Query Optimization in MUMPS

Abstract

MUMPS, which stands for Massachusetts University Multiple Programming Systems, was originally designed to automate medical records, but is now used for a variety of purposes, particularly databases.

The MUMPS database stores information in array data structures called globals. For transaction-oriented applications, MUMPS provides an efficient way for programmers to read and update the database. The end user sees the database through custom procedures designed by programmers to support the user’s job function. Many organizations are successfully using a variety of these types of applications. The end user depends on programmers to retrieve information that is outside the scope of the programmed application set. These requests can often exceed the resources of the programming staff, leaving many requests unanswered.

Relational database technology and Structured Query Language (SQL) can help to solve this problem. The Relational model presents all data to the user as two-dimensional tables. This simple tabular structure is often easier for users to understand than the underlying global design.

A query language and report writer, using SQL as a high level access method can help end users to retrieve data from their databases.

The end user is often an expert on a particular subset of information stored in the database. That same user may have little or no understanding or appreciation for the database structures.

“We SQL users ... don’t worry about physical access to data ... we assume that we’ll be able to retrieve our data without any effort.”

A query entered with minimal user effort can require considerable processing resources. The goal for the query system is to produce an efficient response to each user request. Knowledge Based Systems, Inc. provides a report writer and information retrieval tool designed to achieve this goal. The KB_SQL application translates SQL statements into executable MUMPS routines. The system evaluates each request using information from a relational data dictionary. The system applies optimization strategies to produce an executable routine that references the appropriate MUMPS globals using efficient access strategies. SQL and the relational model are valuable complements to the MUMPS environment. This paper will discuss automatic access planning and code generation as one of many strategies for query optimization.


MUMPS Globals as Tables

The relational model considers all data items as elements of a table, consisting of several rows and columns. The following table of patient information has four (4) columns and six (6) rows of data.

Figure 1 - The PATIENTS table

<table>
<thead>
<tr>
<th>NAME</th>
<th>MRUN</th>
<th>SEX</th>
<th>DOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLY</td>
<td>11209</td>
<td>F</td>
<td>09/17/64</td>
</tr>
<tr>
<td>ROB</td>
<td>10395</td>
<td>M</td>
<td>11/03/59</td>
</tr>
<tr>
<td>JAMES</td>
<td>10444</td>
<td>M</td>
<td>04/06/78</td>
</tr>
<tr>
<td>DAVE</td>
<td>10123</td>
<td>M</td>
<td>03/30/61</td>
</tr>
<tr>
<td>RICK</td>
<td>10456</td>
<td>M</td>
<td>01/12/57</td>
</tr>
<tr>
<td>SHANNON</td>
<td>12110</td>
<td>F</td>
<td>04/09/90</td>
</tr>
</tbody>
</table>

The table provides a two dimensional image that is often more easily understood by an end user than a similar data set in a MUMPS global. Note that the table presents the data of birth (DOB) in an external format. The global stores the corresponding internal value.

Figure 2 - The PATIENTS global

```
^PAT(10123,1)=DAVE;M;43918
^PAT(10395,1)=ROB;M;43405
^PAT(10444,1)=JAMES;M;50134
^PAT(10456,1)=RICK;M;42380
^PAT(11209,1)=POLLY;F;45185
^PAT(12110,1)=SHANNON;F;54520
```

Using MUMPS code, programmers can traverse global structures with terse commands and functions. Even the simplest information request requires some programming training and experience. With SQL, the user can reference data elements by name using simple clauses. The SELECT clause is used to specify columns to be retrieved from the database. The FROM clause identifies the source of the data. The optional WHERE clause can be used to restrict the rows returned by the SELECT clause. The end result is a table of information. The SQL statements can often be more easily formed than the corresponding MUMPS code. The following simple SQL statement is a request to produce a list of female patients.

Figure 3 - SQL example #1

```
SELECT name, mrn, dob
FROM patient
WHERE sex = 'F'
```

For each query, the goal is to provide the results with a minimum amount of system overhead. KB_SQL uses an approach that translates SQL into executable MUMPS routines. The process involves parsing the SQL statements into an internal form, developing an access plan and building the MUMPS code that will extract data from the set of globals referenced by the query.

The role of the planning process is to "choose an efficient strategy for evaluating a given relational expression".

The system generates the following MUMPS routine in response to our original SQL request. All rows in the PATIENTS table will be searched. The result will include only the female patients.

Figure 4 - MUMPS example #1

```
PGM; simplified MUMPS for SQL
#1 S K1=0
A S
    K1=$O(^PAT(K1))
    I K1="" G B
    S X1=^PAT(K1,1)
    I $P(X1,"","",2)="F" G A
    W !,$P(X1,"","",1),?30,K1
    W !,
    W ?50,$$out+%date($P(X1,"","",3))
    G A
```

There are several types of optimizations that can be performed on queries. Constraint optimization, concentrating on the conditions specified in the SQL WHERE clause, will be the focus of this paper. In the first example, the

system applies the test for female patients as a selection constraint to limit the result of the query. A MUMPS routine called the planner organizes the request, evaluates multiple access plans, and ultimately selects an access strategy for the query.

Expectations
The minimum requirement is that the query system produces an accurate result for each user request. The expectation is that the planner produces an access strategy that is an improvement over some default strategy as entered by the end user. The end user forms a valid SQL request and the planner produces the correct results using an efficient strategy. Without optimization, the performance of each data request is limited by the knowledge and skills of the user at the time of the request. The planner uses the most accurate data definitions and the most efficient access strategies to satisfy each user request.

While no plan can be considered optimal in all cases, the expectation is that the selected strategy will be the best considering the information available at planning time. 4

To be effective, the planner must have access to information about all tables referenced by the query. This information must be carefully collected and managed by those individuals who have the most knowledge about the data and the associated global structures. The planner uses the information about where, how, and how much data is stored to produce an efficient access strategy. For MUMPS, an efficient strategy often means using densely packed indexes over full global scans wherever possible.

Justification for the Planning Process
In any environment, change is constant. The distribution of data changes, programmers add new global structures, develop skills and refine access techniques. In a complex database, it is difficult to know everything about the desired data and the associated structures. Through the data dictionary, the planner knows about all tables and indexes and the volume of data stored in those structures. The automatic planner can help to keep the system up-to-date. As the distribution of data changes or new access techniques are developed, queries can be compiled again, without modification, to take advantage of changes and improvements.

The planner will always fully evaluate the costs of alternative access planes.

The time spent is normally much less than the time spent executing a query in an inferior way. 5

The system incurs the expense of planning only once for each compile of a query. This routine can then be executed as often as needed. These are just a few reasons that justify the automatic optimization and code generation methods.

Key Terms and Ideas
It is important to have a general understanding of some key terms used in the description of the access planning process. This terminology provides a common language for the discussion of the process.

The term cost is the expense of the access plan, using row traversal as the unit of measurement. Row traversal for MUMPS is most often done using the $Order function. The cost determines when to use indexes instead of base tables. For multiple table queries, the cost determines the

---

4 Date, p. 334

order to access tables and when to apply constraints.

A base table is a real table, consisting of records that exist in MUMPS globals. A base table can have one or more associated index tables, also MUMPS globals, used for alternative access methods and cross-references. The index table contains a copy of selected data from the base table, possibly stored in a different format.

Primary keys and index keys are the values required to reference a single row in a table. In MUMPS globals, the primary keys are often the subscripts of the global, with rows as data nodes. In other cases, a single MUMPS data field in one global contains a group of values that are the primary keys for another table.

Figure 5 - PAT_BY_NAME index table

<table>
<thead>
<tr>
<th>NAME</th>
<th>MRUN</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAVE</td>
<td>10123</td>
</tr>
<tr>
<td>JAMES</td>
<td>10444</td>
</tr>
<tr>
<td>POLLY</td>
<td>11209</td>
</tr>
<tr>
<td>RICK</td>
<td>10456</td>
</tr>
<tr>
<td>ROB</td>
<td>10395</td>
</tr>
<tr>
<td>SHANNON</td>
<td>12110</td>
</tr>
</tbody>
</table>

In our sample PATIENT table, the medical record unit number (MRUN) column is the primary key. An index table will often have multiple keys. The figures above and below are examples of an index table, in tabular and MUMPS global format. The primary keys are the NAME and MRUN columns.

Figure 6 - PAT_BY_NAME index global

```
^PAT(-1,"DAVE",10123)=""
^PAT(-1,"JAMES",10444)=""
^PAT(-1,"POLLY",11209)=""
^PAT(-1,"RICK",10456)=""
^PAT(-1,"ROB",10395)=""
^PAT(-1,"SHANNON",12110)=""
```

The SQL WHERE clause specifies constraints, or predicates, that restrict the result of a query. SQL supports many advanced predicates, including the relational operators, the Boolean operators, and parentheses to force a specific order of evaluation.

If a predicate can be used to restrict a key, either primary or index, the restriction reduces the search cost. If a predicate can be used to reduce the number of rows selected, and by that decreases the cost of any additional searches, that restriction reduces the select cost.

Predicates are assigned a reduction factor based on how significantly the constraints can reduce the search and select costs. A typical reduction factor of two (2) implies that the predicate cuts the search and select cost in half. The equals (=) predicate is a special case. For example, an equals (=) predicate on the NAME column of the PAT_BY_NAME index table would be applied as a search constraint. The primary key traversal logic can be customized to limit the search and select to a single query. Without the index, the plan would require the query scan the entire PATIENTS table. The difference between search and select cost is subtle, but important in the discussion of the planning process.

Reducing the Search Cost

A major objective of the access plan is to find the access strategy that requires the fewest system resources. The planner considers the strategy with the lowest search cost to be the most efficient. The best way to reduce the cost is to recognize restrictions against primary keys. The planner will attempt to match constraints with keys whenever possible. In the following example, the planner chooses to use the PAT_BY_NAME index to optimize the search constraint on the NAME column.

Figure 7 - SQL example #2

```
SELECT name, mrung, dob
FROM patient
WHERE name = 'M' AND sex = 'F'
```
The optimized code produced for example #2 uses the PAT_BY_NAME index table on patient names. The constraint on the name reduces the search and select costs since it decreases the number of entries that must be checked. The constraint on sex cannot be applied to a primary key. The query searches all rows, selecting only those rows that satisfy the constraint.

Figure 8 - MUMPS example #2

```plaintext
PGM; simplified MUMPS for SQL #2 S K1="M" A S K1=S0(‘PAT(¬1,K1)) I K1="" G C I K2="" B S K2=S0(‘PAT(¬1,K1,K2)) I K2="" G A S X1=’PAT(K2,1) I $P(X1,"",2')="F" G B W !,$P(X1,"",1),?30,K2 W ?50,$out^%date($P(X1,"",3)) G B C W !!,"End>" Q
```

Organization of the Plan

The parsing step comes before the planning step. This step translates the query into an internal representation that can be interpreted and manipulated by the planner. The entire query can be viewed as an algebraic expression that requests data from the database. The first planning step is to examine all predicates specified in the WHERE clause of the query. Some predicates have a greater effect on efficient data retrieval than others. The not equal (<> predicate can reduce the select cost only after searching all entries. The greater than (>) predicate can reduce both the search and select costs. The equals (=) predicate reduces the search and select cost to the absolute minimum; one value will be searched with no looping. The planner organizes the constraints by reduction factor, using the most effective constraints before less effective ones.

If a WHERE clause contains several constraints separated by Boolean operators, the planner must separate the compound constraints into a list of simple constraints. Identify those search reducing predicates, including equals (=), less than (<), and greater than (>). These and other predicates have the potential to reduce the search cost. The planner separates the search predicates from the select predicates. A table of equivalent expressions allows the planner to consider all constraints despite how the user phrases the constraint.

Figure 9 - Constraints

<table>
<thead>
<tr>
<th>Value</th>
<th>Predicate</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>&gt;</td>
<td>‘M’</td>
</tr>
<tr>
<td>SEX</td>
<td>=</td>
<td>‘F’</td>
</tr>
<tr>
<td>‘M’</td>
<td>&lt;</td>
<td>NAME</td>
</tr>
<tr>
<td>‘F’</td>
<td>=</td>
<td>SEX</td>
</tr>
</tbody>
</table>

The planner identifies those constraints that compare columns from one table to another. This information is useful when deciding the order in which tables will be searched. The referenced table must be known at the time the query executes the constraint. The planner isolates all constraints that require special attention, such as compound constraints connected by the OR operator. The result is an internal blueprint of the query. By decomposing the constraints into a simpler form, the planner can evaluate all combinations of columns, predicates and comparison values. The atomic constraints can be manipulated and ordered more easily during the planning steps. By evaluating all options during the planning phase, the code produced by the build function is direct and efficient.

Calculate Baseline Costs

For every table referenced by the query, the planner compares the cost of using an appropriate index table to the cost of using the base table. For constraints involving non-indexed columns, the only option is to apply the constraint as each row is processed. The simple cost of the access plan is approximately equal to the number of rows in the table. When evaluating the cost of a particular key. The
planner considers the number of entries for that key value divided by any predicate reduction factors. In addition, each table can have a density factor, defined as a factor related to the expected number of rows per physical storage block. This allows a densely packed index to produce a lower cost at planning time. The planner evaluates these factors for every table referenced by the query.

### Limit the Planning Space

For even the simplest query, there can be an incredible number of options to consider. The planner considers the cost reduction for search and select constraints. The planner matches search constraints to all appropriate index tables. The select constraints reduce the estimated number of rows that might be selected for each table.

Consider a query that refers to four (4) different tables, each having three (3) candidate indexes. The planner picks a table, performs all calculations and selects an optimal strategy for that table. The planner repeats the process for the three remaining tables. By always selecting the best plan for each set of tables, the planner limits the planning space. The four tables with three indexes each could produce 6,144 (16*12*8*4) possible plan steps to be evaluated. Usually, the practice of choosing the best plan at each step will produce the desired result. It is sometimes possible that by selecting a more costly plan for table #1, the remaining tables could be accessed at an absolute minimum. The planning space would be increased to a total of 65,536 (16*16*16*16) possible evaluations. The planner chooses to consider the best plan at each step, thus reducing the total number of possible plans to be considered.

### A Cost Formula

The planner considers many factors when evaluating alternative access strategies. While sometimes obvious on visual inspection, these factors must be compared by the planner in some systematic way. A cost formula must consider predicate factors, density factors and numbers of rows for all tables requested by the query. It also considers search and select reductions. The following sample cost formula uses statistics, reduction and density factors for each table referenced by the query.

**Figure 10 - A cost formula**

```plaintext
SET C=N(1)/R(1)
FOR I=2:1; N C=N(i)/R(i)*C+C
SET C=C/D

WHERE
    C=cost
    N=number of keys
    D=density factor
    N(i)=number of entries for key(i)
    R(i)=search reduction factor for key(i)
```

The base table statistics for the PATIENTS table show a total cost of 1,000 different values of the MRUN column exist. For the index table, the same 1,000 entries exist with a somewhat different distribution. There are 800 different names. Therefore there is an average of 1.25 MRUN values for each NAME value. Note the density factor of two (2) for the index table. This suggests that the index is a densely packed global structure that is ideal for searches on the patient name column.

**Figure 11 - Base table statistics**

<table>
<thead>
<tr>
<th>PATIENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_NO</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Simple cost = 1000.00
Density factor = 1.00
Total cost = 1000.00

A comparison of the base costs for the PATIENTS table and the PAT_BY_NAME index reveals the expected results. The total cost
for an index is less than the cost for the base table.

Figure 12 - Index table statistics

<table>
<thead>
<tr>
<th>KEY_NO</th>
<th>COLUMN</th>
<th>NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NAME</td>
<td>800.00</td>
</tr>
<tr>
<td>2</td>
<td>MRUN</td>
<td>1.25</td>
</tr>
</tbody>
</table>

When considering index tables, the planner must be careful to select only those indexes that represent an equivalent set of data to that in the base table. Any index key that the query does not constrain must be required for all rows in the table. The planner considers if the format of the index key is different from the base table format. If so, the planner must ensure that comparisons against the index key produce the same result as if applied to the base table.

The planner considers each indexed column in turn as long as the cost for the index is less than the previous best cost. The result of this exercise is a set of baseline costs, the cost to retrieve each individual table. If the query references only one table, the planning is complete.

Calculate Alternative Plans

A typical query requires several tables to be joined together to produce a result. The join process compares values from one table to another. In cases where a join uses non-key values, the planner can choose to build a temporary index to satisfy the request.

The planner tries each table as first in a series of join evaluations. For each join, the planner will consider if a simple match on primary keys is possible. If all keys for a table can be determined from previously selected tables, then the table can be referenced without any additional searching. The cost of the plan can be reduced accordingly. Consider a query that requires data from the patients, visits, orders, and charges tables with the following relationships.

Figure 13 - Visits, Patients, Orders, and Doctors

The third example shows a SQL request for a simple listing of patients that have seen a particular doctor. Note how the WHERE clause includes several constraints comparing the data columns to primary key columns. These data columns that store the primary key to other tables are foreign key columns. The number of joins in this example is not a requirement, but it is typical of most queries.

Figure 14 - SQL example #3

```sql
SELECT patient.name
FROM patients, visits, orders,
     doctors WHERE doctors.name = 'MCLEAN'
     AND orders.doctor_no = doctors.doctor_no
     AND orders.acct_no = visits.acct_no
     AND visits.mrun = patients.mrun
```

Unlike the first examples that reference only a single table, this query requires data from several tables. The planner considers each table separately and as part of the whole. The following table shows the statistics for the four tables plus two very important index tables.
The index tables provide additional opportunities for improving the query. The code starts with the DOC_BY_NAME index on the DOCTORS table. The equals (=) predicate allows the planner to reference the name ‘MCLEAN’ without any searching. If a doctor row exists, the planner specifies that the next step is to get all orders for the doctor. The code uses the ORD_BY_DOC index to find all orders for the doctor. The plan specifies that for each order, the code will attempt to find the visit that created the order. From the visit, the code will find the patient data using the MRUN columns.

The query illustrates how the planner can choose to use indexes to optimize table joins to produce an efficient access strategy. Consider how the result would be different if the indexes did not exist. The planner could scan the entire ORDERS table to find all orders by the doctor. Having identified the orders, the visit and patient information can be retrieved using the ACCT_NO and MRUN columns. Having the information about the indexes helps the planner to produce a more efficient access strategy.

The planner receives the original data request without any specific instructions about how to execute the query. The system decomposes and evaluates the query in many ways.

The end result is a program consisting of a step to evaluate each group in an order such that no group is evaluated prior to a descendant group.\(^6\)

Using the data dictionary as a reference, the planner provides a script from which the build

\(^6\) Ullman, p. 279
function produces an accurate and efficient MUMPS routine. The routine can then be executed as often as needed.

**Special Cases**

Transaction processing applications use MUMPS to ease the detail-oriented work of a group of users. In addition to details, users often want to summarize the information in their databases. They also request “what if?” type of searches for future planning and retrospective analysis.

Sometimes the constraints in the WHERE clause do not reference primary key values. For example, consider a search for patient visits that were the result of an accident. In our simple database, the search could constrain the VISIT REASON column that is not indexed. To avoid a Cartesian product of tables, the planner can choose to build a temporary index to satisfy the request. If possible, the smaller of the two tables would be scanned from top to bottom, building an index on the constrained columns. In our simple example, the planner could build a temporary index to visits where the reason was accident. This intermediate step could then be joined with the larger table, thus making a single pass through the larger structure.

The planner will give special consideration to a WHERE clause that includes a list of conditions combined with the OR operator. The planner can choose to evaluate each component of the OR list separately, building a temporary index structure for each. A final step could then merge the intermediate steps into a single result.

While these types of optimizations are absolutely critical in a mature database environment, the planning information can be valuable in an application development environment as well. Developers can examine the plans for frequent requests to decide those situations that could be done more efficiently with index structures.

**Summary**

End user information requests can often exceed the available programming resources. The gap between required and available resources can be reduced by providing more programmers, making programmers more efficient, or by giving the end user the ability to retrieve information. The end user query system presents a significant challenge. Each request must access the database as accurately and efficiently as if designed by programmers.

SQL and the relational model provide an excellent environment in which to provide the query language and report writer tools. The non-procedural design of the language and the simple representation of tables create the opportunity to apply many automatic improvements. MUMPS has proven to be an excellent choice for database and transaction applications. Knowledge Based Systems uses SQL as an interface between the user and the MUMPS database. The system evaluates each query using information from the data dictionary to produce a set of executable MUMPS routines. This illustrates one of the many ways that SQL and MUMPS can be used to produce effective and efficient end user computing in a database environment.