# A Physical Design Tool For Relational Databases

by

Karel A. Youssefi

Memorandum No. UCB/ERL M83/9

15 February 1983

ELECTRONICS RESEARCH LABORATORY

This work was supported by the National Science Foundation under grant ECS 8007683.

# A Physical Design Tool For Relational Databases

by

Karel A. Youssefi

Department of Electrical Engineering and Computer Science

University of California

Berkeley, CA 94720

## 1. INTRODUCTION

This document describes a physical design tool to be used for relational database systems. It was implemented for the INGRES database management system [1] at UCB. This tool uses the basic concepts presented in the access path model [2] to determine, for a given set of design queries, what is an optimal set of physical storage structures for the relations in the database.

In the remaining sections, we will present a sample session using this physical design tool and a detailed description of the implementation. Also included in Section 3 is a discussion of the access path model and the simplifications which we made to it. Finally, in Section 5 is a description of features we think could be added to this tool and factors which should be studied further to determine their effect on the design process.

## 2. SAMPLE SESSION

(Please refer to Appendix A for a description of the database used here.)

```
Have you created your database and entered the data?
y or n? y


What is the name of your database?
name? kareltest

Have you entered your query set?
y or n? n

Please senter your queries.

Range of p is patient
Range of w is ward
Range of h is hospital
Range of u is uses
Range of l is lab
Range of d is doctor
Range of a is ass-to
Range of s is staff

1:  retrieve (l.lname)
    where p.pname = "Smith" and p.wd_no = w.wd_no and
          w.hosp_no = h.hosp_no and h.hosp_no = u.hosp_no and
          u.lab_no = l.lab_no

2:  retrieve (l.lname)
    where d.dname = "Jones" and d.dr_no = s.dr_no and
          s.hosp_no = h.hosp_no and h.hosp_no = u.hosp_no and
          u.lab_no = l.lab_no

3:  retrieve (h.hname)
    where p.pname = "Smith" and p.pat_no = a.pat_no and
          a.dr_no = d.dr_no and d.dr_no = s.dr_no and
          s.hosp_no = h.hosp_no

4:  retrieve (p.pname)
    where h.hname = "Merritt" and h.hosp_no = s.hosp_no and
          s.dr_no = d.dr_no and d.dr_no = a.dr_no and
          a.pat_no = p.pat_no

processing query  1 ....
processing query  2 ....
processing query  3 ....
processing query  4 ....
analyzing access path usage ...
assigning storage structures .....
relation  patient      was never accessed thru domain  wd_no
```

```
relation  uses          was never accessed thru domain  lab_no
relation  ward          was never accessed thru domain  hosp_no
 COST OF QUERY   1:   8.100000   (optimal cost:   7.200000)
 COST OF QUERY   2:   10.670000   (optimal cost:   9.770000)
 COST OF QUERY   3:   9.780001   (optimal cost:   7.395000)
 COST OF QUERY   4:   496.500000   (optimal cost:   487.100006)
Which query (if any) has an unacceptable relative cost?
query number? 3
 COST OF QUERY   1:   7.200000   (optimal cost:   7.200000)
 COST OF QUERY   2:   10.670000   (optimal cost:   9.770000)
 COST OF QUERY   3:   8.880001   (optimal cost:   7.395000)
 COST OF QUERY   4:   946.500000   (optimal cost:   487.100006)
Do you want the physical design?
y or n? y
```

physicaldes relation

| relation | attribute | structure |
|---|---|---|
| ass_to | dr_no | HASHED |
| ass_to | pat_no | SECONDARY INDEX |
| doctor | dname | SECONDARY INDEX |
| doctor | dr_no | HASHED |
| hospital | hname | SECONDARY INDEX |
| hospital | hosp_no | HASHED |
| lab | lab_no | HASHED |
| patient | pat_no | SECONDARY INDEX |
| patient | pname | HASHED |
| staff | dr_no | HASHED |
| uses | hosp_no | HASHED |
| ward | wd_no | HASHED |

```
query number? 3
 COST OF QUERY   1:   7.200000   (optimal cost:   7.200000)
 COST OF QUERY   2:   10.670000   (optimal cost:   9.770000)
 COST OF QUERY   3:   7.395000   (optimal cost:   7.395000)
 COST OF QUERY   4:   1392.000000   (optimal cost:   487.100006)
Do you want the physical design?
y or n? y
```

physicaldes relation

| relation | attribute | structure |
|---|---|---|
| ass_to | dr_no | SECONDARY INDEX |
| ass_to | pat_no | HASHED |
| doctor | dname | SECONDARY INDEX |
| doctor | dr_no | HASHED |
| hospital | hname | SECONDARY INDEX |
| hospital | hosp_no | HASHED |
| lab | lab_no | HASHED |
| patient | pat_no | SECONDARY INDEX |
| patient | pname | HASHED |

| staff | dr_no | HASHED |
|---|---|---|
| uses | hosp_no | HASHED |
| ward | wd_no | HASHED |

```
query number? 4
 COST OF QUERY   1:   8.100000   (optimal cost:   7.200000)
 COST OF QUERY   2:   10.670000   (optimal cost:   9.770000)
 COST OF QUERY   3:   8.295001   (optimal cost:   7.395000)
 COST OF QUERY   4:   942.000000   (optimal cost:   487.100006)
Do you want the physical design?
y or n? y
```

physicaldes relation

| relation | attribute | structure |
|---|---|---|
| ass_to | dr_no | SECONDARY INDEX |
| ass_to | pat_no | HASHED |
| doctor | dname | SECONDARY INDEX |
| doctor | dr_no | HASHED |
| hospital | hname | SECONDARY INDEX |
| hospital | hosp_no | HASHED |
| lab | lab_no | HASHED |
| patient | pat_no | HASHED |
| patient | pname | SECONDARY INDEX |
| staff | dr_no | HASHED |
| uses | hosp_no | HASHED |
| ward | wd_no | HASHED |

```
query number? 4
 COST OF QUERY   1:   8.100000   (optimal cost:   7.200000)
 COST OF QUERY   2:   10.670000   (optimal cost:   9.770000)
 COST OF QUERY   3:   9.780001   (optimal cost:   7.395000)
 COST OF QUERY   4:   496.500000   (optimal cost:   487.100006)
Do you want the physical design?
y or n? y
```

physicaldes relation

| relation | attribute | structure |
|---|---|---|
| ass_to | dr_no | HASHED |
| ass_to | pat_no | SECONDARY INDEX |
| doctor | dname | SECONDARY INDEX |
| doctor | dr_no | HASHED |
| hospital | hname | SECONDARY INDEX |
| hospital | hosp_no | HASHED |
| lab | lab_no | HASHED |
| patient | pat_no | HASHED |
| patient | pname | SECONDARY INDEX |
| staff | dr_no | HASHED |
| uses | hosp_no | HASHED |
| ward | wd_no | HASHED |

```
query number? 4
 COST OF QUERY   1:   9.100000   (optimal cost:   7.200000)
 COST OF QUERY   2:   12.170000   (optimal cost:   9.770000)
 COST OF QUERY   3:   12.030001   (optimal cost:   7.395000)
 COST OF QUERY   4:   495.899994   (optimal cost:   487.100006)
Do you want the physical design?
y or n? y
```

physicaldes relation

| relation | attribute | structure |
|---|---|---|
| ass_to | dr_no | HASHED |
| ass_to | pat_no | SECONDARY INDEX |
| doctor | dname | SECONDARY INDEX |
| doctor | dr_no | HASHED |
| hospital | hname | HASHED |
| hospital | hosp_no | SECONDARY INDEX |
| lab | lab_no | HASHED |
| patient | pat_no | HASHED |
| patient | pname | SECONDARY INDEX |
| staff | dr_no | HASHED |
| uses | hosp_no | HASHED |
| ward | wd_no | HASHED |

```
query number? 4
 COST OF QUERY   1:   9.100000   (optimal cost:   7.200000)
 COST OF QUERY   2:   13.650001   (optimal cost:   9.770000)
 COST OF QUERY   3:   14.250000   (optimal cost:   7.395000)
 COST OF QUERY   4:   487.100006   (optimal cost:   487.100006)
Do you want the physical design?
y or n? y
```

physicaldes relation

| relation | attribute | structure |
|---|---|---|
| ass_to | dr_no | HASHED |
| ass_to | pat_no | SECONDARY INDEX |
| doctor | dname | SECONDARY INDEX |
| doctor | dr_no | HASHED |
| hospital | hname | HASHED |
| hospital | hosp_no | SECONDARY INDEX |
| lab | lab_no | HASHED |
| patient | pat_no | HASHED |
| patient | pname | SECONDARY INDEX |
| staff | dr_no | SECONDARY INDEX |
| staff | hosp_no | HASHED |
| uses | hosp_no | HASHED |
| ward | wd_no | HASHED |

## 3. MODEL

We started originally by examining the access path model proposed by Katz and Wong [2]. This model consists of objects and functions between pairs of set of objects. Basically, relations and the attributes of a relation are both object sets. Functions define relationships between object sets. These objects and functions are used to describe the schema for the database. Consider a database which stores information about employees, the departments they work in, and the jobs they are assigned. An obvious graphical representation of this schema is shown in Figure 1.

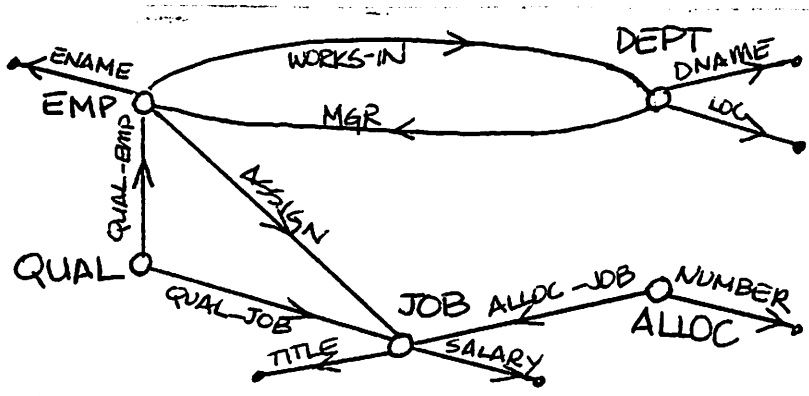


Figure 1. Graphical representation of schema.

where the arrows represent the direction of functionality. Notice that most often, the functions between relations are actually supported by including a foreign key attribute in the source relation. For example, the function

works_in: EMP -> DEPT

usually means that there is an attributed dept-no which is a primary key in DEPT and an associated attribute dept-no is included in EMP as a foreign key. Note that the attributes

used to represent these functions are not named explicitly in the representation, since they are implied by the function.

This logical description of the database expressed in terms of the functional data model is called an integrity schema. In order to facilitate physical design, this integrity schema is augmented by assigning certain access characteristics to each function, yielding what is called an access path schema. The access characteristics to be assigned to each function f:A->B are:

1) evaluated:

   given a in A, f(a) can be found witnout an exhaustive scan of B, i.e., the cost to access f(a) is less tnan the cost to access every element of B.

2) indexed:

   given b in B, $f^{-1}(b)$ can be found without an exhaustive scan of A.

3) clustered:

   the elements of $f^{-1}(b)$ are in close proximity, i.e., the cost to access the elements in the inverse is less than the cost to access an arbitrary subset of the same cardinality.

4) well-placed:

   a and f(a) are stored in close proximity, i.e., the cost to access both is less than the cost to access

them separately.

When trying to associate these characteristics with implementable storage structures, certain constraints on assignment are implied which make the task of maximal labelling more difficult.

Since we were attempting to implement this design aid for the INGRES database management system, we noted that the concept of well-placed is not supported in that system for a function between two relations. To support well-placed, a storage structure which interleaves tuples of more than one relation on a single page is required. If the access characteristic well-placed is eliminated from the model, several simplifications are possible. First, the majority of the constraints on assignment of the remaining characteristics are removed. Second, and most important, the remaining access characteristics can be associated directly with the objects themselves, rather than the functions. Consider the function

works_in: EMP -> DEPT

If works_in is an evaluated function, this can be supported by DEPT being keyed on the attribute dept_no. If works_in is indexed, this can be supported by having a secondary index for the EMP relation on the attribute dept_no. If works_in is clustered, this means that the EMP relation is keyed on the attribute dept_no. Note that each

of these characteristics is supported by a structure affecting only one of the relations; no assumption or implication is made effecting the structure of the other relation involved in the function. Thus, if we limit ourselves to the three access characteristics of evaluated, indexed and clustered, we may make assignments to the relations themselves.

## 4. PROCESSING ALGORITHM

The program to support this physical design tool is an EQUEL program. As such, the data used by it is stored in relations. These relations are created and filled by the program in the user's database and remain there for the life of the database. As there are only four such relations needed and the size of each is very small, there should be no degradation of performance felt by the user in the normal usage of his database.

It should be mentioned first that only portions of this design process were actually implemented. The pieces implemented are the portions pertaining to the actual design decision-making process. The original entry of the queries and some initial processing were not implemented but a description of what should be done and any assumptions about this portion and how it affects later processing will be included.

When the user initiates this program, the first tning that happens is that the name of the user's database is

requested. The user must have created his database prior to use of this tool, and he must have entered his data into the database. Hopefully, he has run a statistics-gathering program over his data also. This gathering of statistics could be an automatic feature included with copy or it could be a stand-alone program the user initiates. (The UCB version of INGRES currently has no statistic-gathering feature, this information must be entered by hand.) Whichever manner is used, statistical information (minimally, the count of unique values in an attribute) must be available for each attribute, relation pair referenced in the set of design queries (attributes appearing only in the target list may be eliminated).

Other information required about the database structure must currently be entered by hand. This is information concerning the semantics of the data and would be available if the logical design package had been used (see 3). Basically what is required is how the relations in the database are semantically linked with each other. This would be stored in the LINK relation.

> LINK relation: this relation contains a description of the logical links that exist between relations in the database. The information in this relation is only changed when the schema description is changed affecting the logical links. [Note: this relation is not used or created by the current working version of phys-

desdb.] The structure of the LINK relation is

```
rel1:        c12
dom1:        c12
rel2:        c12
dom2:        c12
```

Each tuple thus represents a link between two relations specifying the domains involved in the link. These would be a primary key attribute in one relation and its associated foreign key attribute in another relation. The direction of the function is implied by the ordering, that is

f: relation 1 -> relation 2

Associated with the LINK relation is another relation which gathers all the data dependent information about the attributes involved in the links.

REL relation: This relation contains a detailed description of every relation, attribute pair which is involved in a logical link (i.e. is in LINK relation). The structure of the REL relation is:

```
relname:     c12

domname:     c12

key:         i1      - code representing key
                       information for domain
                             1:  primary key
                             2:  foreign key
                             3:  value domain

relcard:     i4      - cardinality of relation relname
                       (number of tuples)
```

| | | |
|---|---|---|
| domcard: | i4 | - cardinality of attribute domname<br>if primary key: relcard=domcard<br>if foreign key:<br>domcard=domcard of attributed<br>which is associated<br>primary key<br>if value domain:<br>domcard=number of values<br>in attribute domname |
| npages: | i4 | - number of pages occupied by relation relname |
| cnt: | f4 | - frequency count of access to this relation through this attribute for set of design queries. |
| strstruct: | c1 | - storage structure assigned for this relation, attribute<br>H - hashed (clustered)<br>I - secondary index (non-clustered)<br>N - heap (unstructured) |
| cost: | f4 | - the cost per unit value access for this relation, attribute given the assigned storage structure. |
| optcost: | f4 | - the optimum cost per unit value access for this relation, attribute. |

The last four attributes of the REL relation are calculated and inserted during the physical design process. Notice that optcost could be computed at this time and entered, because its value does not depend on any decision. The current implementation, however, computes optcost at the same time as cost.

So, at this point we have the logical structure of the database represented and data dependent information about the attributes. The set of design queries should now be input and processed. These queries will be QUEl queries and all parsing and name checking should be performed just as it

is in INGRES. The original design called for a simplified version of the INGRES parser to be included at this point outputting a parse tree. This parse tree will then be processed to produce the QUERY and ONEVAR relations.

QUERY relation: this relation contains a description of the queries involved in the design set. There is one entry per join term in each query plus one entry for the starting point per query. The structure of the QUERY relation is:

| | | |
|---|---|---|
| queryno: | i1 | |
| squeryno: | i1 | - subquery number, only ≠ 1 when query is other than path query |
| entryno: | i1 | - only necessary to allow ordering in this relation to be unimportant |
| relname: | c12 | - relation name |
| domname: | c12 | - attribute name, note that this relname, domname pair must appear in the REL relation. |
| ovflag: | i1 | - flag for one-var restrictions on this relname. if = 1, must check ONEVAR relation for additional selectivity constraints. |
| runcount: | f4 | - count of number of tuples to be passed into this node of query. For first entry of query, this = 1; for first entry of each subsequent subquery, = 0 (see discussion). |
| compcost: | f4 | - the cost associated with this node of the query. = Runcount * cost, where cost is from REL relation for relname, domname pair. |

ONEVAR relation: this relation contains information about all one-variable (one-relation) restrictive

clauses which appear in the query set: all information contained is static for the life of the query set. The structure of the relation is:

| | | |
|---|---|---|
| queryno: | i1 | - the query number of QUERY that this clause is associated with. |
| relname: | c12 | - relation name (must match a relname in QUERY for queryno) |
| domname: | c12 | - the attribute involved in the restriction. |
| relcard: | i4 | - cardinality of relname |
| domcard: | i4 | - cardinality of domname (number of distinct values) |

It's true that this information can be replicated (i.e., same restrictive clause in several queries, relcard same as in REL relation), but the savings in simplicity of design and query complexity are believed warranted. In this way, to get the selectivity of the restriction, only a single-relation query on ONEVAR is needed, otherwise a query involving both ONEVAR and REL is needed. However, this replication of values could be changed if deemed necessary. If histogram information is used instead of only cardinality, it may be necessary to add the constant value into this relation in some manner to give more accurate predictions.

The processing can be done as follows for each query. Before any other processing is done, the query should be checked for cycles, since we cannot handle cyclic queries. This can be done by keeping a map of the relations involved and marking this map appropriately. Now, all one-variable

restrictive clauses are removed from the parse tree and inserted into the ONEVAR relation with the appropriate query number (and possibly into REL as well). Then, having considered the selectivity of these clauses, a starting relation is chosen, based on size (number of tuples).

The relation and its associated attribute (either the 1-var clause attribute or the joining-clause attribute) is the first entry for this query in QUERY. Then, using the information in this LINK relation, subsequent join terms are selected from the parse tree to form a naturally linked path through the data representing the query. Specifically, the next entry in QUERY will be the relation, attribute pair which appears in a joining clause in the query with the first entry and which also has a corresponding entry in the LINK relation. If join clauses appear in the query which do not have corresponding entries in LINK, either the clause will be eliminated from consideration or the entire query will be disqualified.

As each entry is made in QUERY, its appropriate runcount value can be computed. This value is the count of the number of tuples which will be input to this node from the previous portion of the query. Runcount for the first entry of each query is 1. Then by using the selectivity information, the number of tuples per value which will be passed out is calculated (relcard/domcard). This times the previous Runcount (which is number of values) give us runcount for

the next entry, etc.

If the query is a tree query, that is it's processing path is not a simple path, we treat it as a set of subqueries. The first subquery is a simple path following one branch of the tree to completion. Each subsequent subquery starts from the root of a subtree and follows that path to completion. (Subqueries can be nested). The first entry in each subquery after the first is essentially a "dummy" node representing the root of that subtree. This is a dummy entry since runcount is zero for it because runcount has already been computed for this entry and we don't want to count it more than once in our computations.

This processing is done for each query in the design set. At the end of this portion of the program, we have an internal representation of the processing path for each query and a running count of the number of tuples involved at each node of each query. The current implementation essentially starts at this point and continues. The REL relation, ONEVAR relation and certain columns of the QUERY relation must be loaded by hand into the database. It is not important that the data relations themselves exist or not in this database as they are never referenced. The first six elements of the QUERY relation for each node must be entered. However, the code exists to compute runcount over the QUERY relation.

Now, for each entry in the REL relation, the QUERY relation is scanned and runcount is gathered wherever that relation, attribute pair occurs. These runcounts are summed by relation, attribute pair to give the usage count for that pair (cnt in REL relation). Then, REL is again examined, this time by relation, to determine the attribute with the highest usage count. This attribute will be the clustering domain for this relation and thus determine the storage structure of the relation itself. In our implementation, it is assigned a structure hashed on that attribute. All other attributes of the relation which appear in the REL relation and have a usage count greater than some minimum (currently zero), will have a secondary index created for them in the physical design. Associated with each such structure is a cost to access the set of tuples with a given attribute value. This cost is computed by using a formula designed for each structure and dependent on the data characteristics of the attribute involved. So this cost is computed and entered as cost in the REL relation along with a code representing the actual storage structure assigned for that attribute (strstruct).

Once this has been done for all relations, we have our proposed physical design. We now want to compute an approximate cost for processing each of the design queries using this design. We do this by summing, for each query, the product of runcount and the cost entry in REL for that rela-

tion, attribute pair. At the same time, we also sum, by query, the product of runcount and optcost in REL to give us the optimal processing cost which can be used as a base by the user. These two costs are displayed for the user.

At this point, the user may request to see the physical design associated with these costs. This is just a matter of printing selected columns of the REL relation substituting an appropriate message for the code strstruct.

Now, the user may indicate a particular query that should be further optimized. We get the query number for this query and then examine the QUERY relation. We look at all the entries for that query and find the entry for which compcost is maximum. If the storage structure associated with the relation, attribute in that entry is hash, we cannot decrease the cost, so we look for the next largest, etc. until we find a relation, attribute pair which is not hash structured. We then change its structure to hash, recomputing cost and replacing the appropriate values in REL. We must then change the attribute in this relation which previously was hash to a secondary index structure. So we search the REL relation for that tuple and make the appropriate updates after recomputing its cost. Then, using the new storage structure costs, the cost of processing for all the queries is recomputed and output for the user, again with the optimal costs.

This process continues until the user is satisfied with the results of the design process and the relative costs of processing each query are acceptable. At this time, the user will indicate that the design phase is complete and the program will write a series of INGRES statements into a file with the name provided by the user. These statements are the appropriate MODIFY and CREATE INDEX statements to change the structure of the database into that determined by the final design. This final file write is not currently implemented.

There are certain other features which were included in the original design but were not implemented and may require minor modifications. First, the set of design queries may be input with associated weighting factors. This would require that these weights be included in the runcount for the queries. Secondly, it was desired that this package could also be used for tuning purposes. This would require a second mode of processing which bypasses the original input of queries, although modifications could be made to add to or delete from the original design set. This would also require possibly gathering new, more current statistics.

## 5. Areas of Further Study

In this paper, we have presented a new approach to a physical database design tool which is independent of the actual query processing strategies. This approach reduces

the amount of computation required and, we feel, provides more flexibility to the model which is more conducive to easy interaction.

Only one set of design queries has been used to test the design tool. The results seem to indicate that the design recommended by the tool does result in actual lower processing costs. Although the cost estimates for processing were not exact (the error was less than an order of magnitude) the relative ordering of processing costs was correct when compared to actual processing costs (run on the RTI version of INGRES). These results are preliminary, however, and much more extensive testing is needed.

The results of these tests and an examination of the actual processing strategies used did however point to several areas where further study is needed to determine how these factors should be incorporated into the model.

Currently, the target list of a query is not considered at any point in the analysis. However, membership in the target list has an effect on several factors. First, if a relation is not included in the target list, any appearance in the qualification requires only a check for existence, thus all tuples satisfying need not be retrieved. Secondly, the target list membership determines the number of pages in intermediate result relations. It can also be used to determine the potential benefits of an intermediate projection.

In the current implementation which is for the INGRES system, the cost functions used to determine the cost of access for hash and isam structures indicate that hash is always better because the access will be on a per value basis. But, when doing a merge-sort join, isam would definitely be preferable because it provides sorted order if the index and overflow pages are ignored. Some method of deciding between the two structures for a clustered domain based on factors other than the cost function above is needed.

A possible approach would be to do the analysis up to the point when clustering or non-clustering domains are selected and then do some sort of additional analysis on a pair-wise basis. This analysis would be based on the sizes of the relations at that point in the processing path, both in number of tuples and number of pages. It appears that a certain relationship between the number of tuples in A and number of pages in B indicates that a lookup-scan type operation (hash preferable) would be better than a merge-scan type (isam preferable). More work is needed in this area.

Update queries are not accepted in the current model because of the added dimension of complexity involved. However, the model should definitely be extended to include them. One possible way would be to consider an update to have essentially two separate components. The first component would be the equivalent retrieval query, for which

the processing is obvious. The second component would be the updating of all secondary structures effected by the update. This would require new cost functions for updating secondary indices and a method of determining which structures were effected. An obvious feature which would be desirable for this approach is a two-component cost figure and an option to minimize one or the other (or both), thus being able to minimize the overhead of an update by eliminating the lest useful of the secondary structures.

Another obvious extension is to compute an estimated size for the database (in disk pages) given the proposed physical design and being able to minimize or limit this figure.

One type of query which we do not allow is a cyclic query, such as the parts-explosion query. No way of modelling this type of query was found which did not depend on comparing alternative processing paths and since one of our goals with this model was to avoid the comparison process, we have not found an acceptable way of analyzing cyclic queries.

Currently, all attributes which appear in a query are not considered as candidate clustering domains. Some attributes which appear only in single-relation restrictive clauses are considered for their effect on the selectivity but not as clustering or even non-clustering domains. These attributes should be considered. However, exactly how their

associated usage count should be determined is not clear. Logically speaking, access is only made through that attribute once for the entire query, since hopefully the processing algorithm is smart enough to do the restriction only once and save the result, if necessary. But, a usage count of 1 would mean that this attribute rarely would be chosen as a clustering domain. Basically, the effect of these "restrictive" attributes on physical design and how they should be considered is not fully understood.

A feature which was suggested and could be very useful is some sort of graphical interface to display the resulting costs of processing as a curve (graph). The user would be able to interact through this interface by indicating which portions of the curve to fluctuate and which should remain fixed. The idea behind this is that the user could indicate which query to minimize while keeping certain other queries unchanged.

One last area of work is in determining equivalence of queries (using the transitive closure property) and which of the possible paths should be chosen. The transitive closure property and its application to the qualification of a query has been used often [4]. Basically, what this implies is that we have a cycle involving three nodes in our integrity schema.

```
i.e., retireve (t1)
        where A.a = B.a and B.a = C.a
           => A.a = C.a
```

integrity schema graph

(By using the integrity schema we can also detect certain cases where the transitive closure property can be applied which are not obvious, i.e., attribute names are different.) The main problem is in deciding which direction this cycle should be traversed. Since it only involves three nodes, it is possible to consider and compare the alternatives. But it may be possible to use heuristics to make a better determination. Possible factors which should be considered are the direction of functionality (if one goes in the non-functional direction, the one-to-many direction, there is a good chance of fan-out) and whether this cycle appears in the middle of a path, which is the connecting node to the remainder of the path.

# APPENDIX A

## Sample Database

Relation: patient
  attributes: pat_no
              pname
              wd_no
                .
                .
                .
  tuples: 10000
  pages: 1000

Relation: ward
  attributes: wd_no
              hosp_no
                .
                .
                .
  tuples: 500
  pages: 100

Relation: ward
  attributes: hosp_no
              hname
                .
                .
                .
  tuples: 50
  pages: 20

Relation: doctor
  attributes: dr_no
              dname
                .
                .
                .
  tuples: 500
  pages: 50

Relation: ass_to
  attributes: pat_no
              dr_no
                .
                .

```
                    .

  tuples:           15000
  pages:              150

Relation:    staff
  attributes:       dr_no
                    hosp_no
                      .
                      .
                      .

  tuples:           750
  pages:             10

Relation:    lab
  attributes:       lab_no
                      .
                      .
                      .

  tuples:           100
  pages:             20

Relation:    uses
  attributes:       hosp_no
                    lab_no
                      .
                      .
                      .

  tuples:           150
  pages:              5
```

## REFERENCES


[1] Stonebraker, M., et al., "The Design and Implementation of INGRES", TODS 1,3, Sept. 1976.

[2] Katz, R. H. and E. Wong, "An Access Path Model for Physical Database Design", Proceedings of SIGMOD, 1980.

[3] DuMont, Lorna, E., "Designing a Database for INGRES" Memorandum No. UCB/ERL M82/23, February 1, 1982.

[4] Kooi, R. P., "The Optimization of Queries in Relational Databases", Ph.D. Thesis, Dept. of Computer and Information Science, Case Western Reserve Univ., Sept. 1980.