Performance enhancement for relational systems through query compilation*

by RANDY H. KATZ

University of California, Berkeley
Berkeley, California

INTRODUCTION

In recent years, considerable research has been directed toward the relational model of data first proposed in Reference 5. The advantages of this approach have been discussed elsewhere, however they can be summarized as follows: (1) the user is presented with a very simple view of his data (i.e., organized as tables of information), (2) the user is freed from explicitly knowing the underlying implementation structure of his data (i.e., data independence) and (3) very powerful non-procedural, set-oriented query languages can be defined for the relational model because of its simple conceptual structure.

While commercial versions of data base systems based upon hierarchical\(^8\) and network\(^11\) models have existed for several years and are used widely,\(^9,8\) at this time there exist few commercially available relational systems. Two relational prototype systems currently in operation are the INGRES system\(^1\) and System-R.\(^3\)

A major criticism of the relational approach is that the model does not lend itself to efficient processing because there is no provision for the user to explicitly "navigate" through access paths in his data. Until recently, the experimental systems previously mentioned were not overly concerned with efficiency. However, efficiency has now become an important goal.

Hence, from direction of relational data base efficiency, motivation is provided for compilation of data base queries. The goal is to move as much query processing to compile-time as possible, thereby reducing the overhead of query execution at run-time. However, there are several problems. Some query processing algorithms are data-dependent. For example, the decomposition algorithm of Reference 19 makes use of the cardinalities of relations, rather than statistical information, in selecting among alternative plans to implement a query. Such algorithms may actually suffer in performance if the processing strategy is determined at compile-time rather than with the more perfect information available at run-time. Further, any change in the data base schema will invalidate some compiled queries. These include changes to the authorization rights of a given user, changes to views of the data base one is allowed to see and changes to the physical structure of the data. Note that in many data base applications, the structure of the data is relatively static over time. Compilation can be most effective in these cases. Even given these problems, compilation techniques should improve relational system performance.

The goal of this paper is to analyze the problems of query compilation in the specific environment of the INGRES data base system. A proposal was implemented and its space and time efficiency is analyzed.

PREVIOUS WORK

In the early work on relational systems, high level non-procedural query languages were provided to process ad-hoc queries presented interactively by non-programmers.\(^5,6\) Current work is directed towards applications-oriented languages for programmers, which integrate relational query capabilities into programming languages. These languages are meant to complement conversational query languages. In this situation, it is possible to take advantage of the more static non-interactive environment to improve the efficiency of query processing.

In order to classify and to compare the work which will be summarized, it is useful to define "levels of compilation." Query processing consists of the following steps:

1. The query string is analyzed (i.e., parsing).
2. Object names are converted to an internal form via System Catalogs (i.e., lookup).
3. A processing plan is built.
4. The processing plan is executed by calls to the access methods.

Using this, five levels of compilation can be defined:

0—The query is completely interpreted at run-time (i.e., Steps 1-4 are performed at run-time).
I—The internal form of the query is built at compile-time (i.e., the query is converted from a string to a parse tree at compile-time).

* Research supported by the Army Research Office under Grant DAAG29-76-G-0245.
2—The internal form is built at compile-time, with name resolution taking place if possible. Otherwise, Step 2 will have to be executed when the information is available at run-time.

3—The internal processing plan is determined at compile-time and translated into an internal form suitable for run-time interpretation.

4—The internal processing plan is represented within the user program by calls to the underlying access methods.

Level 0 represents the least amount work done at compile-time, namely complete interpretive execution of the query at run-time. Level 4, on the other hand, represents the movement of the most amount of processing to compile-time.

In the remainder of this section, we describe several existing or proposed data base programming languages and describe what level of compilation they support or propose to support.

ASTRA

ASTRA is a relational system built on a hierarchical system at the University of Trondheim, in Norway. It supports a high-level applications-oriented language called ASTRAL (A Structured Relational Application Language), based upon SIMULA. The compiler maps an ASTRAL program into a SIMULA program augmented by subroutine calls to the underlying data base system. It is the job of the compiler to choose the best access paths when translating relational expressions into sequences of host language statements and subroutine calls. The functions of query parsing and of determining a query processing plan are moved to compile-time. The plan, represented by calls to the underlying system, is interpretively executed at run-time. Hence, the ASTRAL compiler supports Level 3 compilation. No mention is made of the recompilation problems associated with changes to the data base schema.

Extended PL/1

Reference 16 describes extensions to PL/1 which allow an applications programmer to access data from a relational data base. The language is extended by introducing the notion of a template, which is a PL/1 structure containing data base relation names and attribute names, along with a SELECT statement, that qualifies those tuples which are to be retrieved. In Reference 10 the problems of recompilation under a changing schema are explored. The authors point out that the performance advantages for a compilation approach are somewhat offset by the increased sensitivity of a compiled program to schema changes. They propose that the program be separated into three modules—one for applications-oriented processing, one for data base interactions and one for authorization and integrity enforcement. If there is a change in some aspect of the schema, only the relatively small module of code which is effected by that change will have to be recompiled. This approach has actually been used in a system to support Extended PL/1. An architecture similar to this proposal will be necessary for any system which attempts to provide Level 3 or Level 4 compilation, because the internal processing plan will become invalid if the schema changes. Note, however, that the proposed architecture implies many inter-module communications. This overhead may outweigh the advantages of compilation.

System-R

System-R is an interesting system from the standpoint of compilation because it originally supported only interpretive query execution, but now supports both compiled and interpretive queries. The origin of System-R can be found in SEQUEL-XRM, a single-user relational system designed to support the SEQUEL query language. The system interpretively mapped the non-procedural statements of the query language into tuple at a time commands to the underlying relational memory system, XRM (extended n-ary relational memory). The interpreter was organized into three modules—parser, optimizer and scanner. The parser translated a SEQUEL query into a corresponding parse tree. The optimizer, when given a query tree, selected a processing plan which would minimize the number of tuples retrieved based upon the available access paths. The scanner actually executed this plan by judiciously scanning the underlying relations.

Note that the parsing step is independent of whether the query is being interpreted or compiled. Furthermore, if the optimization step is independent of the actual data in the database then the optimized processing plan can be determined at compile-time as well. It then becomes a simple matter to compile the scanner into a query specific data access routine. This approach was taken in System-R. Because the data access routines consist essentially of access method calls, System-R supports Level 4 compilation.

All data access routines are separated from the user program to facilitate recompilation when the data base schema changes. A routine is marked invalid by the system if it depends on the schema change and is recompiled the next time it is called. The recompilation process is simplified if the query tree is saved along with the data access routine. This is because only the optimization and code generation phases need to be re-executed.

COMPILED FOR INGRES

In this section, a scheme for the compilation of QUEL, the data sub-language of the INGRES Data Base System, is presented. Complications due to the compilation approach are then examined.

The INGRES system consists of five concurrently executing processes:

```
+---+---+---+---+---+---+---+
|DBU|Decomp|OVPQ|Parser|User|PGM|
+---+---+---+---+---+---+
```

From the collection of the Computer History Museum (www.computerhistory.org)
These are the user program, the parser, the one-variable query processor, decomposition and the database utilities. The parser converts query strings into trees. OVQP interpretively executes one-variable queries, i.e., those involving a single relation. Decomps controls the decomposition of multivariable queries into a sequence of one-variable queries. The DBUs provide utility support. Communication between the processes is accomplished by "pipes," which are essentially message buffers.

Borrowing from System-R terminology, there are three phases in the lifetime of a program—preprocess-time, compile-time and run-time. Preprocess-time refers to the time when data manipulation statements of a program are first analyzed and translated. Compile-time refers to the time when the actual host language program, with associated data base calls, is compiled. Run-time refers to the time when the program is in execution. Compilation of data base queries is most advantageous if it can be performed at preprocess-time. However, flexibility considerations may make it necessary to defer compilation of the data base portion of a program until run-time, when complete information is specified.

It is possible to describe the various levels of compilation presented in the previous section in terms of the INGRES system:

Level 0—INGRES interpretively executes all queries. The QUEL preprocessor translates a user program into a sequence of host language statements and calls on the underlying data base system. These calls pass the queries in the form of character strings to INGRES for run-time execution.

Level 1—INGRES translates the query string into a tree at run-time. The structure of the query tree can always be determined from the string. Thus, it is possible to parse the query and build the tree at preprocess-time. If the data base, relation, and attribute names are known then information associated with those names can be placed into the tree. If we make the restriction that this information must be known at preprocess-time, Level 1 compilation can be achieved.

Level 2—If the restriction that all names must be known at preprocess-time is lifted, then it is necessary to fill in missing information at run-time when all names must become known. Thus Level 2 compilation can be accomplished. Note that the form of the tree is unaffected by the lack of information at preprocess-time. The information in the nodes of the tree, on the other hand, depends upon information associated with unresolved names. These names must be looked up in the system catalogs at run-time.

Level 3—INGRES constructs an execution plan by decomposing a complex query into a sequence of one-variable queries. This is accomplished by alternatively applying tuple substitution and reduction. Once a collection of decomposed query trees is available, it is possible to generate code at preprocess-time which will perform the decomposition at run-time. Because tuple substitution can not be performed until run-time, the generated code will have to be parameterized in order to allow actual values from the data base to be substituted. It is useful to think of the generated code as query procedures which can be invoked by other query procedures, with the lowest level of invocation representing parameterized one-variable queries. These queries can then be passed to the OVQP for interpretive execution. Note that the invocation of the compiler to build these query procedures will have to be postponed until run-time unless complete information is available at preprocess-time.

Level 4—Based upon the physical structure of the underlying relations, OVQP determines the efficient access paths for the processing of a one-variable query. Once the query procedures are constructed, it is possible to generate access method calls for the query procedures. Thus, Level 4 compilation is easily achieved once Level 3 compilation is operative.

At this point, we will give an example of Level 4 compilation. Consider the following QUEL query:

```
range of E, M is employee
range of D is department
retrieve (E.name) where E.salary > M.salary
   and E.manager = M.name
   and E.dept = D.dept
   and D.floor# = 1
   and E.age > 40
```

The query requests the names of all employees over 40 years old who make more than their managers and work in a department which is situated on the first floor. The following two queries can be detached from the original:

```
range of D is department
retrieve into T1 (D.dept) where D.floor# = 1
range of E is employee
retrieve into T2 (E.name, E.salary, E.manager, E.dept)
   where E.age > 40
```

The original query becomes:

```
range of D is T1
range of E is T2
range of M is employee
retrieve (E.name) where E.salary > M.salary
   and E.manager = M.name
   and E.dept = D.dept
```

The decomposition algorithm chooses a variable for substitution. If D is chosen, D.dept is replaced by actual values from the data base. A parameterized query of the above form is decomposed further, with the value of D.dept being the parameter. Continuing, the following query is detached:

```
range of E is T2
retrieve into T3 (E.name, E.salary, E.manager)
   where E.dept = value
```

The above query now becomes:

```
range of E is T3
range of M is employee
retrieve (E.name) where E.salary > M.salary
   and E.manager = M.name
```

From the collection of the Computer History Museum (www.computerhistory.org)
At this point, another tuple substitution is necessary. Suppose the variable $E$ is chosen. The resulting queries have one variable and can be executed directly by OVQP.

The compiled version of this query would look something like this:

$$Q(,)$$

- access method code for "retrieve into $T_1$ ($D$.dept)
  where $D$.floor# = $1$
- access method code for "retrieve into $T_2$ ($E$.name, $E$.salary, $E$.manager, $E$.dept) where $E$.age > $40"
  for each tuple in $T_1$, call $Q'(dept)$
- access method code for "retrieve into $T_3$ ($E$.name, $E$.salary, $E$.manager) where $E$.dept = $d"
  for each tuple in $T_3$, call $Q'(name, salary, manager)$
- $Q'(n,s,m)$:
  - access method code for "retrieve ($n$) where $s$ > $M$.salary and $m$ = $M$.name"

The compilation approach for INGRES just described causes several problems. Data bases are potentially dynamic objects. A system based upon compilation must be able to deal with changes to the logical and physical schema. Also, the architecture of the system must ensure security for user data.

Due to protection considerations, (e.g., only processes owned by INGRES are allowed to execute access method calls), the query portion of a program will have to reside in a separate process from the rest of the program. Requests for the execution of a query can be passed down the pipe from the user program to the query process, with results being returned in the opposite direction. The process structure would be as follows:

```
user | Exec Qt | query | process | DATA BASE
```

The five-process system has been replaced with a three-process system. Note, however, that the query process is unique to each program, whereas the processes of the current INGRES system can be shared among concurrent users.

Because the access method code is dependent upon the physical structure of the relations, the query process can become invalid if there is a change to the physical schema. A scheme similar to that used in System-R can be used to recompile those query processes that have become invalid. A system catalog, in the form of a relation, associates query processes with data base objects they depend on. A process can be marked invalid if an object it depends upon has changed. When a program uses an invalid query process, the compiler can be invoked to recompile the program. If the query trees are maintained in the data base, only the query process itself need be recompiled. Otherwise, the original program will have to be reprocessed for the regeneration of the query parse trees.

Because authorization and views are implemented by query modification, the techniques described above can be used to recompile a program if authorization privileges or views are changed. In this case however, the query tree will have to be reconstructed. To avoid reprocessing the user program, it may be useful to maintain the query string in the data base and to perform all four steps of query processing on this string.

**RESULTS**

Four versions of the INGRES system were configured for comparison. These were:

1. The standard five-process system, available through the EQUEL preprocessor (the EQUEL system).
2. A modified five-process "parse at compile-time" system, with the parser process copying pipe input to pipe output (the C-EQUEL (5) system).
3. The four-process 'parse at compile-time' system, available through the C-EQUEL preprocessor (the C-EQUEL (4) system)
4. A C program augmented with hand-coded access method calls (called AM code).

Three test programs were selected as benchmarks, called Test Programs 1, 2, and 3. The structure of the programs is essentially the same. Each program consists of a single retrieve statement which is executed one thousand times. The retrieve statement in Test Program 1 has no qualification, hence the entire relation must be scanned. Test Program 2 contains a relatively complex qualification. However, the entire relation must be scanned, because the qualification is true for every tuple. The only difference between Test Programs 1 and 2 is the complexity of the qualification. Similarly, Test Program 3 contains a relatively complex qualification for which no tuples of the relation will qualify.

The first comparison made between the configurations was in user program size. Table I summarizes the results. Basically, as the level of compilation increased (e.g. approached complete compilation), the user program size increased. This result can be misleading. The total size of an INGRES program is actually the sum of the sizes of the processes which make up the system. Table II contains the

<table>
<thead>
<tr>
<th>Compiler Used</th>
<th>Test Program 1</th>
<th>Test Program 2</th>
<th>Test Program 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQUEL</td>
<td>10694</td>
<td>10756</td>
<td>10758</td>
</tr>
<tr>
<td>C-EQUEL (5)</td>
<td>16180</td>
<td>16306</td>
<td>16306</td>
</tr>
<tr>
<td>C-EQUEL (4)</td>
<td>16192</td>
<td>16318</td>
<td>16318</td>
</tr>
<tr>
<td>AM Code</td>
<td>24400</td>
<td>24418</td>
<td>24418</td>
</tr>
</tbody>
</table>
relative sizes of the major INGRES processes. Using this measure of program size, the EQUEL programs required 224K bytes, the C-EQUEL programs required 181K bytes, and AM programs required 24K bytes.

INGRES processes are actually shared among concurrently executing INGRES programs, since the processes are re-entrant. Thus, the storage requirements for EQUEL and C-EQUEL programs should be distributed among the concurrent users of the system. This would require approximately ten concurrent users of INGRES for an EQUEL program to match the storage requirements of the AM program. The results indicate that for certain kinds of programs, the compilation approach may actually decrease the overall storage requirements for data base programs. This will depend upon the complexity of the program as well as the number of concurrent users. In any case, the storage requirements for compiled programs do not appear to be prohibitive.

The second comparison made was in speed of query processing. For each of the four systems, five different tests were run. These tests differed mainly on the test program used and the number of tuples scanned during the test. The first three tests consisted of the three test programs run on a relation with a single tuple. The remaining tests consisted of Test Programs 1 and 2 run with the same relation, but this time with 25 tuples. The timing results are listed in Table III.

For each experiment, the four versions of the same program were run 20 times while there was no activity on the system. Elapsed time was used for the measurements because it is relatively easy to measure and because it is a good indicator of combined CPU and I/O time when there is little other activity on the system. The mean and standard deviation of the readings were computed and are reproduced in Table III. For easy comparison, bar graphs of these results can be found in Figure 1. Because the standard deviations are typically quite small (i.e., less than one percent), the mean run-times are a good basis for comparing the benchmarks.

Table IV is a compilation of the percentage improvements between two given configurations. The improvement due to parsing at compile-time is presented in Line 1. The improvement due to eliminating the parser process is listed in Line 2. The total improvement of C-EQUEL over EQUEL is available as Line 3. The improvement of complete compilation over interpretation is recorded in Line 4.

The advantages of parsing at compile-time are obvious from Table IV. The more complex the query is, in terms of the time it takes to parse the query string, the greater the improvement possible with C-EQUEL. This is illustrated in the percentage improvement for Run 1 and Runs 2 and 3. However, because parsing represents a fixed overhead, as the total time to process the query increases, the improvement due to compile-time parsing decreases. This is evident from the improvements for Run 1 and Run 4.

The major difference between C-EQUEL (5) and C-EQUEL (4) is the inclusion of a dummy parser process that copies data from input to output. The improvement represents the benefits gained by reducing the size of the system by one process. As can be seen from the results of Runs 1, 2 and 3 and Runs 4 and 5, the overhead associated with the existence of an extra process is essentially fixed. As the
total processing time increases, the associated overhead becomes less significant, as shown in Runs 4 and 5.

The improvement of C-EQUEL over EQUEL is represented by the percentage improvement of C-EQUEL (4) over EQUEL. In the case of a complex query with little associated processing, the improvement is over one-third. In the case of simple queries which require much processing, however, this improvement is considerably less.

The most striking result is the improvement of compiled programs over interpreted ones. The result that compiled programs will run faster than interpreted ones is not surprising; the magnitude of this improvement is. It indicates that there is considerable overhead associated with a multiple process interpretive system that has little to do with performing the actual functions necessary for query processing.

CONCLUSIONS

The compilation approach to query processing has the potential of greatly reducing the overhead associated with execution of queries. An actual parse at compile-time system was implemented and was shown empirically to reduce query processing time, at a modest increase in storage requirements. This is only one possible point in a continuum of possible levels of compilation, each with its own time/space tradeoffs. Complete compilation was shown to produce substantial time savings, with possible space savings as well.

A general notion of compilation level was presented and it was shown how to describe an interpretation-based system such as INGRES in terms of different levels of compilation. On the basis of experience with C-EQUEL, some conclusions about the ease of implementing other proposed levels of compilation are possible. The extension of Level 1 to Level 2 would require little additional work. All that is needed is to look up names in the INGRES system catalogs. The extension of Level 2 to Level 3 would entail a considerable implementation effort. Programs would become sensitive to changes in the physical schema. Decomposition would have to be executed at preprocess-time in order to generate the query procedures. Further, the compiler would have to be invokeable at run-time, because data base, relation and attribute names may not be known until then. Once these implementation obstacles have been surmounted, Level 4 is relatively easy to implement. All necessary information is known when the Level 3 program is created.
Essentially, code is generated directly, as opposed to being generated with conditional statements to control execution.

REFERENCES

8. IDMS Sales Literature, Cullinane Corporation.