Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34