REPRESENTATION AND EXECUTION OF SEARCHES
OVER LARGE TREE-STRUCTURED DATA BASES

A. Reiter *
A. Clute @
J.M. Tenenbaum @


November 1970

Technion

* Computer Science Department, Technion, Haifa, Israel. The senior author wishes to acknowledge the support of the National Science Foundation (Postdoctoral fellowship number 49073) which made possible the writing of this paper.

@ Information Science Laboratory, Lockheed Research Laboratory, Palo Alto, California.
ABSTRACT

An algorithm is described for searching hierarchically structured data bases. Given a perfectly general collection of self-descriptive data organized as a tree, it is possible to define and implement a query language whose search execution strategy utilizes this algorithm. The search to be performed is represented as a tree, and the algorithm is the simultaneous traversal of two trees. The search is "efficient" in looking only at local properties of the data base and in never examining a data item redundantly. The implications of selecting an internal representation and an execution strategy on the semantic and syntactic structure of the resulting query language are discussed.
I. Introduction

Recent years have seen the emergence of a large number of question-answering systems; for a survey, see Simmons [1]. These systems accept input questions in a variety of English-like languages, and attempt to answer the questions on the basis of a body of information called a **data base**, which is stored inside the computer.

Following Woods [2], we conceptually divide the question-answering process into three phases: syntactic analysis, semantic analysis, and retrieval (search execution), illustrated schematically in Figure 1. The first phase consists of parsing the input sentence into a structure which explicitly represents the grammatical relationships among the words of the sentence. Using this information, the second component constructs a representation of the semantic content or "meaning" of the sentence, by canonically translating the sentence into an explicit intermediate form, which reflects unambiguously the structure of the data base. The remaining phase consists of procedures which operate on the intermediate form representation of the sentence in order either to retrieve the answer directly from the data base, or else to deduce the answer from the information contained in the data base. The dotted lines in Fig. 1 represent possible feedback from later phases to aid in parsing and interpretation or to aid the user in reformulating the sentence.

Of the systems known to the present authors, most of the work has gone into designing suitable approaches to the first two phases, that is into the **compilation** phase. The query execution, if mentioned at all in the published accounts, is usually glossed over as being uninteresting or obvious. The systems are implemented (if at all) over relatively small collections of data, sometimes small enough to be core resident. While there is no intention here to minimize the importance of the results obtained to date on natural-language parsing, it is nevertheless true that understanding the user is by no means the only problem to be solved in the design of a query language. Given a very large data base
structured in a certain way, a number of different methods of searching it will typically exist, and these will by no means be equivalent in the cost of search execution, even if they would produce the same answers. Since a particular search strategy chosen by the computer will frequently depend on the formulation of the query by the user, execution efficiency considerations may (or should) have some implication on the syntactic and semantic structure of the query language; these implications have hardly been treated in the literature.

Most query language systems are developed (as in [2]) by "first selecting the semantic primitives and defining an adequate query language without concern for the data structures or for the retrieval techniques." A different approach is taken in this paper. We start out with given data structures (chosen to allow utmost generality for manipulations over hierarchically structured data bases) and with associated retrieval techniques; working backward, we derive the desirable semantic primitives of the language in terms of those search structures which we are willing to execute. The syntax is defined as the last step of the language definition. The resulting language is not guaranteed to be complete, in the sense that there may exist many "reasonable" questions which cannot be asked in the language. On the other hand, the resulting software system is guaranteed to be viable in a real-life setting.

In searching a large data base, the principal overhead is the time involved in physically transferring a record from bulk storage into core. It is especially important

1. Not to visit a physical record unnecessarily; once a record is brought into core all of the information relevant to answering the query should be extracted from it, so that there is no need to look at it again;

2. To have localization of references; if movable-arm direct access devices are involved, localization implies cutting down on arm movement.
While no formalization of the concepts (such as defining a topology over the search space) is attempted here, the strategy described below is intuitively efficient from this point of view: the tree traversal algorithm looks only for local features of the database at any point in the search.

The concepts presented here are being implemented in a system called SQL (for Simplified Query Language). The primary interest in this paper is the representational device RT, into which the user's SQL input is translated by the semantic interpreter. Queries expressed as RT structures can be executed neatly and efficiently; the execution algorithm is presented in Section 5 and extended in Section 6. Not all possible queries are expressable as RT structures, however.

The RT would be a convenient output for a natural-language parser. This paper is not however concerned with parsing English, and the actual choice of the syntax for the user's input language resembles Kellogg's Intermediate Language [3], into which he parses users' input automatically.

SQL is designed and written without any specific database in mind, and is intended for a variety of data files with widely differing logical organizations. The objective is to facilitate the "instant" creation of new management information systems without having to add special-purpose programs. The implementation to be described is most effective for data bases with deeply embedded hierarchical structure. Representation of hierarchical structure in systems such as [4, 5, 6, 7] (designed for associative or relational structures) is frequently clumsy and involves enormous overhead in storage and in retrieval time. The present algorithm is not affected adversely by embedded structure irrelevant to delimiting the search, and makes use of the relevant structure to narrow the search boundaries. The algorithm is least effective with "shallow" data bases which contain a large number of records but little structure within a record or among records. SQL does contain provisions for handling associative relations; the mechanism is described in section 7.
SQL was implemented on an IBM System/360 Model 40 computer at the Lockheed Information Sciences Laboratory. It is part of the ALERT project [8]. It runs under the ALERT time-sharing monitor [9] and makes heavy use of the HODS data management system [10]. The mechanisms incorporated into SQL execution depend vitally on the tree-structure physical organization of the data, and require a HODS "machine" for the implementation. SQL is not otherwise machine-dependent, and can be implemented on any computer on which HODS can be implemented; that is, on any modern computer equipped with a large random access peripheral store. Section 2 of the paper briefly discusses the HODS data structures and services.

In addition to the data manipulation services provided by HODS, SQL requires some appropriate data description mechanism. The one used in ALERT is sketched in Section 3.

2. An introduction to HODS

HODS (for Hierarchically Organized Data Structures) is a software interface between data and the applications programs (such as SQL) which operate on the data. HODS presents applications programs with a single-level virtual store, in which addressing is done symbolically by specifying a path name as a succession of node keys. The data groups and items may be of variable length. HODS automatically handles the function of partitioning the data base into physical records and of bringing the records into and out of core. All data are subject to dynamic relocation under HODS control.

The logical organization of the data base is that of a tree. Figure 2 shows a typical application file as it would be represented under HODS. A typical path name (to the field which contains B. Jones' age) is "Personnel application". "Employee file". "Jones, B.". "age". The following points should be noted:
1. Most nodes have a key. This variable-length key must be unique within the parent node, but may be duplicated in other nodes. Keyed nodes may be accessed either by key or by (a left-right) sequence number; unkeyed nodes may only be accessed by sequence number.

2. Atomic nodes contain unstructured data. The information content of other nodes is usually precisely their filial set, except that a short descriptive element may be stored within any node.

3. No distinction exists at the level of HODS between specific keys (e.g. "Jones, B.") and generic keys (e.g. "age"). Specific keys can alternately be represented as atomic data; for instance, instead of using "Jones, B." to key the employee record, it is possible to represent the information as an atomic subnode of the employee record under a key of "employee name". Since the subnodes within a node are ordered on their keys, direct retrieval on key is much faster than searching the entire node to find the occurrence of a given atomic data value. However, keys are subject to the following restrictions: the occurrence of a key within a node must be unique, and a subnode can have at most one key. Thus the system designer has to make choices in the logical organization of the data base which can significantly affect retrieval times.

4. No distinction whatever exists between repeated instances of the same type of subnode within its parent (e.g. employee record within the employee file, or the repeating group describing education) and totally disparate data items which happen to appear as subnodes within the same node (e.g. "age and "education").
The physical organization of the data is also of a tree, except that the use of pointers from parents to their filial set is avoided as much as possible. The parent-child relationship is usually represented by physical set containment, within the same physical record; only when unavoidable is the actual subnode replaced by a pointer to another physical record containing the subnode. As a result, HODS helps minimize the number of accesses to peripheral storage. When a given node of the tree is in core, there is a high probability that at least part of its filial set is also in core.

To manipulate data, a program must declare variables which take on as values pointers into the data base. These variables are called data windows; a program may have any number of these at a given time, pointing at various nodes in the data base tree. Data windows are dynamically created (=declared) and erased.

When a data window is created, it is assigned an initial value. This pointer can be moved around with the LOCATE command. The possible nodes of tree traversal, one node at a time, are DOWN, UP, NEXT, and BACK. For DOWN, a parameter KEY=<key of desired subnode> or SEQ=<sequence number of desired subnode> must be indicated. New data may be added into the node currently pointed to by the data window via the INSERT command. Unneeded data is purged via the DELETE command.

The pointer is moved to a specific node, by issuing the LOCATE command repeatedly. Thus, in order to get to B. Jones' age, the following sequence of commands have to be executed:

CREATE p (This declares p as a data window)
LOCATE p,DOWN,KEY="personnel application"
LOCATE p,DOWN,KEY="employee file"
LOCATE p,DOWN,KEY="Jones, B."
LOCATE p,DOWN,KEY="age"

At this point, p contains a pointer to the field "age", which incidentally is guaranteed to be in core memory by this time.
HODS is intended to be used recursively. Recursive use of subroutines is also facilitated by the program linkage mechanism in the ALERT time-sharing monitor.

3. The data description mechanism

A comprehensive description of the structural organization of a data base must be available to the system in order to be able to interpret input from a user and to build a suitable internal representation of the query. Before commencing the query dialogue, the user must somehow specify to the computer which segment of the data base is of interest to him; as part of this specification he also indicates the data descriptor to be used by the system.

The description of the logical structure of a particular subtree of the data base is accomplished by a special data element called a FAD (File Attribute Dossier). The FAD is a file schema which contains a semantic description of the subtree (the names of various subfields and their structural relationship to each other) as well as a syntactic description (e.g. admissible data values for the particular field). It is derived from similar entity called a DDR (Data Descriptor Record) described in [5].

A FAD is associated with a given node in the data base tree (e.g. with the "employee file") and generically describes the form of the various fields contained within the descendant subtree. The FAD is itself a subtree (as are all HODS data entities). All of the subfields which can appear somewhere with the data subtree appear in the FAD which can at the same level, regardless of the actual structure of the data base. (This enables the system to find the required entry quickly, without having to search the entire FAD; a user thus can specify a field name without having to specify all of the intermediate nodes along the path.) Ambiguities in field name specification are immediately apparent. For fields with generic keys (such as "Jones, B.") a FAD may contain a generic description for the class of keys; for example, the FAD may contain an entry called "name" which identifies the class of keys "Jones, A.", "Smith, L.", etc. Thus the on-line
user does not need to know whether the system designer has chosen a particular
data element as they key or has represented it as an atomic data value;
from the point of view of SQL semantics the two are equivalent. The FAD can
also handle synonyms for field names.

The FAD describing the employee file is shown in Figure 3. Note
that multiply-defined field names (such as "address") cause no special
problems: the semantic interpreter can easily recognize ambiguities and
direct the user in resolving them using the FAD description.

A wide variety of field descriptions can be handled by the FAD,
including the recoding of keys for internal representation in order to
save storage space. The FAD can also contain auxiliary information
(such as the availability of inverted files, the number of occurrences
of certain nodes, etc.) which can be used in a preliminary computer-user
dialogue to elicit a more effective query formulation from the user.

4. The syntax of SQL

In its simplest form, the syntax of an SQL sentence is a
single question followed by one or more qualifiers:

\[
\text{sentence} :: \text{question} / \text{sentence}\langle\text{qualifier}\rangle
\]
\[
\text{question} :: \langle\text{field name}\rangle
\]
\[
\text{qualifier} :: \langle\text{field name}\rangle = \langle\text{list of values}\rangle
\]

A simple fact-retrieval question in SQL might look like this:

4.1. Age? Name = 'Jones, B.';

Much more complex questions are possible. For example, to obtain a
list of all employees who have an MA, MSc, or PhD degree, are over 35, and
have some experience with circuit design (job type 19) the following
query sentence can be formulated:

4.2. Name? Degree = 'MA', 'MSc', 'PhD'; Age '35'; Job type = '19';

A list of data values implies a logical OR. Lists of values for relational
operators other than = are handled by pairing operators with data values:
Age >'25', <'65'. In the case of possible ambiguity in field names, such
as in the case of the field FROM, a descriptor can be supplied: FROM(EDUCATION),
or FROM (WORK HISTORY), or FROM(JOB HISTORY); the last two are equivalent.

The current syntax of SQL makes no provisions for infix operators,
for a logical OR connective on qualifiers, or for nesting queries; it is
however sufficient to illustrate the simple and efficient execution algorithm.
Section 6 discusses the implementation of the execution algorithm to accomo­
date the above syntactic extensions.

Besides the easily-handled problem of ambiguous field names, the
current syntax of SQL allows another kind of semantic ambiguity. What for
example is the meaning to be assigned to the following sentence?

4.3. Name? Employer="Lockheed"; Supervisor="Duncan";
Possibility 1 is "Who has once worked under Duncan at Lockheed?". Alternately,
the meaning might be "Who has once worked for Lockheed and has also (possibly
somewhere else) worked under a supervisor named Duncan?" In the current
version of SQL, it is always the first meaning which is chosen (because it is
more conveniently handled for execution) and there is no simple way to
obtain an answer if the second meaning is desired; the mechanisms described
in Section 5 cannot execute the implied search. Treatment of the second form
is discussed in Section 6.

5. The execution of searches
Consider the following two queries:
5.1. Name? Employer = "Lockheed";
5.2. Job type? Employer = "Lockheed";
Although they appear to be syntactically similar, they are very different semantically and imply strikingly different search requirements. In (5.1) it is desired to know the names of all employees who had formerly worked \textit{at least once} for Lockheed. That is, "Name" is the desired output value as long as the value Lockheed appears at least once in the field "Employer" in that particular subtree. Once its occurrence has been verified, no further searching of this particular "employee record" need be done.

For 5.2. however it is desired to know all "Job types" held by employees during previous associations with Lockheed. This requires that for each employee record, each occurrence of Job History must be checked to see if the Employer happens to be Lockheed. If so, all occurrences of the field "Job type" associated with that particular employment must be added to the output list.

This semantic interpretation is the "natural" one given the logical organization of the data.

The distinction between queries 5.1. and 5.2. is made sharp and algorithmic when the implied searches are expressed by the representational device RT (for resolution tree). This representation is developed by the semantic interpreter from the user's input and the FAD. The substantive content of the data base tree DBT is not examined until the execution stage, and does not enter into the creation of the RT.

The RT is a file schema which describes precisely that part of the DBT relevant to answering the query. It identifies each node relevant to the query, and labels it with some auxiliary information used when executing the search.

The RT contains two types of branches: those which lead to the desired output of the query, and those which contain qualified nodes, i.e. those nodes whose corresponding values in the data base tree DBT need to be checked against values provided by the query sentence. The latter type of branch is known as a \textit{verification path}. Thus, in query 5.1. the node
"Employer" is on a verification path. A node which must be traversed on the way down to the query output node is said to be on the Q path. Nodes which are on the verification path but not on the Q path are called V nodes; the other RT nodes are called QV nodes. The desired query output node is simply labelled Q.

Note that in searching the DBT for output values, a set of candidates must be selected for the QV (and Q) nodes, and for each such collection of candidates, the corresponding V nodes must be visited to determine whether all of the specified qualifiers are satisfied. Note that all possible candidate sets for QV and Q nodes must be proposed, while for a given set of candidates, each qualifier must be satisfied by any one of the occurrences of the corresponding V node. Thus, in 5.1., "Name" is a Q node; for each value of "Name", any occurrence of the value "Lockheed" in the node "Employer" is sufficient to satisfy the qualifier. Conversely, in 5.2. the node "Employer" is a QV node, and all of its occurrences are potentially relevant.

For each node in the RT we must also know what selection criteria if any the candidate values selected from the DBT are subject to. There are essentially three possibilities:

L - The value in this node (or the key of this node) must be screened using the attached list of admissible values provided by the query sentence;

* - This node is unqualified, and hence all occurrences of this node are potentially valid;

F - This node (with a fixed key which is given) must be traversed in order to get to the qualified nodes or to the query output below this node. The key for an F node typically is supplied from the FAD, and reflects the logical structure of the data base; it is otherwise similar to an L-node with an = condition provided on the key.
The RT's for queries 5.1. and 5.2. are shown respectively in figures 4 and 5. For the syntax sketched above, the RT for query 4.2. is given in figure 6.

Unlike the FAD, the organization of the RT is not inverted; the hierarchical relationships among the various relevant nodes is represented directly. However, for the purpose of efficiency, it is important that the Q or QV nodes appear after all of the V nodes in the same parent in the RT; this forces all possible verification to be done before candidate answers are elected. In general, the order in which the nodes appear in the RT will affect the efficiency of the search, and it may be possible to introduce heuristics for deriving a good order, or (better) to allow the user to suggest the order. These refinements have not been attempted but can in principle be added later.

The query execution can be expressed very neatly as the simultaneous traversal of two trees: the RT and the DBT. The execution is highly recursive and very efficient with respect to accesses to peripheral storage. First we must give some definitions.

5.3. A node of the RT will be said to be verified relative to a given DB path down to and including the corresponding DB node if both of the following conditions hold:

5.3.1. The corresponding DB node satisfies the list conditions, if any; and

5.3.2. Either a) The RT node is terminal; or
   b) Each descendant of the RT node can be verified, i.e. if there exists an extension of the current DB path corresponding to each descendant RT node relative to which that RT node is verified.

The corresponding DB node value is said to be the verifying value, and the DB path to this node is called the verifying path. Note that the verifying path contains only partial information as to how this RT node is verified, unless the RT node is terminal; in general, some choice among
possible values in the DB had to be made at lower levels which are not explicit in the verifying path itself.

The above definitions are not meant to imply that every antecedent of a verified node is itself verified. Several given conditions may affect a node, and one of these may affect only some of the descendants. The other descendants may be verified, but the parent may eventually be rejected because the one condition is not met. Ultimately, of course, we only want those nodes which are verified with respect to all conditions. This leads to the following definition:

5.4. A node value of the DB tree is said to be a valid query output if both of the following hold:

5.4.1. The corresponding RT node is a Q node;

5.4.2. It and all of its antecedents are verifying values for the corresponding RT nodes.

The algorithm for query execution is now clear. We wish to traverse the DB tree looking for valid query outputs. This is done by trying to verify the topmost RT node, which will in turn cause all lower RT nodes to get verified. By arranging the RT so that the Q nodes are verified last, we can be sure that the moment we find a verifying path to a Q node we are looking at a valid query output; we simply add it to the output list and proceed.

Figure 7 shows the basic plan. Two data windows are required for traversing respectively the RT and DB; a third data window will enable us to insert values into the output list. (Other data windows for traversing qualifier list values are not shown.) Once all data windows are initialized, execution reduces to the verification of the top RT node.
Verification consists of two highly recursive routines, each one calling the other. They are shown respectively in Figures 8 and 9; the flow charts correspond closely to the definition 5.3. "Verify RT node" consists of looking at all descendants of an RT node to see if a verifying DB value can be found; if an unverifiable subnode is encountered, the current RT node is also unverified. "Find verifying DB value" consists of choosing values from the DB in succession for a given node. For V nodes, as soon as a verifying value is found, the search is abandoned. For Q or QV nodes, all values are examined. Any time a Q node is verified, the corresponding value is added to the output list.

As an example of the operation of the algorithm, consider query 5.2. and its RT (shown in Figure 5). We start out to "Verify RT node"; this takes us down from a node not shown in the figure to the topmost node of the RT. "Find Verifying DB value" causes the first employee record to be selected. "Verify RT node" takes us to the "Job History" field in the RT; from there we find the "Job History" field in the current employee record in the DBT. "Verify RT node" we pick up the * field in the RT, and (going back to the DBT) look at the first occurrence of a job. Back at the RT, we find the field "Employer"; going over to DBT, we check the value there against "Lockheed". In this case, there will be only a single value at the DBT node; more generally, we would keep checking the values until we find a match ("verified") or until the list is exhausted ("unverified"). If verified, we go up and descend along the second RT branch, and for all occurrences of the "Work History" instances we add the corresponding "Job Type" to the output list.

Whether verified or not, we then go up to the * under "Job History" in the RT in order to pick up the next instance of a "Job" for this employee. The verification and query output are then repeated. This continues until no further instances of "jobs" remain. At that point, we go up to the topmost * in the RT which produces (from the DBT) the next employee record, and repeat the process.
The algorithm can be seen to have the property that any given DB node is examined at most once. Maximum information is extracted from the node before it is abandoned. The examination of a node consists precisely of the examination of the relevant subnodes.

It can be seen that the algorithm is perfectly general, and operates regardless of embedded structure. The implementation of the algorithm is not only efficient, but also is remarkably short, due to the powerful tree-traversal commands supported by HODS.

6. Extensions to SQL

In this section we consider how various extensions to SQL could be implemented. We are concerned with the internal representation by the RT of the additional syntactic entities, and do not consider possible problems in the semantic interpretation necessary to translate the user's input into the RT.

6.1. Disjunctions. To allow qualifiers to be connected by an OR, we need only be able to mark certain RT nodes as being OR-nodes. The RT form of the query "who has experience with JOB TYPE 19 or has been working since 1955", expressed as

6.1.1. NAME? JOB TYPE = '19' OR FROM (WORK HISTORY) <1955 is shown in Figure 10. To verify an OR node in the RT, we need to verify any one of its branches. We extend definition 5.3.:

6.1.2. An OR node of the RT will be said to be verified (relative to a given DB path down to and including the corresponding DB node) if any one of its descendant nodes can be verified, i.e. if there exists an extension of the current DB path corresponding to one of the descendant nodes of the RT relative to which that RT node is verified. That extension of the DB
path, as well as all extensions along paths corresponding to the other descendant nodes of the RT, are said to be verifying values.

(The italicized phrase is equivalent to the propositional calculus axiom $\exists x \forall y (A \lor B).$)

With this definition, the required modifications to Figure 8 and 9 are minor. Answering a query still consists of obtaining all valid query outputs according to definition 5.4.

The presence of OR nodes suggests for convenience also the addition of an AND to the RT. This is a dummy node, which does not have an DB node corresponding to it. Its use is illustrated in Fig. 11 in the RT for the query.

6.1.3. NAME? JOB TYPE=19 OR (FROM(JOB HISTORY) <1965 AND SUPERVISOR='DUNCAN')

6.2. Forks. Even with the addition of OR nodes, the semantic structure of SQL is limited. Consider the simple question 6.2.1. "List all of the schools which our MIT graduates attended". The sentence

6.2.2. SCHOOL? SCHOOL='MIT'

will produce the tautological answer 'MIT', since the RT representation (given in Fig. 12a) treats both occurrences of the bound variable "SCHOOL" as being the same occurrence. A fork is an extension of the RT which allows multiple occurrences of a field name as subnodes of the same parent node within the RT so that values could be assigned to them independently.

The basic use of the fork is illustrated in Fig. 12b. With the proper syntactic extensions of SQL to allow formulation of question 6.2.1., the resulting RT is shown in the figure. Note that no modifications are required either to definition 5.4 or to the execution algorithm. For example, the RT form of question 4.3. under interpretation 2 is shown in Figure 13.
By using forks, the scope of some qualifiers could be limited without affecting other parts of the query sentence; the branches of the database which are qualified differently depending on context appear a number of times in the RT with the appropriate qualifiers.

Nested queries, especially in conjunction with infix operations, can also be represented with forks. Consider for example the question "Who made more at Boeing (at any one time) than his average salary at Lockheed?" This query, given the proper syntactic extension of SQL, might look as follows:

6.2.3. NAME? (EMPLOYER = 'BOEING', SALARY > X1),

X1 = (AVERAGE (SALARY? EMPLOYER = 'Lockheed'))

Note that the average is to be taken for each given employee, not (as is a reasonable alternative) over all employees. The RT (with some extensions) is shown in Figure 14. There are still some existential problems to be resolved (What does the question mean if the employee did not work for Lockheed?); ignoring these, the execution is straightforward.

It should however be realized that introduction of the forking device may cause many DB nodes to be examined more that once. Thus, the intuitive claim as to the general efficiency of the SQL algorithm does not hold for searches with forks in the RT. For many such questions, the local search algorithm of SQL may not be appropriate; it may be preferrable to use an algorithm which tries to apply all global criteria to each candidate datum value.

6.3. The "front end" of SQL. To date, not much effort has been devoted toward making SQL more convenient or "natural" to use. It is however, planned to add a syntax-directed parser front-end to SQL, similar to the one described in [12]; a candidate vehicle for this purpose is McKeeman's XPL [13].
7. **Inverted files.**

An inverted file expresses associative structure by grouping together data nodes which contain the same value for some given attribute. This grouping is typically represented internally by storing, for each attribute value, a list of pointers to the data nodes containing that value. Such lists can greatly speed up retrieval when the inverted attributes are explicitly specified in a query. On the other hand, "inverting" on every attribute value frequently involves a prohibitive overhead in storage space and in other ways.

In ALERT, a system designer has the option of specifying some fields (=attributes) as invertible, and these are indicated in the FAD. The inverted file entry for a given attribute value consists as usual of a list of pointers. The nodes pointed to are (by convention) the subtrees one level below that of the FAD; in the example of Fig. 2, the pointer would be to the "employee records". Each "pointer" is the key of the corresponding node; because of the location conventions, this is sufficient to specify completely the path to the node. Thus, in Fig. 2, an inverted file entry would contain the names of employees possessing some attribute value.

In SQL, inverted files are utilized in a pre-search phase. Returning to query 4.2. suppose that an inverted file for JOB TYPE exists. We can then read the list of employees whose JOB TYPE = 19, and use this list to supply potential names for the search instead of looking at all names in succession. The effect is precisely that of changing the top node in the RT from specification type * to L, exactly as if the user had himself supplied the list of potential names.

Note, however, that the qualifier JOB TYPE = '19 has additional information value and cannot be deleted from the RT even after the inverted list has been added to RT. This is because the qualifier might qualify lower nodes. To see this, consider the query sentence
Without the existence of inverted files, the SQL algorithm will search for employees who have done JOB TYPE = 19 under Duncan, which is what is desired. If the inverted file is however used to replace the qualifier JOB TYPE = 19, the effect would be to find employers who have independently done JOB TYPE 19 and have also worked under Duncan.

If several different inverted files exist, the pre-search phase can form new lists by Boolean operations on the inverted file lists, as is usual for inverted file systems. The resulting list is then used to drive the search phase.

8. **Summary and conclusions**

We have described a "bottoms up" approach to the design of query language. Starting with some notions on how to represent the data base in storage and how to express the logical relationship of the component data elements, we derived some concepts of how to effectively search such a collection. This led to the development of the RT as an internal representation of a search. Finally, given the RT, we were ready to define the class of questions answerable by our language, and to describe its syntactic and semantic structure.

How good was the resulting language? Not too bad, judging by the ability of the implemented version of SQL to answer useful questions. Some questions, however, could not be formulated in the implemented version. Others, which could be expressed in the syntax, would cause the semantic interpreter to assign a meaning different from the intended one. This was due to the fact that the interpretation of the scope of qualifiers was determined by the internal query representation. No other interpretation was supported.
The authors have yet to determine the extent to which the expanded RT (described in section 6) constrains the semantics of the language.

SQL implementation would have been very difficult without the powerful system primitives of ALERT. The HODS data management system was especially indispensable. The authors contend that data management is a key problem in computing today; developing powerful general-purpose software for data management is a worthwhile investment before undertaking special-purpose systems. Other useful ALERT primitives include device-independent user-communication modules, and a specially developed on-line language for specifying on-line dialogues to be described in a forthcoming paper.

The intention of the ALERT project is to develop a set of powerful tools which would enable implementation of on-line management information systems with little if any additional conventional programming. The development of SQL was a necessary condition to demonstrate the feasibility of the ALERT project. SQL contains sufficient power even in its first version to satisfy this goal.

SQL is not yet a production system, and not enough experimental evidence exists to substantiate or contradict the authors' claims as to the efficiency of the search algorithm. Nevertheless, on the basis of the SQL experience, we are willing to conclude that the RT is an adequate tool for the internal representation of searches over hierarchical data bases, and therefore a suitable target for the compilation phase of a query interpreter.
Figure 1. Basic organization of a question-answering system (after WOODS [2]

Technion - Computer Science Department - Technical Report CS0010 - 1970
Figure 2. Logical data organization for a typical application under HODS. Each subnode of "employee file" is informally called an "employee record". Square nodes indicate atomic data.
Figure 3. Fad describing organization of the employee file. Square boxes denote atomic nodes each of which contains the syntactic description of the corresponding field. Circles indicate non-terminal nodes, and contain the key. '* ' indicates that every occurrence of the corresponding data base node is relevant, i.e. denotes an occurrence of a member of a repeating group.
Figure 4. Resolution tree for the query

NAME ? EMPLOYER = Lockhead.

Figure 5. Resolution tree for the query

JobType = EMPLOYER = Lockhead.
NAME? Age > '35'; Degree = MA, MSE, PhD; Job type = '191.

Resolution tree for the query.
Figure 7. Basic flow chart for search execution, given the RT.

Figure 8. "VERIFY RT NODE" flowchart.

Note: "Locate Down" which is not successful still requires "Locate Up" to negate its effect.
Figure 9. Choosing candidate values from the data base.

See note on Figure 8.
Resolution tree for query

NAME of job type = '19'

\[ \text{NAME of job type = '19'} \]
Use of the fork in delimiting scope of qualitizers.

Figure 12.

are now possible.

Same query with scope delimiters as in (6.2.1). Many output

scope delimitation. Only output can be MIT.

A Representation of the query: SCHOOL = MIT, without

Technion - Computer Science Department - Technical Report CS0010 - 1970
Figure 13. Representation of query (6.3) under meaning 2.

Figure 14. RT representation of query (6.2.3). Note nested query Salary:\ Employer = 'Lockheed' on left side of fork node JOB HISTORY. When query is answered, the operator AVERAGE is to be applied and result assigned to query variable X1.
REFERENCES


[1] Bleier, R.E., "Treating hierarchical data structures in TDMS" 


pp. 617 - 635.