DATABASE DESIGN USING COMPILER THEORY TECHNIQUES
Wade A. Pedlow, Pennsylvania Blue Shield

Abstract
With databases becoming larger and more complex, sometimes it is advantageous to create database calls which are a bit more sophisticated than traditional calls. Database Design using Compiler Theory Techniques is an overview of a project which involved the creation of a large database which could be transparently accessed in any order, not just the key, without destroying response time. The compiler theory techniques utilized are: Finite State Machines, Parsing Production Rules, Tree Structures, Look-up Tables, and Recursion thrown in for some fun.

Background
Pennsylvania Blue Shield has an approximate claim flow of 24 million claims per year. Of these claims, 33%(on average) are rejected for one reason or another. Our department Account Information Management, is responsible for back-end reporting to our accounts and, increasingly, this has included rejection reporting in addition to the regular paid utilization. This department thrives on responsiveness and the ability to obtain accurate information quickly is vital to our effectiveness. An hierarchical SAS [1] database, developed by Daryl Witmer, was created to house the accepted claims. This paper will describe enhancements to this system during the rejection database development.

The accepted claims database was designed using SAS, which allows access to claims indexed by issuing group number. The source of the database are unload tapes from an on-line finalized claims system. This database is hierarchical and has several tables. The database holds 16 million records with a response time, if accessed by the index, of five seconds. To access this database, a macro call is issued which has several parameters. Some are: an output SAS dataset, a list of group numbers to retrieve, the SAS variables needed from the database, and a condition statement, if any.

This database works fine should the call include the groups parameter(index); however, if the index is not used, a building process occurs for all records prior to applying the condition. This causes work space problems and slows the request with unnecessary processing.

When time came to create an on-line database for rejected claims, the same hierarchical approach was used; yet some interesting concepts were applied to make the call much more efficient.

It is these concepts which this paper will address. The concepts employed are: finite state machines, parsing, minimal node spanning and recursion. All of the code was developed using the SAS Macro Language.

Goals and Objectives
The major goals to be accomplished while developing the rejected claims database were the deficiencies which existed with the accepted claims database. They are:

- Minimal Node Spanning: Given the organization of the hierarchical database, some methodology needs to be incorporated to skip over the unnecessary tables during the downward processing.
- Flexible Start Location: In the general sense, the root table of the hierarchical database is the key. When accessed by the key extraneous records are not processed. Given that calls to the database will not necessarily include the key, some method of entering the database at the table of greatest impact, determined from the where clause, needs to be addressed. Therefore, eliminating the processing of extraneous records.
- Table Integration: Although a hierarchical database is designed to be accessed in top-down fashion, this is very limiting. The ability to access any table from any other table will enable the database to be together the two aforementioned objectives.
- The above objectives are obtainable. A programmer, given table layouts and a description of a particular call, can produce an efficient extraction path to the database.

Given that the database is finite, in terms of variables and tables, a program should be able to produce the same, efficient extraction path.

The basic necessities needed by either the programmer or the program are:

1) Variables wanted from the database
2) Groups, or index, for which to extract
3) Any applicable condition or where clause

With this information, the program, or programmer, should be able to generate the appropriate extraction path.

With our aggressive goals outlined, let's approach the project.

Approach
To understand the approach, we first must understand the basic hierarchical database. The hierarchical database is simply a modified SAS dataset which is sorted by all the variables. The variables are grouped in a fashion which is logical such that the variable which follows another variable can do so. Once sorted, the variables which define a particular group are broken into separate SAS datasets and the first occurrence of all values within a grouping is kept—eliminating duplicate observations.

eg. Example of groupings (variable associated with...)

<table>
<thead>
<tr>
<th>Issuing group</th>
<th>Group Number, Group Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benefits</td>
<td>Type Benefit, Effective Date</td>
</tr>
<tr>
<td>Claims</td>
<td>Claim number, Amount, Date</td>
</tr>
</tbody>
</table>

1334
Therefore, if a group has 12 claims, the claim SAS dataset will have
the twelve observations but the issuing group SAS dataset will only
have 1 observation. Effectively eliminating 11 redundant group
observations!

For each value within a table, two pointers exist which indicate the
beginning and ending observations numbers associated with that
value in the next table down. One note, accessing is completed for
all values in group to benefits prior to accessing the claims table.

This is intuitive and easily accomplished utilizing FIRST, and LAST,
options.

Refer to the diagram as I explain the approach to our objectives.

Minimal Node Spanning: For a particular call, I am not
interested in the variables on the BENEFITS table. To reconstruct
the call using the basic hierarchical database, I would have to
access the BENEFITS table to obtain the beginning and ending
pointers for each claim. However, if I add beginning and ending
pointers for the claim table to the group table, I can avoid the call to
the BENEFITS table altogether.

Although I have added some space to the group table, I have
shortened my response time by ignoring the unnecessary table.

Table Integration: Given our new database, if we did not have a
group, or index, specified, yet a where clause existed for, say, a
specific claim number, we would process an exorbitant amount of
records, (group to benefits then to claim), prior to finding the claim
number. We could find the claim number by entering the claim table,
but the variables under GROUP and BENEFITS would not be
available. By adding a single pointer for each table, which point to
the other tables, we now have threaded our hierarchical database
and integrated the database.

Flexible Start Location: Up to now, we have haphazardly
referred to the where clause. Under table integration, we stipulated
the where clause asked for a specific claim number. What if it also
asked for a specific benefit? How can the clause be applied at the
different levels if it contains variables across different tables?
Moreover, how can the start table be determined with multiple
variables in the where clause?

Seemingly complicated, there is a way to accomplish this. Two
processes are needed. First, the where clause needs to be split into
several where clauses which can be applied to the appropriate
tables. This can be accomplished using some compiler techniques.
Next, a table look-up needs to be established such that the
combination of variables wanted from the database and the
variables in the split where clauses will return the appropriate
extraction path.

One note, given the nature of the parent-child relationship,
recursion can be utilized during the creation of the extraction path.

Specifically

I believe the creation of the bypass pointers and back pointers is
self-apparent. I would like to start with the where clause actions.

Should a where clause be present, this clause needs evaluated.
During evaluation, two goals need to be accomplished. First is to
maintain a list of the tables to which portions of the clause can be
applied. This information is then used to help determine the
extraction path and start location. The other goal is to produce, if
logically possible, a where clause for each table. These clauses are
then applied to the particular table during extraction. The following
explains this further.

The first phase of evaluation involves a technique called scanning.
A scanner is a network of related states, figure 1, among which are
start, accept and fail states. Each character is scanned, which
places the network at a given state for the character evaluated.
Once all characters have been scanned, the final state is returned.
Should the returned state be acceptable, this indicates that the
scanner was able to create tokens, logical clusters of characters,
from the original where clause. These tokens are placed in a queue
and the queue is sent to a parser.
A parser is a set of specific production rules which enables the condition structure to be evaluated from the queue. Another added feature is the ability to easily create a tree structure of the resultant queue. See figure 2 for an example of production rules and figure 3 for an example of the scanner and parser in concert. The tree returned from the parser is instrumental for splitting the where clause among the tables.

One consideration that must be applied to the tree is the concept of "MAJOR OR". A MAJOR OR is defined as an OR relationship between variables located in different tables. These must be pruned, logically removed, from the tree prior to splitting the tree into several trees. Again, each tree will represent a where clause. Once pruned, the tree is split. All variables which can be applied to a table are logically connected to create one clause for each table.

At this point, we have the applicable where clauses, the variables used in the where clauses, the variables wanted from the database, and whether the index is to be used. This information, fed as parameters into a look-up table, will generate the appropriate, predefined, access path.

Once the access path is returned, the path is interpreted and a queue is created which contains strategically placed SAS control code and the split where clauses. One of the last items in the the queue is the full where condition as entered to maintain integrity and to apply any MAJOR OR conditions which had been previously pruned.

Finally, having a queue of the strategic SAS code, which is the access into our hierarchical database, which can build the actual SAS call recursively.

Conclusion
We have done a great deal by incorporating compiler structures to create a sophisticated data retrieval system. However, the benefits, both in minimized CPU and turnaround time, enable the end-user to spend more of his/her time analyzing, than waiting to analyze, the retrieved data.

The splitting of the where clause is a beneficial asset which eliminates excess record processing. The ability to start the extraction where the table look-up indicates the greatest constraint exists also eliminates excess processing. The real beauty of this type of design is the quickness of the extraction and the transparency of the call development to the user.

I would like to mention that the Condition Acceptance Network and the Parser where created quickly. With some additional effort, the lacking structures and syntax, addition of functions and dates, can easily be added. Each is a separate Macro and therefore can be replaced when fully tested.

Acknowledgments
I would like to thank Daryl Witmer for initially introducing me to the SAS Macro Language. The power behind this "Text Generator" can easily be exploited. Daryl's ability to utilize the Macro Language opened my eyes to its potential. I also would like to thank James Gigliotti for allowing creative latitude on this project.

References
1. SAS is a registered trademark of SAS Institute, Inc.
The created queue from the Network:

```plaintext
1 ( 
2 CLAIMDATE
3 =
4 '890101'
5 AND
6 BENEFIT
7 =
8 'D'
9 )
```

7  -> The accept state!!