HANDLING COMPLEX QUERIES IN A DISTRIBUTED DATA BASE

Technical Note 170

October 8, 1979

By: Robert C. Moore
Artificial Intelligence Center

SRI Projects 6891 and 7910

The research reported herein was sponsored by the Defense Advanced Research Projects Agency under Contracts N00039-78-C-0060 and N00039-79-C-0118 with the Naval Electronic Systems Command.

ABSTRACT

As part of the continuing development of the LADDER system [1] [2], we have substantially expanded the capabilities of the data base access component that serves as the interface between the natural-language front end of LADDER and the data base management systems on which the data is actually stored. SODA, the new data base access component, goes beyond its predecessor IDA [3], in that it accepts a wider range of queries and accesses multiple DBMSs. This paper is concerned with the first of these areas, and discusses how the expressive power of the query language was increased, how these changes affected query processing in a distributed data base, as well as what are some limitations of and planned extensions to the current system.

CONTENTS

I INTRODUCTION

As part of the continuing development of the LADDER system [1] [2], we have substantially expanded the capabilities of the data base access component that serves as the interface between the natural-language front end of LADDER and the data base management systems on which the data is actually stored. SODA, the new data base access component, goes beyond its predecessor IDA [3], in that it accepts a wider range of queries and accesses multiple DBMSs. This paper is concerned with the first of these areas, and discusses how the expressive power of the query language was increased, how these changes affected query processing in a distributed data base, as well as what are some limitations of and planned extensions to the current system.

To explain the new features of SODA, it will be useful to review briefly the capabilities of IDA. IDA is designed to access a relational data base. That is, it expects the data base to be organized as a set of relations (files), each of which contains a set of tuples (records) that are in turn composed of various fields. The IDA query language permits the user to view the entire data base as if it were a single relation, with IDA being responsible for planning which actual data base relations have to be accessed to answer the query. An IDA query is interpreted as a request to:

(1) Generate the set of all tuples satisfying a given description expressed as a Boolean combination of simple comparisons on the fields of the tuple.

(2) (Possibly) select the member of the set for which some attribute is largest or smallest, or count the members of the set.

(3) Return the values of certain attributes for each member (or for the selected member) of the set.

For instance, in the Blue File command and control data base [4] which we have been using, the English query, "What is the longest American ship?" could be expressed using IDA as:

```
((* MAX LGHN)(NAT EQ 'US')(? NAM))
```

The term (NAT EQ 'US') tells IDA that the tuples we are interested in are those for which the NAT field has the value 'US', i.e the tuples pertaining to American ships. The term (* MAX LGHN) tells IDA that we want to select from this set of tuples the tuple for which the field LGHN has the highest value, i.e. the tuple for the longest American ship. Finally, the (? NAM) field tells IDA that we want to return the value of the field NAM from this tuple, i.e. the name of the longest American ship.

IDA would interpret this query by finding the smallest set of relations in the data base that contains all the fields mentioned in the query and specifying to the DBMS what it believes to be the semantically meaningful links among those relations. IDA then generates a program in the DBMS access language that interprets the query with respect to these selected relations.

This approach limits the expressive power of the query language in a number of ways. First, only one set of objects can be talked about in each query. The only way in which two sets of objects can be referenced is if the set that the query is "about" is their intersection or union. Thus we can express the query "Which ships are American submarines?" (intersection), or "Which ships are American or are submarines?" (union), but there is no way to express "Which American ships are less than 100 miles from which submarines?"

Another restriction is that only one maximization, minimization, or count operator is allowed in each query, and it must be applied after all other operations. For example, we cannot express as a single query "How many ships are in each ship class?" since this requires forming a set of counts, rather than simply counting a set. Also, we cannot express "Which ship class has the most ships in it?" since this requires a count operator and a maximization operator in the same query.

Finally, only Boolean restrictions are allowed in specifying a set of objects. That is, all restrictions must be simple comparisons between fields or predefined functions of fields (such as distance functions computed on position fields), or combinations of simple comparisons using AND and OR. Thus IDA gives us no way to express a restriction involving a quantifier as in "Which American ships are more than 500 miles from every American port?"

II EXPRESSING COMPLEX QUERIES IN SODA

The SODA query language has features that enable it to overcome all of the limitations of IDA discussed in the previous section. It allows queries that refer to more than one set of objects, it permits queries to specify the logical scoping of operations, and it allows quantifiers to be used in specifying restrictions on sets of objects. This section informally discusses a number of examples which illustrate these points. The details of the syntax of the SODA query language may be found in Appendix A.

In the examples, we will assume that we have the following subset of a simplified Navy command and control data base:

```
SHIP:      (NAM,  CLASS,  TYPE,  NAT,  LGHN,  POS)
SHIPCLASS: (CLASS,  TYPE,  LGHN,  DRAFT,  BEAM)
PORT:      (PNAM,  PNAT,  PPOS)
```

In the SHIP relation, NAM is the name of the ship, CLASS is her class, TYPE is her type (e.g. 'SS' for submarine), NAT is her nationality, LGHN is her length, and POS is her current position. The SHIPCLASS relation gives information that is common to all ships of the same class. CLASS, TYPE, and LGHN are as in the SHIP relation, and DRAFT and BEAM are the corresponding dimensions of the ships in the class. In the port relation, PNAM is the name of the port, PNAT is the country in which the port is located, and PPOS is the geographical position of the port. We will also assume that the DBMS has the ability to compute the function GCDIST, which gives the great circle distance between two geographical locations.

SODA avoids the first limitation of IDA, the inability to refer to more than one set of objects per query, by using an IN expression to associate a variable with each of the sets we want to mention in the query. The query "Which American ships are less than 100 miles from which submarines?" (which could not be expressed in IDA) can be expressed in SODA as:

```
((IN S1 SHIP ((S1 NAT) EQ 'US'))
 (IN S2 SHIP ((S2 TYPE) EQ 'SS'))
 ((GCDIST ((S1 POS) (S2 POS))) LT 100)
 (? (S1 NAM))
 (? (S2 NAM)))
```

In this SODA query the expression (IN S1 SHIP ((S1 NAT) EQ 'US')) sets the variable S1 to range over tuples in the SHIP relation for which the NAT field has the value 'US', i.e tuples for American ships. Similarly, the expression (IN S2 SHIP ((S2 TYPE) EQ 'SS')) causes S2 to range over tuples for submarines. Then for each pair of ships in the Cartesian product of these two sets, the additional restriction ((GCDIST ((S1 PTP) (S2 PTP))) LT 100) is applied. That is, we check whether the great circle distance between the two ships is less than 100 miles. For each pair of ships that satisfies all these restrictions, we return the names of the ships. This is indicated by the selectors (? (S1 NAM)) and (? (S2 NAM)).

We can illustrate SODA's ability to express the relative scoping of operations with the query, "How many ships are in each ship class?" This could be expressed in SODA as:

```
((IN C SHIPCLASS)
 (COUNT CNT1
        (IN S SHIP ((S CLASS) EQ (C CLASS))))
 (? (C CLASS))
 (? CNT1))
```

The form of a counting operation is a list where the first element is the symbol COUNT, the second element is a count variable, and the rest of the list is a subquery which defines the set of tuples to be counted. The effect of a count operation is to set the value of the count variable to the number of tuples in the indicated set. In this example, since the set to be counted is defined in terms of the field

(C CLASS), and since this occurrence of C is bound outside of the COUNT expression and ranges over all tuples in the SHIPCLASS relation, the query is interpreted to mean that the count operation is to be performed once for every tuple in the SHIPCLASS relation. Thus, the interpretation of the entire query: for each tuple in the SHIPCLASS relation, count the number of tuples in the SHIP relation which have the corresponding value for the CLASS field and return the name of the class and the count.

An example of of a COUNT and a MAX in the same query is provided by the SODA representation of the query, "What ship classes have the most ships in them?":

```
((MAX CNT1
      (IN C SHIPCLASS)
      (COUNT CNT1
             (IN S SHIP ((S CLASS) EQ (C CLASS)))))
 (? (C CLASS))
 (? CNT1))
```

This query simply embeds the body of the preceding query inside a maximizing operation over the count variable. The basic form of a maximizing operation is a list where the first element is the symbol MAX, the second element is the term to be maximized, and the rest of the list is a subquery that defines the set of tuples to be maximized over. In this case the term to be maximized is CNT1 in the set consisting of the tuples in the SHIPCLASS relation augmented by the corresponding values of CNT1, the number of ships in each ship class. The effect of the MAX operation is to set the occurrences of the variables bound by the MAX (in this case C and CNT1)) to range over the values for which the maximized quantity has the greatest value. So in this example, the MAX operation sets the variable C to range over the tuples in the SHIPCLASS relation for the ship classes with the most ships in them and sets CNT1 to the corresponding number of ships. The rest of the query simply returns the name of those ship classes and the number of ships they contain.

Finally, SODA includes several types of quantifiers that can be used to express complex restrictions on the sets of objects referenced by queries. As an illustration of the use of a quantified restriction, recall that there was no way in IDA to express the query "Which American ships are more than 500 miles from every American port?" In SODA this could be expressed by:

```
((IN S SHIP ((S NAT) EQ 'US'))
 (ALL (IN P PORT ((P PNAT) EQ 'US'))
      ((GCDIST ((S POS) (P PPOS))) GT 500))
 (? (S NAM)))
```

The first line of this query restricts the system's attention to American ships via a simple restriction on the SHIP relation. The second expression in the query further restricts this set but involves a universal quantifier. The simplest form of a universally quantified restriction is a list consisting of the symbol ALL, an IN expression, and any number of restrictions. An ALL restriction is satisfied if all the tuples in the set specified by the IN expression satisfy all the restrictions in the list. If there is more than one binder expression in the list, then the join of the sets they specify must satisfy all the restrictions in the list.

In the current example, all the values of P that satisfy

```
(IN P PORT ((P PNAT) EQ 'US'))
```

must also satisfy

```
((GCDIST ((S POS) (P PPOS))) GT 500)
```

for the ALL restriction to be satisfied. Informally, this means that all American ports must be more than 500 miles from the ship in question, for that ship to meet this restriction. Finally, the NAM field from every tuple that meets these restrictions is returned to the user.

A SOME restriction has the same syntactic form as an ALL restriction, the difference in interpretation being that the restriction is satisfied if <u>some</u> tuple in the set specified by the binder expressions satisfies the other restrictions within the SOME expression. Thus, to change the previous query to "Which American ships are more than 500 miles from some American port?" we only have to replace the ALL by a SOME:

```
((IN S SHIP ((S NAT) EQ 'US'))
 (SOME (IN P PORT ((P PNAT) EQ 'US'))
       ((GCDIST ((S POS) (P PPOS))) GT 500))
 (? (S NAM)))
```

Notice that in these examples, there are some restrictions placed inside the IN expression itself and some restrictions placed after the IN expression. In a SOME restriction this distinction is of little consequence, since placing a restriction one place or the other does not change the interpretation of the query. If we place a restriction inside an IN expression, we are using it to define the set that is being quantified over. This is equivalent, however, to quantifying over a less restricted set, but being more restrictive as to the additional conditions that one of the members of the set has to satisfy, which is the interpretation of placing a restriction outside the IN expression. Thus, we could have expressed the previous query by either of the following expressions:

```
((IN S SHIP ((S NAT) EQ 'US'))
 (SOME (IN P PORT)
       ((P PNAT) EQ 'US')
       ((GCDIST ((S POS) (P PPOS))) GT 500))
 (? (S NAM)))
```

or

```
((IN S SHIP ((S NAT) EQ 'US'))
 (SOME (IN P PORT ((P PNAT) EQ 'US')
                  ((GCDIST ((S POS) (P PPOS))) GT 500)))
 (? (S NAM)))
```

In an ALL restriction, however, this distinction is crucial. If we move a restriction from inside an IN expression to outside, the interpretation is changed completely, since instead of the restriction partially defining what set is being quantified over, it partially defines the condition that all the members of the set must meet. In this respect the syntax of the SODA query language is designed to mirror the syntax of English, so that the process of translating from English to SODA will be simplified. The idea is that the restrictions derived from noun phrase modifiers like "American" in "all American ships" would be placed inside an IN expression, but restrictions that come from predicate modifiers would be placed outside the IN expression. If this rule is followed, then the resulting SODA queries will exactly capture the difference between "Are all American ships submarines?" and "Are all ships American submarines?" Conversely, the SODA queries for "Are some American ships submarines?" and "Are some ships American submarines?" will be logically equivalent, as are the English questions.

As a final point on this topic, it should be noted that, although the two questions with "some" must have the same answer, they do differ slightly in what they suggest about the assumptions of the person asking the question. "Are some American ships submarines?" suggests that he believes that there are American ships, whereas "Are some ships American submarines?" suggests only that he believes that there are ships. As Kaplan [5] has pointed out, it can be very important to inform the user of a data base system when the assumptions behind his queries are wrong, so that he can properly interpret the answers he gets from the system. The distinction in SODA between restrictions inside an IN expression and those outside could be used to differentiate the restrictions whose satisfiability the user is <u>assuming</u>, from those whose satisfiablity he is <u>questioning</u>.

III EXPANDING VIRTUAL RELATIONS AND ORDERING ACCESSES TO RELATIONS

In the previous subsection, we assumed that SODA always used the relations specified by IN-expressions to retrieve the requested fields. For instance, if the variable S is introduced in the expression (IN S SHIP), then any subsequent reference such as (S NAM) would be interpreted as indicating the NAM field in the SHIP relation.

In fact SODA is more flexible than this. The relations specified in the initial SODA query are interpreted as virtual relations that may refer to fields stored on several actual data base relations. In SODA, there is one virtual relation for each type of object that we want to talk about (i.e. allow as the value of a variable), and for each type of object there is a schema that indicates the semantically meaningful ways of linking fields in different relations. (In the current implementation, the same schema is used for all virtual relations. This is an artifact of the particular data base being used, and would not be possible in general.) For instance, the schema for the virtual SHIP relation would specify that when talking about a ship, if we mention a field in the data base SHIP relation (e.g. NAM) and another field in the SHIPCLASS relation (e.g. DRAFT), then the way to link them is to join the two relations via the CLASS field.

SODA uses this information to transform the references to virtual relations in the initial query into references to actual data base relations. It does this by scanning the query for all the fields mentioned in connection with each variable introduced by an IN expression. It then uses the schema for the virtual relation that the variable ranges over to find the smallest set of data base relations that include all the fields and to specify the links between these relations. SODA then replaces the original IN expression that mentions the virtual relation with a series of IN expressions that mention the selected data base relations and specify the joins between them. The references to the fields in the virtual relation are replaced by the corresponding references to fields in the data base relations. For example, if we wanted to retrieve the name and draft of all the ships in the data base, the initial query would be:

```
((IN S SHIP)
 (? (S NAM))
 (? (S DRAFT)))
```

Since the NAM field occurs only in the data base SHIP relation and since the DRAFT field occurs only in the SHIPCLASS relation, both relations must be accessed. SODA therefore transforms this query into:

```
((IN S SHIP)
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

In this expanded query, SHIP and SHIPCLASS are interpreted as being actual data base relations, whereas in the initial query SHIP was interpreted as a virtual relation.

In expanding references to virtual relations, SODA must choose which relation to use to retrieve a particular field if that field is available from more than one relation. In our sample data base the type and length of a ship can be retrieved either from the SHIP relation or the SHIPCLASS relation. To solve this problem, SODA uses heuristic techniques developed for IDA to attempt to minimize the number of relations accessed. For more information on how this is done, see [3].

Another problem for SODA is to choose the order in which to access the relations mentioned in a query. We could interpret a SODA query as specifying a particular procedure by making a fixed processing strategy (such as strictly sequential processing) part of the definition of the language. The user would then be responsible for determining the order in which relations are accessed by choosing the order in which they are mentioned. Since SODA's main purpose is to be the target language for a natural-language processor that makes no attempt to order the queries it generates for efficiency, we use a few simple heuristics to reorder the initial query. First, restrictions are applied as soon as all the relations that they mention have been accessed, since this cuts down the amount of data that must be processed in the rest of the query. Next, any maximization, minimization, or counting expressions that can be processed are taken, since these expand the amount of data only slightly. After these expressions, IN expressions which can immediately be restricted are preferred over IN expressions which cannot. These heuristics are all intended to help choose the most restricted parts of the query first in hopes of minimizing the size of intermediate results.

IV PROBLEMS IN DISTRIBUTED PROCESSING OF COMPLEX QUERIES

If all the relations mentioned in an expanded reordered SODA query are stored at one data base site, then all that remains to be done is to translate the query into the query language of the DBMS at that site and execute the query. If, however, the data is distributed over two or more sites, some strategy must be devised for combining information from several locations.

What type of strategy is used will depend on assumptions about the relative efficiency of various operations. Since our data base is distributed over several sites on the ARPANET, a relatively low-speed communications channel, we have assumed that query processing will be most efficient if as much work as possible is done at a single site, and the amount of data transmitted between sites is kept to a minimum. (If transferring data between sites were fast compared to query processing at one site, the best strategy might be to spread the data over as many sites as possible to take advantage of concurrent processing.)

Given these assumptions, there seem to be two simple approaches that might be followed. One approach is to move all the relevant data to a single data base site and execute the query in one access to that site. We will call this the centralized approach. An efficient implementation of this approach would involve doing any local processing that would reduce the amount of data transmitted, such as taking projections, restrictions, or joins of relations, before sending the data to the primary site.

The other approach, which we will call the incremental approach, is to decompose the query into a series of simpler queries, each of which can be executed at a single data base site. Then each query is executed in turn at the corresponding site, and the results are transferred to the site where the next query is to be processed and combined with the information there. An efficient implementation of this approach would attempt to order the execution of the queries so as to minimize the total amount of data transmitted.

These two approaches do not exhaust the range of possibilities, of course. In fact, from a slightly more general point of view, they can be seen to be the two extremes of a spectrum. Since the final answer to a query will be generated at only one of the data base sites, we can view the problem of distributed query processing as how to organize the data base sites as a "data-flow tree," with information being transmitted up the branches towards the root, where the final answer is generated. From this point of view, the centralized approach limits its attention to the maximally branching, minimally deep trees, and the incremental approach limits its attention to the minimally branching, maximally deep trees. The most efficient organization may well be found in one of the intermediate possibilities, but we only consider these two approaches, as they are the easiest to implement.

If used intelligently, the incremental approach is often much more efficient than the centralized approach. The reason for this is not hard to see. Using the incremental approach, if we begin processing with a partial query that is highly restricted, that restriction will be "inherited" by all the subsequent partial queries that are processed, since at every stage we combine everything we have done so far before transferring the data to the next site. In the centralized approach, however, any processing that is done before transferring data is done independently of processing at other sites, so there is no way to take advantage of restrictions that may have been computed elsewhere.

For instance, in our sample data base, suppose that the PORT relation and the SHIPCLASS relation are stored at site 1 and the SHIP relation is stored at site 2. If we wanted to know the name and draft of all the ships currently in American ports, we would have to access all three relations and, therefore, both data base sites. The expanded SODA query for this request would be:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (IN S SHIP ((S POS) EQ (P PPOS)))
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

The most natural way of processing this query using the incremental approach would be to retrieve the locations of American ports from site 1, transfer this information to site 2 to find the names and classes of the ships at these locations, and then transfer that information back to site 1 to find the drafts of the ships and return the answers. Presumably, the amount of data transferred during this process would be significantly smaller than the amount that would be transferred either by moving the required fields of the SHIP relation to site 1 or by moving the required fields of the SHIPCLASS and PORT relations to site 2, as would be required by the centralized approach.

Examples such as this suggest that the incremental approach is to be generally preferred to the centralized approach. However, in complex, quantified queries, which are the major concern of our work on SODA, the incremental approach may be impossible to apply. This fact seems not to have been generally recognized in the literature on distributed query processing (e.g., [6]), where joining is typically the only method considered for combining data from two or more relations.

The problem of distributed query processing is considerably simplified by considering only joins for two reasons: First, joins specified over more than two relations can always be decomposed into a series of binary joins. Thus, if some of the relations to be joined are at one site and some are at another site, the relations at the same site can be processed first, and the intermediate results can be combined later. In the previous example, the query specified a join over the PORT, SHIP, and SHIPCLASS relations. In processing, this was decomposed into a join over the PORT and SHIP relations, and a join between the result of this operation and the SHIPCLASS relation.

The second simplification that joining permits is that, since the join operation is associative, it doesn't matter logically how the decomposition is done. Therefore, the decomposition can be chosen to suit the way the data is distributed. In our example we first joined the PORT relation to the SHIP relation and then joined the result of that operation to the SHIPCLASS relation. If the distribution of the data or the expected sizes of intermediate results had been different, however, it might have been more efficient to join the SHIP and SHIPCLASS relations first, and then add in the PORT relation.

In complex, quantified queries, on the other hand, the possible ways of decomposing queries are much more restricted. It is often impossible to break up queries to match the distribution of the relations, and in some cases, queries over several relations cannot be decomposed at all.

This point can be illustrated by changing our previous example slightly. Consider the same query, finding the name and draft of all ships in American ports, but with the PORT and SHIP relations at site 1 and the SHIPCLASS relation at site 2. In this case, it is probably most efficient to find the ships that are in American ports by joining the PORT relation and SHIP relation at site 1 and transfer the result to site 2 to join it with the SHIPCLASS relation to form the final answer.

Now let us alter the query so that it includes a universal quantifier, but still refers to the same relations in the same order, e.g., "Which American ports contain only ships which have draft greater than 50 feet?":

```
((IN P PORT ((P PNAT) EQ 'US'))
 (ALL (IN S SHIP ((S POS) EQ (P PPOS)))
      (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
      ((C DRAFT) GT 50))
 (? (P PDEP)))
```

The logical structure of this SODA query can be indicated by paraphrasing it back into English as follows:

```
For each tuple P in the PORT relation
          with (P PNAT) equal to 'US'
     such that, for all tuples S in the SHIP relation
                    with (S POS) equal to (P PPOS)
          and all tuples C in the SHIPCLASS relation
                    with (C CLASS) equal to (S CLASS),
          (C DRAFT) is greater than 50,
return (P PDEP).
```

Since the PORT and SHIP relations are both stored at site 1, and since there is a link specified between them, ((S POS) EQ (P PPOS)), we would like to decompose the query by first operating on these two relations and transferring the intermediate result to site 2 for processing with the SHIPCLASS relation. Unfortunately, the universal quantifier ALL does not permit any such decomposition. There is no way to combine the data referred to outside of the ALL expression with only part of the data referred to inside. We would need a distribution principle analogous to A*(B+C) = (A*B)+(A*C) to distribute the universal quantifier over the relations mentioned inside of the ALL expression, but no such principle exists.

Any query decomposition that is performed must respect the scope of quantifiers. We can independently process a portion of a query that lies entirely within the scope of a quantifier or entirely outside the scope of a quantifier, but we cannot independently process a portion of a query that splits the scope of a quantifier.

By nesting quantifiers more deeply, it is possible to construct queries over several relations that cannot be decomposed at all. Suppose we wanted to know the ship classes for which every American port contains some ship in that ship class. This could be represented in SODA as:

```
((IN C SHIPCLASS)
 (ALL (IN P PORT ((P PNAT) EQ 'US'))
      (SOME (IN S SHIP ((S POS) EQ (P PPOS))
                       ((S CLASS) EQ (C CLASS)))))
 (? (C CLASS)))
```

The English paraphrase of this SODA query would be:

```
For each tuple C in the SHIPCLASS relation
    such that, for all tuples P in the PORT relation
              with (P PNAT) equal to 'US',
         there is some tuple S in the SHIP relation
                        with (S POS) equal to (P PPOS)
                        and (S CLASS) equal to (C CLASS),
return (C CLASS).
```

This query cannot be decomposed. We cannot combine the data from the SHIPCLASS relation with the data from either the SHIP relation alone or the PORT relation alone, because this would cut across the scope of a quantifier. For the same reason, the SHIP relation and PORT relation cannot be combined without processing the whole SOME restriction. But this cannot be done independently of the SHIPCLASS relation, because the SOME restriction refers to the data from the SHIPCLASS relation via the term (C CLASS). Answering this query, therefore, requires simultaneous access to three relations.

Even though in queries such as these we cannot always combine relations locally before transferring data, we still can use projections and restrictions to cut down the amount of data that must be transferred. It turns out that in some cases we can add logically redundant restrictions that have this effect, although this is not done in the current implementation. Recall the previous query, "Which American ports contain only ships which have draft greater than 50 feet?" We could add a redundant restriction without changing the answer to the query and get "Which American ports contain only ships which are in some American port and have draft greater than 50 feet?" The SODA representation of the modified query would be:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (ALL (IN S SHIP ((S POS) EQ (P PPOS))
                (SOME (IN P1 PORT)
                      ((P1 PNAT) EQ 'US')
                      ((P1 PPOS) EQ (S POS))))
      (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
      ((C DRAFT) GT 50))
 (? (P PDEP)))
```

We still cannot independently combine the data generated by the expression (IN P PORT...) with the data generated by (IN S SHIP...), but if the PORT and SHIP relations are at the same site, we can compute the restrictions on (IN S SHIP...), including the restriction to ships in American ports using (IN P1 PORT...). So, although this restriction is logically unnecessary, it permits us to transfer much less data than would be required without it.

V DISTRIBUTED QUERY PROCESSING IN SODA

As the previous section indicated, complex queries do not always permit decomposition into sequences of simpler queries that match the distribution pattern of the data base. As a result, we have chosen to base the initial implementation of SODA on the centralized approach to distributed query processing. In doing so, we have traded the efficiency of the incremental approach in handling simpler queries for the generality of an approach that handles the more complicated queries which are our primary interest. A more sophisticated implementation could employ a mixed strategy, using the incremental approach when it is applicable and falling back on the centralized approach when it is not. Also, we have not implemented the type of query transformation discussed in the preceding subsection, since further research is needed to determine what the scope and limits of such techniques are.

In processing a query, SODA must first decide which data base site to use as the primary site for executing the query. A set of reasonable candidates is selected by starting with a list of all the sites that contain at least one of the relations mentioned in the query. Then redundant sites are eliminated until no site remaining in the list has the property that some other site in the list contains all the relations mentioned in the query that are contained by that site. Processing of the query is then simulated, trying each of the remaining sites as the primary site. The choice of primary site that appears to result in the least amount of data being transferred is selected to be the primary site for actually processing the query. This measure is currently crudely estimated by choosing the site that results in the fewest unrestricted queries requesting data from a secondary site. If this leaves more than one possibility, then one of those that results in the fewest restricted queries is chosen. A query is considered to be restricted if there is a restriction on any of the relations mentioned in the query.

Once the primary data base site has been chosen, SODA reformulates the query for execution at that site. The query is examined, one expression at a time. Expressions which refer only to data that is already at the primary site are left unchanged. If an expression refers to data that is not stored at the primary site, then this data is transferred to a temporary relation at the primary site, and the query is reformulated to refer to this relation. To take an example from Section IV, recall that the SODA query for retrieving the name and draft of all ships in American ports is:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (IN S SHIP ((S POS) EQ (P PPOS)))
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

If the PORT relation is stored at site 1 and the SHIP and SHIPCLASS relations are stored at site 2, site 2 will be chosen as the primary site for execution of the query, as this results in only a single restricted query being executed at a secondary site. Since the PORT relation is not stored at site 2, SODA first obtains the information needed from the PORT relation by dispatching the query:

```
((IN P PORT ((P PNAT) EQ 'US'))
 (? (P PPOS)))
```

to site 1 and stores the result in a temporary relation at site 2, say in field FIELD1 of relation TEMP1. The transferred data is constrained as much as possible by applying the restriction ((P PNAT) EQ 'US')) before the transfer, and only the fields required by the rest of the query are moved, in this case, just the PPOS field. The main query is now reformulated as:

```
((IN T TEMP1)
 (IN S SHIP ((S POS) EQ (T FIELD1)))
 (IN C SHIPCLASS ((C CLASS) EQ (S CLASS)))
 (? (S NAM))
 (? (C DRAFT)))
```

Since the query now refers only to relations stored at site 2, it can be executed in a single access to that site.

The process just described is complicated somewhat by a set of issues involving redundantly stored data and error recovery. One of the principal advantages of a distributed data base is that the system can be made more reliable by storing data redundantly at several data base sites. If this is done, then the system can tolerate failure of one or more data base sites and still be able to answer all the queries covered by the data base (although not always with the most recent information).

In SODA, therefore, we take into account the possibility that a given relation may be stored at more than one data base site, and we use this fact to try to recover from data base failures. Because of our centralized approach to query processing, we distinguish between failure of the primary site and failure of one of the secondary sites. Since all intermediate results are stored at the primary site, a failure there requires complete replanning and re-execution of the query. If a secondary site fails, however, SODA backs up only as far as the beginning of the portion of the query involved in the current access to that site and begins replanning from that point. This preserves any intermediate results that have actually been extracted from secondary sites and thus avoids unnecessary recomputation.

A more difficult question for SODA is at what site to access a particular relation, when more than one possibility is available. Solutions to this problem are also constrained by our use of a centralized approach for query processing, since, in general, we want to access relations at the primary site if possible. There are exceptions to this rule, however. In particular, if we must transfer information from a secondary site, it may be more efficient to go ahead and combine that information with data from another relation at the secondary site, even though the other relation may also be stored at the primary site.

SODA uses a number of simple heuristics to decide whether to access a relation at the primary site or a secondary site. Roughly, SODA will prefer a secondary site if the relation is joined to another relation which must be accessed at the secondary site, and if performing the join appears likely to cut down the amount of data retrieved from that relation. A join is assumed to cut down the amount of data retrieved if it puts more restrictions on the data.

For instance, if we wanted to know about American ships with a draft of more than 50 feet, we would have to access the NAT field in the SHIP relation and the DRAFT field in the SHIPCLASS relation. Suppose the SHIP relation is stored only at a secondary site and the SHIPCLASS relation is stored both at that site and the primary site. In this case, we would access the SHIPCLASS relation at the secondary site because it would further restrict the set of ships for which data must be transferred to the primary site. If, on the other hand, we simply wanted to retrieve the drafts of American ships, we would access the SHIPCLASS relation at the primary site, since this would not further restrict the data being transferred.

These heuristics are rather crude, since they do not take into account the relative sizes of relations, how constraining a particular restriction is, or the functionality of joins between relations (e.g., many-to-one, one-to-many). There is clearly a trade-off, however, between time spent in access planning and time spent in query execution, and it is not clear how much more effort could be put into access planning that would justify itself in more efficient query execution.

Compared to the SDD1 distributed DBMS [6] [7] [8], the techniques used in SODA have both advantages and disadvantages. SDD1 takes what is essentially a centralized approach to query processing, but not as completely as SODA. The main difference is that for purposes of assembling all the relevant data at a single site, SDD1 treats the query as if it contained only joins and restrictions. If a query specifies a more complex way of combining relations than joining, SDD1 will find a join that "covers" that portion of the query, in the sense that the data it retreives includes as a subset all data required to answer the query. However, it does not perform a precise logical analysis, as SODA does, to retrieve exactly the required data. Because

SDD1 takes this simpler view, it is able to use more sophisticated heuristics for combining partial results from several secondary sites before transferring them to the primary site. However, since the partial results are only approximate, the entire query must be re-executed at the primary site.

One clear advantage that SDD1 has over SODA is that SDD1 maintains statistical information about the size of relations and the distribution of values of fields. This enables SDD1 to predict more accurately than SODA the size of intermediate results, and hence do a better job of query optimization. It should be noted, however, that SODA is designed to permit use of such information without any changes to the basic structure of the system.

One final difference between SDD1 and SODA is that, although SDD1 permits arbitrarily redundant data bases, a particular query is answered only with respect to a single nonredundant mapping of the data base. Because SODA can decide at processing time where to access a redundantly stored relation, it is possible to answer some queries more efficiently and to recover from the failure of a secondary data base site without completely reprocessing a query.

VI LIMITATIONS AND POSSIBLE EXTENSIONS OF SODA

Like any real system which addresses a complex problem, SODA offers only partial solutions to the issues it raises. There are several areas where significant improvements or extensions could be made. One of these areas is the expressive power of the query language. Although SODA is a richer language than IDA and many other data base query languages, there are still useful queries that it cannot express.

One of the constructs that SODA lacks is some kind of conditional expression. For example, it might be desirable not to store the current position of ships that belong to task forces individually in the SHIP relation, but rather to have this information derived by looking up the location of the task force to which a ship belongs in a TASKFORCE relation. This would make it possible to update the location of the entire task force at once, rather than ship by ship. If we do this, however, retrieving the position of a ship becomes a conditional procedure, depending on whether the ship belongs to a task force. To retrieve the position of a particular ship, such as the Fox, we would have to be able to express in SODA the following query, which we currently cannot handle:

```
For the tuple S in the SHIP relation
        with (S NAM) equal to 'FOX',
    if (S TFNAM) has the undefined value
        then return (S POS),
    otherwise, for the tuple T in the TASKFORCE relation
            with (T TFNAM) equal to (S TFNAM),
        return (T TFPOS).
```

Another class of queries that cannot be expressed in SODA is queries that involve following chains of indefinite length through the data base. For instance, in the personnel data bases that are commonly used to illustrate concepts of data base access, a classic problem is to answer the query, "Which employees earn more than their managers?" Many of the simpler query languages that have been proposed, including IDA, cannot represent such a question, although for SODA this would be no problem.

However, if we want to define the relationship "superior of" to be the transitive closure of "manager of" (i.e., the manager of the manager, etc.), we are in trouble. There does not seem to be any non-procedural query language, including SODA, that could express queries such as, "Which employees earn more than all/some of their superiors?" The problem is that expressing this type of query asks a question about all chains through the data base of a certain kind, whereas existing query languages only allow asking about all tuples of a certain kind.

Another general area where SODA could be improved is query optimization and access planning. The heuristics used to pick the order in which relations are accessed are quite crude, taking into account only which references to relations are restricted. As we pointed out in discussing the heuristics for distributed query processing, it would also be useful to consider the relative sizes of relations, how constraining a restriction is, and the functionality of joins between relations.

It will never be possible to guarantee that a query will be processed in the optimum way, however. First of all, to do so would require knowing the size of all of the possible intermediate results that might be generated in processing the query, and in general the only way to get this information is to execute the query. Second, even if we had good enough estimates for all of the relevant factors, choosing the most efficient way to process a query would still be a combinatorial search, which might take longer to perform than executing the query with the few simple heuristics we currently have. So any technique for query optimization must be empirically tested to see whether the savings it produces are worth the cost of applying it.

Finally, some of the improvements planned for SODA concern pragmatic problems in dealing with interactive users. One of these problems is that if there is no information in the data base satisfying a complex query, the system simply returns a null result with no further explanation. Often it would be much more helpful to the user if the system would provide some indication of why it failed to find an answer. For instance, if we ask the system to compute the distance between the Fox and the Kennedy and get no answer, it might indicate that the Fox is not listed in the data base, or the position of the Fox is not given in the data base, or the Kennedy is not listed in the data base, or the position of the Kennedy is not given in the data base, or any combination of the above. We are currently investigating how this information might be obtained from the data base and supplied to the user in a form he can understand. For more discussion of problems of this kind, see Kaplan [5].

The other pragmatic problem we are looking at is how to save and make use of previously retrieved information to avoid recomputing it. For example, if we ask the LADDER system in English, "Which American ships are in the Atlantic?" followed by the question "What are their fuel states?" the pronoun "their" will be correctly resolved to the phrase "American ships in the Atlantic" by the natural-language front end, but this set of ships will be recomputed by the data base access component. Although the natural-language processor realizes that the two queries are related, SODA does not. We are examining various issues that arise in dealing with this problem, including what information to save, how long to save it, and where it should be stored.

Appendix A

FORMAL DEFINITION OF THE SODA QUERY LANGUAGE

Appendix A

FORMAL DEFINITION OF THE SODA QUERY LANGUAGE

The syntax of the SODA query language is most easily described by a context-free grammar, plus some constraints on the occurrence of variables. The grammar for SODA is extremely simple, having only the following five rules:

```
query -> ([binder ¦ restriction ¦ (? term)]+)

subquery -> [binder ¦ restriction]* binder [binder ¦ restriction]*

binder -> (IN tuplevar relation [restriction]* ) ¦
          (MAX term subquery) ¦
          (MIN term subquery) ¦
          (MAX1 term subquery) ¦
          (MIN1 term subquery) ¦
          (COUNT countvar subquery)

restriction -> (ALL subquery) ¦
               (SOME subquery) ¦
               (NONE subquery) ¦
               (AND [restriction]+) ¦
               (OR [restriction]+) ¦
               (NOT restriction) ¦
               (term comparison term)

term -> (function ([term]+)) ¦
        (tuplevar field) ¦
        countvar ¦
        constant
```

In this grammar, nonterminal symbols are written in lower case, and terminal symbols are written in upper case. To make the grammar more concise, we have allowed the right side of a rule to be written as a regular expression. The notation "...¦..." indicates an alternative, the notation "[...]*" indicates a sequence of any length greater than or equal to zero, and the notation "[...]+" indicates a sequence of any length greater than zero. Note that the parentheses appearing in the grammar are part of the SODA language which is being defined, while the square brackets are part of the notation in which the grammar is written.

SODA queries are composed of three principal types of expressions: binders, which bind variables to refer to data extracted from the data base, restrictions, which restrict the data, and question-mark expressions (selectors), which request retrieval of parts of the data. A SODA query is any nonempty, parenthesized sequence of these expressions that satisfies the constraints on the occurrence of variables which will be discussed shortly. If the sequence of expressions includes one or more binders, then the query implicitly defines a set of tuples, and the selectors in the sequence specify a projection of that set which is to be returned as the answer to the query. If there are no selectors in the query, then it is interpreted as a yes/no question, asking whether the set defined by the query is nonempty. If the query is simply a sequence of restrictions, then it is interpreted as a yes/no question asking whether all of the expressions in the sequence are true.

The structure of the language is recursive, with MAX, MIN, MAX1, MIN1, COUNT, ALL, SOME, and NONE being operations on certain sequences of expressions which would themselves be well-formed queries. We will call such sequences subqueries, and they must meet the following conditions: first, since it only makes sense to return information from the top level, no selectors are allowed in the sequence. Furthermore, we insist that there be at least one binder in the sequence, since there must be some data from the data base to maximize, minimize, count, or quantify over.

The binders include IN expressions, COUNT expressions, and MAX, MAX1, MIN, and MIN1 expressions. An IN expression sets a variable to range over the set of tuples in a relation (or a restricted subset, if any restrictions are specified). A COUNT expression sets a variable to the number of tuples in the set defined by a subquery. MAX and MIN expressions pick out all the tuples in a set for which some term has the largest or smallest value. MAX1 and MIN1 expressions do the same, except that they pick out a single tuple from this set. MAX1 and MIN1 can be executed more efficiently than MAX and MIN, so they are to be preferred when applicable, such as when it is known on semantic grounds that there can be only one tuple in the set of interest that has the maximum or minimum value (e.g., there can be only one most recent position report for a ship).

Restrictions include simple Boolean restrictions with AND, OR, and NOT, plus universally quantified restrictions using ALL and existentially quantified restrictions using SOME. [(NONE...) is an abbreviation for (NOT (SOME...))]. The details of how these constructs are interpreted are explained in the discussion of the examples in Section IV.B.

The grammar does not specify what the relations, fields, functions, comparisons, constants or variables are. Constants include numbers and any character strings enclosed in single quotes, such as 'US'. Any other alphanumeric character string can be used as a variable. There are two types of variables: tuplevars, which range over the tuples of a given relation, and countvars, which are used to refer to the result of a counting operation. There need not be any difference in form between tuplevars and countvars, but no symbol can be used as both within the same query.

The other categories not specified by the grammar are all implementation-dependent. Fields and relations obviously depend on the particular data base being accessed. The functions and comparisons depend on the capabilities of the underlying DBMSs in which the queries are actually executed. In the current implementation of SODA, there are four navigation functions (e.g., GCDIST, for computing the great circle distance between two geographical locations), and the comparison operators are EQ, NE, LE, GE, LT, and GT, representing "equal," "not equal," "less than or equal," "greater than or equal," "less than," and "greater than or equal," respectively.

To explain the constraints on the occurrence of variables, we need to define several notions. A variable which is the second element of an IN or a COUNT expression is said to be introduced by that expression. The smallest COUNT, SOME, NONE, or ALL expression that includes the expression that introduces the variable is said to be the scope of the variable. If the expression that introduces the variable is not inside any COUNT, SOME, NONE, or ALL expression then the scope of the variable is the entire query. An occurrence of a variable is bound by the expression which introduces it if the occurrence is contained by every MAX, MAX1, MIN, or MIN1 expression that contains the expression which introduces the variable; otherwise, the occurence is bound by the largest MAX, MAX1, MIN, or MIN1 expression which does not contain the occurrence but does contain the expression which introduces the variable. We can now state the contraints on the occurrence of variables as follows:

(1) No variable may occur in the query unless it is introduced by some expression in the query.

(2) No variable may be introduced by more than one expression.

(3) No variable may occur outside its scope.

(4) The relation "X contains an occurrence of a variable bound by Y" must not form any circular chains of MAX, MAX1, MIN, or MIN1 expressions.

The first rule ensures that the range and scope of every variable is defined. The second rule simply means that the same variable can't be used in two different ways in the same query. This is actually slightly stronger than it needs to be, since two variables which have nonintersecting scopes could be the same without creating logical confusion, but queries are simpler to process and easier to understand if this is not done.

The third rule prevents using a variable in a context where the reference doesn't make sense semantically. It is easiest to think of a variable as referring to a particular tuple in a set of tuples. Inside a COUNT, SOME, NONE, or ALL expression, the variable refers to each tuple in the set in turn, as in "for all tuples in the SHIP relation such that the tuple..." Outside the expression, however, there is no way to determine which tuple is being referred to. This contrasts with MAX (and MIN) expressions, where, although the variable refers to each tuple in turn inside the expression, there is a definite referent for the variable outside the expression, namely, the tuples for which the term being maximized has the greatest value.

The final rule forces the definitions of sets that are being maximized or minimized over to be noncircular. One MAX or MIN operation can refer to the result of another, but only if the second is well defined without referring to the first.

To put the SODA query language in perspective, we can compare it to Codd's original language based on the relational calculus, DSL ALPHA [9] [10]. One major difference between the two languages is that SODA is only a data retrieval language, whereas ALPHA also permits updating the data base. In their power to express queries, the two languages are fairly close. ALPHA has the ability to request retrieval in a specified order or to set a limit on the number of tuples to be used in computing the answer. These features were left out of SODA because they do not have a natural interpretation in purely set-theoretic terms. On the other hand, SODA has more powerful counting, maximizing, and minimizing operators. In SODA, these can operate on sets of tuples defined by arbitrary subqueries; in ALPHA, they are much more restricted.

There are important differences in the syntax of the languages as well. SODA is, in fact, a data sublanguage of LISP, and thus it shares LISP's highly parenthesized syntax. While in some ways this makes SODA queries more difficult for people to read, it greatly facilitates the generation of SODA queries by other programs, a primary requirement for use in the LADDER system.

Finally, the syntax of SODA has been designed to facilitate translation of natural-language questions into formal queries. This has resulted in departures from typical relational-calculus syntax, particularly in quantifier expressions. In most "English-like" formal languages, English words are simply tacked onto a semantic structure that bears little relation to English. In SODA, the correspondence of the symbols of the language to English words is of minor importance and is basically just a mnemonic device. What is important is that the semantics of several of the constructs of the language have been designed to correspond to the semantics of certain types of English phrases. For an illustration of this point see the discussion of quantifier expressions at the end of Section II.

REFERENCES

1. G. G. Hendrix, et al., "Developing a Natural Language Interface to Complex Data," ACM Transactions on Database Systems, Vol.3, No. 2, pp. 105-147 (June 1978).

2. E. D. Sacerdoti, "Language Access to Distributed Data with Error Recovery," Proceedings of the Fifth International Joint Conference on Artificial Intelligence, pp. 196-202, Cambridge, Massachusetts (August 1977).

3. D. Sagalowicz, "IDA: An Intelligent Data Access Program," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan (October 1977).

4. Naval Electronics Laboratory Center, "The Relational Model for the Blue File Data Base (Revised)," Project Scientist: Garrison Brown; San Diego, California (November 1976).

5. S. J. Kaplan, "Cooperative Responses from a Natural Language Data Base Query System: Preliminary Report," Technical Report, Deptartment of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania (November 1977).

6. E. Wong, "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," Technical Report CCA-77-03, Computer Corporation of America, Cambridge, Massachusetts (March 1977).

7. "A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 1," Technical Report CCA-77-06, Computer Corporation of America, Cambridge, Massachusetts (July 1977).

8. "A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 2," Technical Report CCA-78-03, Computer Corporation of America, Cambridge, Massachusetts (January 1978).

9. E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, available from ACM.

10. C. J. Date, An Introduction to Data Base Systems, pp. 63-82 (Addison-Wesley Publishing Company, Reading, Massachusetts, 1975).