HEURISTIC SEARCH IN DATA BASE SYSTEMS

by

R-M. Kung, E. Hanson, Y. Ioannidis, T. Sellis,
L. Shapiro and M. Stonebraker

Memorandum No. UCB/ERL M84/58

23 July 1984

Heuristic Search in Data Base Systems

by

Ru-Mei Kung, Eric Hanson, Yannis Ioannidis,
Timos Sellis, Leonard Shapiro, and
Michael Stonebraker

Memorandum No. UCB/ERL M84/58

23 July 1984

ELECTRONICS RESEARCH LABORATORY

mentation chosen and the actual data on which the algorithm is run. Hence, one of the tasks of an expert system implementor is to discover which class of algorithms appears practical for his problem and then to code a variation of the algorithm which performs effectively on his data. In addition, if the expert system in question must deal with dissimilar data sets, then the implementor must either code a collection of algorithms and have a meta expert system to perform algorithm selection or accept poor performance on some data sets.

The basic objective of our research is to move the search portion of an expert system into a data base management system. On large data sets, a DBMS is a practical necessity; hence, expert systems for such data sets must act as application programs to a data base manager. Consequently, one has the opportunity of selectively moving function from such application programs into the DBMS. Common function can thereby be written once within the DBMS and not once per application as is common today. Moreover, performance improvements are often realizable from such function migration.

There are two components to heuristic search in an expert system, namely the selection of which of several possible algorithms to use, and the execution of the chosen algorithm. In this paper we will show that with relatively minor extensions to a classical query language such as QUEL [STON76] it is possible to integrate the execution of many algorithms into a DBMS. In addition, we show how selection of an efficient algorithm can be effectively performed within the DBMS.

We have focused on a particular shortest path problem as the context for our study. We have coded several of the popular heuristic algorithms for this problem and measured their performance in a DBMS context. Our results are quite different from those predicted for main memory data [PEAR84].

The remainder of this paper is organized as follows. In Section 2 we explain our example shortest path problem in detail. Then in Section 3, we present an extended version of the query language QUEL [STON76] which will allow us in Section 4 to express a variety of common algorithms for this problem. Section 5 then indicates experimental results on a real data base. We

turn in Section 6 to automatic selection of an efficient algorithm by the DBMS and suggest several appealing strategies. Lastly, Section 7 contains our conclusions from this study.

2. THE OVERLAND SEARCH PROBLEM

The task we will focus on is to move a vehicle from a given point (say San Francisco airport) to a second point (say the Computer Science building at the University of California in Berkeley) in minimum expected cost. The applicable cost function might be the minimum expected time of travel or the minimum expected gasoline consumption.

There can be many constraints on the feasible path. For example, certain vehicles (e.g. Hertz rental cars) are restricted to travel on roads while other (e.g. tanks, off road vehicles) can choose an overland route. Even overland vehicles are constrained in route selection because they cannot pass through rivers greater than a certain depth or traverse terrain which exceeds a specific pitch. Additionally, the cost of certain paths can vary with time of day or day of week; for example the cost of traveling on a highway changes during rush hour.

The specific search problem is to find a minimum cost route from a point START to a point FINISH that satisfies a collection of constraints of the above form. Moreover, it is a simple bookkeeping problem to keep track of the total route; hence we will content ourselves with simply finding the cost of the best route.

We assume that a topographical map of the area in question is available. It is a separate problem to process the raw map data into a form usable by a heuristic search program. For the moment, we assume that the map has been processed into a relation of the form

FEASIBLE (source, dest, cost)

where the cost attribute indicates how difficult it is to travel from point "source" to point "dest". We also assume the availability of point as a fundamental DBMS data type in this discussion. It can be realized as a built-in data type or added through the mechanism proposed in [STON83].

Notice that if a map is discretized at 100 foot intervals, this will result in about 2500 points per square mile. If each point is accessible from its eight nearest neighbors, we require 20,000

tuples in FEASIBLE per square mile. A 50 mile by 50 mile map would then consume about 600 Mbytes if each tuple requires 12 bytes. Clever coding of the data might reduce this amount; however, it should be clearly noted that FEASIBLE is a very large relation. Hence, it is not reasonable to assume that it is a main memory data structure as is commonly done by algorithm designers [PEAR84].

We now turn to describing an extended version of QUEL which will allow us to express a collection of solutions to this problem.

3. DATA BASE SOLUTIONS

It is clear that for all paths between pairs of nodes in

FEASIBLE (source, dest, cost)

the transitive closure of FEASIBLE contains and edge. In fact, assuming that FEASIBLE is acyclic, the following algorithm will generate the cost of all possible paths from any START to any FINISH:

```
retrieve into TEMP (FEASIBLE.all)

repeat
   append to TEMP ( TEMP.source, FEASIBLE.dest,
                cost = FEASIBLE.cost + TEMP.cost)
   where TEMP.dest = FEASIBLE.source
until TEMP does not grow
```

The cost of the best path between two points START and FINISH is the minimum "cost" value in TEMP with source equal to START and dest equal to FINISH. The goal of any heuristic search algorithm is to do better than the above computation.

It should be noted that the computation of transitive closure cannot be expressed in QUEL (or in most of the relational query languages that we are aware of). Only QBE [ZLOO75] and ORACLE [ORAC84] have limited capabilities to express such recursive calculations. Hence, our first task must be to enlarge QUEL with the appropriate function. We follow the extensions of [GUTT84] in including a * operator in QUEL.

Using the * operator, the above calculation becomes:

```
retrieve into TEMP (FEASIBLE.all)
append* to TEMP ( TEMP.source, FEASIBLE.dest,
            cost = FEASIBLE.cost + TEMP.cost)
where TEMP.dest = FEASIBLE.source
```

Basically, the semantics of APPEND* are to repeat the command forever. In practice, the execution of the command can cease when an iteration produces no additional tuples in TEMP. Similar semantics hold for REPLACE*, DELETE* and RETRIEVE* into a result relation. For example, consider the following alternate algorithm that computes the cost of the best path between all pairs of points (even for cyclic graphs):

```
range of F1 is FEASIBLE
range of F2 is FEASIBLE
retrieve into TEMP (F1.source, F2.dest, cost = INFINITE) where
      F1.source != F2.dest
replace TEMP (cost = FEASIBLE.cost) where
      TEMP.source = FEASIBLE.source and
      TEMP.dest = FEASIBLE.dest
range of T is TEMP
replace* TEMP( cost = T.cost + FEASIBLE.cost) where
      T.cost + FEASIBLE.cost < TEMP.cost and
      T.dest = FEASIBLE.source and
      FEASIBLE.dest = TEMP.dest
```

The semantics of REPLACE* are to execute the command indefinitely. In practice execution can cease when one of two things happens:

1) the qualification becomes false. In this case one need not continue to execute the command because it will not have any effect at any time in the future.
2) the command fails to update a tuple. Again, continuing to run the command will have no effect.

Two comments should be made at this time. First, in the case that there is more than one adjacent point that can be used to lower the cost of a given tuple in TEMP, the REPLACE* command will attempt to replace a single data item by two or more calculations. This is an example of a non-functional update [STON76] which is allowed by some versions of INGRES and disallowed by others. Non-functional updates will be discussed further in section 4. Second, notice that it is possible to define REPLACE* as non-terminating. For example, one could simply make the command inactive until some other update caused the data base to change so that the command could be rerun and have an effect. For example the following command will run forever

and update the salary of Mike to be equal to that of Leonard whenever Leonard's salary is changed:

```
range of E is EMPLOYEE
replace* EMPLOYEE (salary = E.salary) where
        EMPLOYEE.name = "Mike" and
        E.name = "Leonard"
```

In this paper we assume that * commands terminate. The efficient implementation of the non-terminating version of these commands (which are in effect triggers [ESWA76]) is described separately [STON84].

Our second necessary construct is the notion of storing collections of QUEL commands in the data base and then being able to execute them. For example, suppose both routines for constructing TEMP are stored in the relation, COMMANDS:

```
COMMANDS (id, QUEL-desc)
```

If one has id "mostly-replace" and the other "mostly-append", then we could execute the "mostly-replace" routine as follows:

```
range of C is COMMANDS
execute (C.QUEL-desc) where C.id = "mostly-replace"
```

Suppose we change the "mostly-replace" routine so that the first retrieve and replace commands are stored separately in COMMANDS as a routine called "warm-up". Then, the REPLACE* is changed into a simple REPLACE and stored in "inner-loop". The following commands are an equivalent specification of the "mostly-replace" routine:

```
execute (C.QUEL-desc) where C.id = "warm-up"
execute* (C.QUEL-desc) where C.id = "inner-loop"
```

In the next section we turn to expressing a collection of popular search algorithms in this extended language, QUEL*.

4. SHORTEST PATH PROBLEMS IN QUEL*

We have coded and simulated two kinds of algorithms, Breadth-First and Best-First. By Best-First we mean that only the node(s) with the least accumulated cost are expanded in each

iteration. This section contains QUEL* code for these algorithms and an extension of the algorithms which uses a hierarchical decomposition of the map space.

In addition to the FEASIBLE relation, each algorithm will also use a relation describing the state of the system as the algorithm progresses:

STATES (dest, cost)

where cost is the cost of the cheapest path, found by the algorithm so far, from START to the node labeled dest. We use the following range variables in the QUEL* code:

```
range of s,t is STATES
range of f is FEASIBLE
```

Our first coding of the algorithms follows the classic approach, in which only visited nodes are kept in the state space.

BREADTH FIRST - VISITED NODES

```
/* Initially STATES consists only of the starting point */
append to STATES(cost = 0, dest = START)

execute*
{       /* Append the newly expanded nodes */
        append to STATES(dest = f.dest, cost = f.cost + s.cost) where
        s.dest = f.source

        /* Delete states that are dominated */
        delete s where
        s.dest = t.dest and s.cost > t.cost        /* t is a cheaper path */
                or
        s.cost > t.cost and t.dest = FINISH /* the cost of getting to s is already more
                                               than the cost of getting to FINISH */
}
```

BEST FIRST - VISITED NODES

```
/* In the best first algorithms, a status field is added to the STATES
relation, with values "open", "current and "closed". The value "open"
means the node has not been expanded, "current" means it is being
expanded in the current step, and "closed" means it has been expanded. */

retrieve into STATES (dest = START, cost = 0, status = current)

execute*
{       /* expand best node */
        append to STATES (dest = f.dest, cost = s.cost + f.cost, status = open)
        where f.source = s.dest and s.status = current

        /* Close the state that was expanded */
        replace s(status = closed) where s.status = current

        /* Delete states that are dominated */
        delete s where
        s.status = open and
        s.dest = t.dest and s.cost > t.cost

        /* Mark best nodes as current */
        replace s(status = current) where
        s.status = open and s.cost = min(t.cost where t.status = open)
        and s.dest != FINISH    /* If we are done, nothing is marked current */
}
```

We also coded these two algorithms with a state space equal to all nodes, i.e. one state for each node in the graph. This allows us to use fewer calls to the database inside the loop, which may improve performance for reasons we discuss below. While the routines which keep only visited nodes in STATES use an append followed by a delete, our next approach needs only a replace* command to accomplish most of the processing.

BREADTH FIRST - ALL NODES

```
/* Initially all nodes in STATES have infinite cost, except that START has zero cost */

retrieve into NODES (s.dest)
range of n is NODES
append to STATES(n.dest, cost = INFINITE)
replace s(cost = 0) where s.dest = START

/* In this algorithm, all the processing is done by one replace* */

replace* t (cost = s.cost + f.cost) where
        f.source = s.dest and  f.dest = t.dest  /* f is path from s to t */
                and
        t.cost > f.cost + s.cost /* this path is cheaper */
```

The code above includes a non-functional update [STON76]. In the case that the cost of a given tuple in STATES can be improved in more that one way, the system will attempt to update it multiple times. Many database systems refuse non-functional updates, including the version of INGRES used for the performance tests that follow. The stated reason is that such an update replaces a value by two or more different values and the last value processed is the one that persists. Hence, a random answer is recorded as a result of such an update. However this will not lead to incorrect results in the above code, since a better improvement will be chosen in the next iteration of the loop to replace the random improvement selected. If the system does not allow non-functional updates, additional queries must be added to the above code.

Finally, in the routine for the "best first - all nodes" algorithm that follows, we have added tuples (z,z,0) to the FEASIBLE relation for all nodes z. This results in fewer queries in the loop.

BEST FIRST - ALL NODES

```
/* In this routine a field "open" is added to the STATES
relation, with values true or false.  A state is open (i.e.
its open attribute is "true") if it has not been expanded */

retrieve to nodes (s.dest)
range of n is nodes
append to STATES(n.dest,cost = INFINITE, open = true)
replace s(cost = 0) where s.dest = START
append to FEASIBLE(n.dest,n.dest,0)

execute*
{ /* find the minimum cost node */
        retrieve(mincost = min(s.cost where s.open = true) )
        retrieve(mindest = min(s.dest where s.open = true and s.cost = mincost) )

        /* expand that node */
        replace t(open = (t.cost != mincost), cost = s.cost + f.cost) where
        s.dest = mindest /* s is being expanded */
                and
        f.source = mindest and f.dest = t.dest /* t is neighbor of s, or s = t */
                and
        (
                t.cost > f.cost + s.cost /* This route is cheaper */
                        or
                t.dest = s.dest          /* or s = t */
        )
}
```

It is possible to improve the performance of all the algorithms by using a variety of estimators to decide which nodes should be expanded, but we do not address that subject in this paper.

If START and FINISH are widely separated, and the maps are of fine granularity, then the algorithms may take prohibitively long to find a shortest path, even with a judicious use of estimators. To deal with such problems, we can embed the above algorithms in a hierarchical algorithm which uses main road maps. The main road algorithm can be expressed in QUEL, so it together with any of the above algorithms will provide a mechanism for finding shortest paths over a wide range of START and FINISH points.

In the hierarchical algorithm we assume that the database is presented with a tree of maps, each in the form of the FEASIBLE relation. The leaves of the tree are detailed maps and the higher levels include only nodes from the lowest levels which are on main roads. For example,

the leaves might be plat maps, the second level might have main road maps through a city, and the highest level could have main roads through a state. There are at least two ways to use the main road maps in the search process, a top down or a bottom up approach. In the top down approach, one first finds a main road from the plat containing START to the plat containing FINISH and then searches the individual plats for a path from the chosen main road to the START and FINISH points. The bottom up approach first finds shortest paths from START and FINISH to any main road, then searches the main road network for the shortest path between the two chosen points on the main roads. Each model has its own advantages and the choice of bottom up vs top down again depends on the data. An improvement to either algorithm would go straight to the bottom level if the points in question are close enough together. Any of these choices can be easily coded in QUEL*,

5. PERFORMANCE RESULTS

We coded the algorithms from the previous section in EQUEL [ALLM76] on the Relational Technology version of INGRES by simulating the effect of the * operator by rerunning a command until the tuple count of modified tuples was zero. We also coded the Best-First algorithm in FORTRAN using virtual memory data structures in the program address space for STATES and FEASIBLE. This program kept only visited states in its data structure for STATES. Our data set was a 36 square mile area of Santa Cruz county, California. This area is extremely mountainous with deep valleys and an extensive river system. It was digitized into three FEASIBLE relations of different granularity, representing grids which are 10, 20 and 30 nodes on each side. The START and FINISH points were at opposite corners of the map.

The CPU time of the various algorithms is indicated in Table 1. All times are in seconds.

	10 by 10	20 by 20	30 by 30
FORTRAN	24	1,010	6654
Breadth first - Visited Nodes	949	62,071	...
Best first - Visited nodes	108	964	2962
Breadth first - All nodes	84	718	1361
Best first - All nodes	126	1449	4455

Figure 1: CPU time of algorithms

The times for the FORTRAN program are growing much more rapidly than those for the best QUEL* algorithm. This supports our thesis that a database manager is a practical necessity for large maps. The reason that the FORTRAN code is not as fast as INGRES is because it does joins by scanning one relation repeatedly for every tuple of the other relation (the nested loops approach to join processing). On the other hand, the INGRES query optimizer considers a nested loops approach as well as a sort-merge tactic. On this data sort-merge is far superior. Of course, the FORTRAN program could have coded a sort-merge join and obtained considerably better performance. However, that would have made the program a great deal longer. In addition, it is not clear that sort-merge is the algorithm of choice on all maps. Hence, a dynamic query optimizer is a valuable tool for this problem.

The FORTRAN program was two pages of code, while the QUEL* program was one half page long. This supports our thesis that shortest path algorithms will be easier to write in QUEL* than in a programming language.

Within the different QUEL* algorithms, we note that the Breadth-First algorithm with all nodes in STATES outperforms all other choices. The Best-First approach (known as the A* algorithm when augmented with estimators) is known [PEAR84] to be more efficient than the Breadth-First approach when the database can fit in main memory. We speculate that Breadth-First is generally better in a database environment because it makes few calls to the set-oriented database system. Hence, it may expand a larger number of nodes than Best-First, but each node is expanded more efficiently. Hence, algorithms with good performance in main memory environments do not necessarily exhibit the same behavior in a data base context.

The performance of Breadth-First when only visited nodes appear in STATES is especially poor. This is because in our formulation of the algorithm all nodes in the STATES relation are expanded in every execution of the append statement. Towards the end of processing, the STATES relation contains many tuples, most of which no longer need to be expanded. Hence, care must be taken in efficiently coding an algorithm of a given type, because large performance differences can be observed.

6. SELECTION OF AN ALGORITHM

In previous sections a query language was described in which a variety of shortest path algorithms can be implemented and performance results are presented for several of them. Since each algorithm is relatively straight-forward to specify in QUEL*, it is apparent that a library of alternate algorithms and of various implementations of the same algorithm could be easily constructed. This section suggests ways in which algorithm selection from such a library could be performed using the DBMS.

We will suggest two levels of facilities that can be implemented in this context. In the discussion which follows, we assume the presence of an algorithms relation

COMMANDS (id, alg-name, type, QUEL-desc, ...)

where "id" is a unique identifier for the routine in "QUEL-desc," "alg-name" is the name of the algorithm implemented by that routine and "type" specifies the function of the algorithm, e.g. "shortest-path." We assume that QUEL-desc contains QUEL* with a variety of parameters. For example, one could specify the following routine (using '$' to denote parameters):

```
retrieve into $result ($name.$source, $name.$dest, cost = $name.$cost)
append* to $result ($result.$source, $name.$dest,
                cost = $name.$cost + $result.cost)
        where $result.$dest = $name.$source
```

Assuming that the above routine has id "mostly-append" one requires a syntax such as:

```
range of C is COMMANDS
execute (C.QUEL-desc with (name = FEASIBLE, source = source,
                    dest = dest, cost = cost, result = TEMP))
where C.id = "mostly-append"
```

In the sequel we assume that a parameterized mechanism such as the above would be used in practice, but for simplicity we code specific examples on the FEASIBLE relation.

The first level of routine selection is the EXECUTE-1 command. Suppose one has several routines in the COMMANDS relation implementing an algorithm with the name "dy-prog". One could specify that one be executed by:

execute-1 (C.QUEL-desc) where C.alg-name = "dy-prog"

After the collection of appropriate routines has been found, the query optimizer must choose exactly one to execute. Our suggestion is to run the optimizer on all the routine descriptions to find the expected cost of each collection of commands. In the case that only normal QUEL appears, the optimizer can generate a complete estimate for the cost using the techniques of [SELI79] and then choose the one with expected minimum cost. In the case that statements in QUEL* appear, the choice is much more difficult because the number of iterations is not known for the various routines. In this case we suggest the following heuristics:

1) Assume that any QUEL* expression will be executed a constant number of times. Choose the expected cheapest routine on this basis.
2) Assume that any QUEL* expression will be executed a number of times equal to the cardinality of the largest relation which it acts on. Choose the expected cheapest routine on this basis.
3) Run each QUEL* expression once and record how many tuples were changed. Assume that the number of iterations is equal to this number. Choose the expected cheapest routine on this basis.

Other heuristics appear possible. It is a problem for future research to generalize a query optimizer to choose a good candidate in this extended environment.

Notice that the query optimizer can make its selection decision based only on expected execution time of the candidate routines. This formulation has several shortcomings. First, a user may have additional constraints. For example, he may insist on obtaining the optimal answer rather than an approximate one. Second, the running time of a routine and the quality of its answer depend crucially on the characteristics of the data. For example, any implementation of a greedy algorithm will perform well on a flat uniform landscape (such as Illinois) but will perform poorly in mountainous terrain. Such data specific information is not captured by the above formulation. Moreover, a knowledgeable user may know that a dynamic programming algorithm is

always preferable to a greedy algorithm unless the terrain is perfectly flat. Such comparative information is not captured by a mechanism minimizing expected algorithm running time.

In order to solve the above problems, we suggest the inclusion of an algorithm selection level, which uses one or more relations containing auxiliary information. Moreover, we will need another extension to QUEL to properly utilize the added information. The QUEL extension is to allow a QUEL qualification to be a data type in a relation. This would allow statements of the form:

execute (relation.QUEL) where relation.qualification

For a given relation with fields of type QUEL and qualification, the QUEL commands would be executed in each tuple for which the qualification field evaluated to true. This construct is used in the auxiliary relations, an example of which is the BEATS relation:

BEATS (winning-alg, losing-alg, condition)

The two algorithm name fields identify algorithms appearing in the COMMANDS relation while the condition field contains a qualification in the QUEL language. A tuple in this relation indicates the conditions under which a given winning algorithm should be chosen in preference to a given losing algorithm. Assuming that there is a relation

path(START,FINISH)

containing one tuple specifying the start and finish points of the desired path, the following insert specifies that the Best-First algorithm is preferable to the Breadth-First algorithm if the distance between START and FINISH is less than 10.

```
append to BEATS (
        winning-alg = "Best-First",
        losing-alg = "Breadth-First",
        condition = "distance(path.START, path.FINISH) < 10)"
```

We expect that a skilled algorithm designer will insert such conditions. To utilize the knowledge contained in this relation, we would expect the designer of an expert system to obtain a command from an end user of the form:

range of CO is COMMANDS

```
execute-1 (CO.QUEL-desc) where CO.type = "shortest-path"
```

The expert system could alter this command to the following:

```
retrieve into CANDIDATES (CO.id, CO.alg-name, CO.QUEL-desc)
        where CO.type = "shortest-path"
range of CA1 is CANDIDATES
range of CA2 is CANDIDATES
range of B is BEATS
delete CA2 where CA2.alg-name = B.losing-alg and
                    CA1.alg-name = B.winning-alg and
                    B.condition
execute-1 (CA2.QUEL-desc)
```

In this way the condition is checked in BEATS whenever both a winning and losing algorithm appear in CANDIDATES and the loser is removed if the condition is true. Hence, the knowledge in BEATS has been used to eliminate choices and the EXECUTE-1 command has fewer options to evaluate. Some other relations which might be useful are:

```
IS-REASONABLE (alg-name, condition)
WONT-WORK (alg-name, condition)
HAS-SMALL-RUNNING-TIME (alg-name, condition)
```

The job of the expert system implementor is to accept a specification from an end user of the command to be run and then to utilize the information in the relations such as the one above to restrict the options to be evaluated. This restriction process can be programmed completely in QUEL. The last QUEL command in such a program should be an EXECUTE-1 command to turn over final routine selection to the data base system.

7. CONCLUSIONS

We have shown that our proposed extensions to QUEL make it possible both to express search algorithms as collections of DBMS commands and to support the selection of a candidate algorithm based on performance considerations. This is a first step to capturing general knowledge bases inside a data base management system.

We have shown that an expert system can employ this extended DBMS to provide unified management of data and part of the knowledge base. For the moment, an expert system designer is required to intervene between the end user and the data base system to specify how algorithm

knowledge should be used to restrict the search for a candidate algorithm. In the future we will investigate how to factor more of the algorithm selection process into the data base system.

We implemented several common search algorithms in a data base context. We found that Breadth-First parallel algorithms consistently outperformed serial Best-First ones. This contradicts the behavior that would be expected on main memory resident data sets. Additionally, data base algorithms were found to be competitive against a hand coded Fortran program.

The requirement that updates be functional was found to cause substantial increase in running time of the dynamic programming algorithm. It is thus desirable to allow non-functional updates as a user-selectable option in a relational data base system.

REFERENCES

[ALLM76] Allman, E. et. al., "EQUEL Reference Manual," ERL Report, University of California, Berkeley, 1976.

[ESWA76] Eswaren, K., "Specification, Implementation and Interactions of a Trigger Subsystem in an Integrated Database System," IBM Research, San Jose, Ca., Research Report RJ1820, August 1976.

[GUTT84] Guttman, A., "Extending a Relational Data Base System to Effectively Manage CAD Data," PhD Thesis, University of California, Berkeley, June 1984.

[MCDO80] McDermott, J., "R1: A Rule Based Configurer of Computer Systems," Carnegie Mellon Univ., Pittsburgh, Pa., Computer Science Report CMU-CS-80-119, April 1980.

[ORAC84] "ORACLE REFERENCE MANUAL," ORACLE, Inc., Menlo Park, Ca., 1984.

[PEAR84] Pearl, J., "Heuristics: Intelligent Search Strategies for Computer Problem Solving," Addison Wesley, Reading, Mass., 1984.

[SELI79] Selinger, P. et. al., "Access Path Selection in a Relational Data Base System," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[STEF78] Stefik, M., "Inferring DNA Structure from Segmentation Data," Artificial Intelligence 11, pp. 85-114, 1978.

[STEF82] Stefik, M. et. al., "The Organization of Expert Systems: A Prescriptive Tutorial," Xerox Palo Alto Research Center, Palo Alto, Ca., Memo VLSI-82-1, January, 1982.

[STON76] Stonebraker, M., Wong E., Kreps P. and Held G., "The design and implementation of INGRES," ACM Transactions on Database Systems, Vol. 1, No 3, pp. 189-222, (Sep. 1976).

[STON83] Stonebraker, M., et. al., "Application of Abstract Data Types and Abstract Indices to CAD Data," Proc. Engineering Applications Stream of the ACM-SIGMOD International Conference on Management of Data, San Jose, Ca., May 1983.

[STON84] Stonebraker, M., "A Novel Specification and Implementation of Data Base Triggers," (in preparation).

[STON84a] Stonebraker, M. et. al., "QUEL as a Data Type," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[ZANI83] Zaniolo, C., "The Database Language GEM," Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.

[ZLOO75] Zloof, M., "Query-by-Example: Operations on the Transitive Closure," IBM Research, Yorktown Heights, N.Y., Research Report 5526, July 1975.