# END USER QUERY FACILITY

## INTRODUCTION

### Related Applications

This application is a continuation in part of U.S. Patent Application Serial No. 08/154,343, filed November 17, 1993, which in turn is a continuation-in-part of U.S. Patent Application Serial No. 07/846,522, filed March 4, 1992, (now U.S. Patent No. 5,325,465, issued June 28, 1994).

### Technical Field

This invention pertains to end user query technology, and more specifically to an end user query facility which scouts for information by understanding the database model and guiding the user.

### Background

This invention pertains to end user query technology, introducing a novel approach to end user information access. Current end user query techniques require the user to understand database models in order to access information. For example, using prior art database models in which it is not uncommon to have dozens if not in excess of a hundred separate data base files interrelated, it is necessary for the database programmer to know in which file the desired piece of information is located, and then appropriately connect the files to achieve an orderly access of the specific file containing the desired information. This requires a fair amount of skill on the part of the database programmer and intimate familiarity of that programmer with the structure of the database which may be extremely complex. Furthermore, training new database programmers on an existing database model requires considerable amount of time and effort. One example of a prior art knowledge-based information retrieval system is the EASY-TALK product

This Sheet Has Been Replaced



available from Intelligent Business Systems of Milford, Connecticut. However, it appears that the EASY-TALK product requires the database developer to explicitly input to the EASY-TALK system the semantics of the database.

## SUMMARY OF THE INVENTION

In accordance with the teachings of this invention, it has been determined that there would be great usefulness in providing an end user query technology which is capable of automatically understanding the database model and guiding the user to scout for the desired information, thereby increasing productivity and ease of information access. In accordance with the teachings of this invention, the user is freed from the need to understanding the database model,. with the end user query facility of this invention quickly guiding the user to acquire the information. This is made possible by the end user query facility of this invention first recapturing the application semantics from the existing database model to provide a set of derived semantics. The derived semantics are then used by the end user query facility to intelligently guide the user to scout for the desired information in the database. In addition, the derived semantics can be easily updated by the end user query facility when the database model is changed.

In accordance with further teachings of this invention, the user is provided with a "natural" description of the database model to further ease his effort in information access. The "natural" description includes classes with an entity-relationship (E-R) model describing each class. The classes represent the different types of high-level objects whose information are contained in the database. The description of these classes is made possible by the end-user query facility first recapturing additional application semantics from the existing database model to provide a richer set of derived semantics. This enriched set of derived semantics is then used to identify the classes and generate their definitions. The definition of each class is then subsequently translated into an E-R model of the class.



This Sheet Has Been Replaced



In addition these classes and their E-R models can be easily updated by the end-user query facility when the database model is changed.

In accordance with the teachings of this invention, it has also been determined that there would be some usefulness in providing an end-user query technology that allows a user at a remote site with no on-line access to the database to still be able to make a query. This is made possible by integrating the end-user query facility of this invention with an electronic mail system so that the user at the remote site can send his query as a mail message and have the result of his query posted to him also as a mail message. In addition, a log of all query requests and their processing can be kept and analyzed to track usage and performance of the end-user query facility.

## BRIEF DESCRIPTION OF THE DRAWING

Figure 1 is a flow chart depicting one embodiment of an end user query facility constructed in the accordance with the teachings of this invention;

Figure 2 is a flow chart depicting one embodiment of semantics extractor 12 of Figure 1;

Figure 3 is a flow chart depicting one embodiment of a method suitable for use to build basic knowledge threads step 25 of semantics extractor 12 of Figure 2;

Figure 4 is a flow chart depicting one embodiment of a method suitable for use as information scout 15 of the embodiment to Figure 1;

Figure 5 is a flow chart depicting one embodiment of a method suitable for use as the search knowledge base step of the embodiment to Figure 4;

Figures 6a and 6b is a flow chart depicting one method suitable for use as the infer new threads from knowledge base step of the embodiment of Figure 4;

Figure 7 is a flow chart depicting an alternative embodiment of this invention including a model purifier in accordance with this invention;



This Sheet Has Been Replaced



Figure 8 is a flow chart depicting one embodiment of a semantics extractor constructed in accordance with the teachings of this invention interfacing with a model purifier;

Figure 9 is a flow chart depicting an alternative embodiment of a semantics extractor including a program analyzer in accordance with the teachings of this invention;

Figure 10 is a flow chart depicting an alternative embodiment of this invention including a security model specifier;

Figure 11 is a flow chart depicting an alternative embodiment of this invention which includes a model purifier, a security model specifier, an alternative embodiment of a semantics extractor, a class generator, and an entity-relationship (E-R) model translator;

Figure 12 is a flow chart depicting an alternative embodiment of a semantics extractor to extract additional semantics in the form of types of binary relationships and entity types of files in order for the class generator to generate class definitions;

Figures 13a-1 and 13a-2 form a flow chart depicting one embodiment of a procedure to identify kernel entities, suitable for use in the step of identify entity type of each file in the embodiment of Figure 12;

Figure 13b is a flow chart depicting one embodiment of a procedure to identify characteristic and associative entities, suitable for use in the step of identify entity type of each file in the embodiment of Figure 12;

Figure 14 is a flow chart depicting one embodiment of a procedure to reclassify kernel entities into pure lookup entities, suitable for use in the step of reclassify certain entities and binary relationships in the embodiment of Figure 12;

Figures 15a and 15b depict a flow chart of one embodiment of a procedure to generate class definition file A, suitable for use by the class generator in the embodiment of Figure 11;

This Sheet Has Been Replaced

Figure 17 is a flow chart depicting one embodiment of a procedure to generate class definition file B, suitable for use by the class generator in the embodiment of Figure 11;

Figures 18a and 18b depict a flow chart of one embodiment of a procedure for Process_File suitable for use with the embodiment of Figure 17; and

Figure 19 is a flow chart depicting an alternative embodiment of this invention in which a query facility is integrated with an e-mail system; and

Figure 20 is a flowchart depicting one embodiment of this invention which includes a Knowledge Thread Analyzer and

## DETAILED DESCRIPTION

### Overview

The following briefly describes one embodiment of this invention which is in the reporting language known as QUIZ. The language is part of a fourth generation language known as POWERHOUSE from COGNOS Incorporated of Canada. However, it is to be understood that the teachings of this invention are equally applicable to any end user query technology, including languages other than QUIZ.

Figure 1 depicts the operation of and interaction between the following modules:

a. The Semantics Extractor. Semantics Extractor 12 reads Database Model 10 of an application system and extracts the semantics of the database model. The semantics are stored in Keyword Library 13 as a list of keywords based on the item definitions and in Knowledge Base 14 as a list of file linkages of the database model. This is a key feature of the present invention and differs from the prior art in that prior art systems require a database developer or manager to explicitly define and provide the semantics of the database to the end user query facility.

b. The Information Scout. Information Scout 15 guides the user to specify the items to be reported in order to obtain the information he wants. This is carried out in two



This Sheet Has Been Replaced



steps. First, Report Item Selector 16 prompts the user for keywords that suggest the desired data item, for example DATE. Using a look-alike keyword search method, all items in Keyword Library 13 containing DATE are listed. The user then makes the appropriate selection. After selecting the items, Inference Engine 17 identifies the files containing the selected items. Inference Engine 17 then searches for the linkage(s) in Knowledge Base 14 connecting the identified files.

c. The Program Generator. Program Generator 18 accesses the linkages obtained by Information Scout 15 and generates the corresponding Source Program 19 to extract the information requested by the User.

d. The Compiler/Executor. The source program is compiled and executed against the database to generate the report using Compiler/Executor 20.

A more detailed description of one embodiment of these modules is now provided.

The Semantics Extractor

Figure 2 shows a flow chart of Semantics Extractor 12. In order to extract the semantics of an application, an application system must have a data dictionary that represents the application data model. In the POWERHOUSE fourth generation language which is used in one embodiment, all data dictionaries are available and can be found in "QDD" or "PDL" formats. They are created by the analyst when developing an application system. The data dictionary is made up of system options, security and definitions of each element, file, record and item used in an application system. The data dictionary of Database Model 10 is parsed by Data Dictionary Analyzer 21 in order to obtain the keywords and the information about each file and item in Database Model 10.

The keywords are derived from the item definitions in the data dictionary in the following manner. The name of



This Sheet Has Been Replaced



each item may be in the form of one complete word (e.g. SALARY) or may use more than one word separated with a hyphen (e.g. DATE-JOINED). Hyphens are removed from those item names with hyphens by Data Dictionary Analyzer 21. The resulting individual words obtained from the item names are then stored as keywords in Keyword Library 13, e.g. SALARY, DATE, and JOINED. These keywords are used later by Report Item Selector 16 (Fig. 1).

Next, Knowledge Base 14 is built. The first step involves extracting the following information from each file in Database Model 10:

a. File name
b. For every item in each file
    i) Item name
    ii) Item type (e.g. character, numeric, date)
    iii) Item size
    iv) If an item is a key, its key type (e.g. unique key, repeating key) or non-key

These files make up File Definition List 22 which is then used to determine the file relationships using the step Derive Binary Relationships 23. This step involves comparing each file definition in File Definition List 22 with the other file definitions in the same list to obtain its binary relationships with other files. A binary relationship consists of two files that can be linked. Two files can be linked if:

(i) both files have one item with the same name, AND
(ii) the item in the target (second) file is a unique or repeating key.

For example, in a Personnel Information System, the EMPLOYEES file would have an item called Emp-no which is a unique key to identify each employee and the SKILLS file would also have an item called Emp-no. Each record in a SKILLS file would contain a particular skill an employee has. As a single employee would have many different skills the



This Sheet Has Been Replaced

Emp-no would be repeated for each skill and so would be a repeating key. These two files would form a binary relationship since Emp-no is the common item and Emp-no in SKILLS file is a repeating key. The item Emp-no in the source (first) file which in this example is the EMPLOYEES file need not be a key.

Often two items could be coded differently but mean the same thing, for example, both P-NO and PART-NO could be used to represent a part number in a Inventory Control System. P-NO may occur in some files and PART-NO in others. These two items are said to be in the same "domain" called P-NO as shown below:

| File name | Item name | Domain |
|---|---|---|
| A-PARTS-FILE | P-NO | P-NO |
| B-PARTS-FILE | PART-NO | P-NO |

In such a case, we would not be able to establish a file relationship between the two files as the item names are different. But it would still be meaningful to establish a link between the two files with such items as the items are in the same domain.

In the example above, both A-PARTS-FILE and B-PARTS-FILE have items in the same domain called P-NO though the item names are different. A-PARTS-FILE and B-PARTS-FILE should therefore be linked. The linkage rule defined earlier is thus modified as follows to take into account items with not only the same name but with different names in the same domain:

(i) both files have one item in the same domain, AND
(ii) the item in the target (second) file is a unique or repeating key.

For every linkage between two files, the item in the source file can be a unique key, repeating key or non-key while the target file must either be a unique or repeating



This Sheet Has Been Replaced



key. From this restriction we can derive six possible valid types of file linkages. These are as follows:

| | Source file | Target file | Notation |
|---|---|---|---|
| a. | Unique key | Unique key | UU |
| b. | Unique key | Repeating key | UR |
| c. | Repeating key | Unique key | RU |
| d. | Repeating key | Repeating key | RR |
| e. | Non-Key | Unique key | NU |
| f. | Non-Key | Repeating key | NR |

However, in one embodiment the repeating to repeating (RR) combination which is item d above is not stored because this represents a bad file design. A repeating to repeating relationship indicates a many to many relationship which preferably should not exist in a normalized data model. The analyst is informed of such a finding and attempt to rectify it.

All the binary relationships found using the above rules are stored as follows:

a. Source file
b. Source file item to link to target file
c. Target file
d. Target file item to link to source file
e. Relationship (e.g. UU,UR,RU,NU,NR)

From the earlier example using EMPLOYEES and SKILLS files, the binary relationship are stored as:

| Source file | Item | Target file | Item | Relationship |
|---|---|---|---|---|
| EMPLOYEES | Emp-no | SKILLS | Emp-no | UR |

The set of binary relationships derived from the above step is stored in a Binary Relationship File 24. The next step, namely Build Basic Knowledge Threads 25, involves deriving knowledge of Database Model 10 of an application system from these binary relationships which is then stored

This Sheet Has Been Replaced

in Knowledge Base 14 in the form of knowledge threads. Each thread represents a linkage of many files. The following is an example of a knowledge thread:

EMPLOYEES → BRANCHES → EXPENSES

It contains an EMPLOYEES file linked to BRANCHES file which is then linked to EXPENSES file. Inference Engine 17 (Fig. 1) in Information Scout 15 later uses these threads in isolation or combination to infer the access paths in order to obtain the information requested by the user. For example, when the user wants to report the employees and their expenses in each branch, the above thread is used to generate the required path to navigate through an application database in order to acquire the required information.

Knowledge Base 14 is made up of basic and acquired knowledge threads. The following describes the derivation of basic knowledge threads by joining binary relationships from Binary Relationship File 24. The derivation of acquired knowledge threads is described later.

Figure 3 is a flow chart of one method suitable for use as Build Basic Knowledge Threads 25 of Fig. 2. The following describes how a single basic knowledge thread is built using the method of Fig. 3, with reference to the following example of a procedure Find-Thread:

```
Procedure Find-Thread (file-count, pointer 2)
begin
  old-file-count = file-count
  while (pointer 2 <> end of file)
  begin
    read binary relationship file using pointer 2
    until (thread-end = source file of binary relation
record)
       or (end of binary relationship file)
    if (binary relation record exists) and
       (target-file of record not exist in current-thread)
    then begin
```



This Sheet Has Been Replaced



```
            call Procedure Find-Thread(file-count, pointer 2)
            file-count = file-count + 1
            if (max-count < file-count)
            then max-count = file-count
            current-thread = current-thread + target-file
            reset pointer 2 to start of binary relationship
        file endif
  endwhile
  if file-count <> old-file-count
  then write current-thread into Basic Knowledge file
end

Note: Parameters file-count and pointer 2 are passed by value
      Current-thread, max-count are global variables
```

The formation of each thread begins with a first binary relationship record read from Binary Relationship File 24. This record forms an initial thread. However, if the first binary relationship record is of type 'RU' (repeating to unique) relationship, it is ignored and no initial thread is formed from it. The next record is then read and if it still is of type 'RU', it is again ignored until the next record is found that is not of type 'RU', which is then used to form the initial thread. The reason for doing this is that it is only necessary to form threads using either 'RU' or 'UR' types and not both as they lead to the same access path being inferred by Inference Engine 17. In this embodiment we are using 'UR' types. To keep track of the next record to read from Binary Relationship File 24, a pointer called pointer 1 is used.

The initial thread can be extended by linking with other valid binary relationship records. To do so, Binary Relationship File is searched. This time another pointer, namely pointer 2, is used to keep track of the next record to read from Binary Relationship File 24 to link to the thread to be extended. A link to extend the thread is said to be found if the first file of the binary relationship record and the file at the end of the thread, called the thread-end



This Sheet Has Been Replaced



file, are the same, or else the next binary relationship record is read using pointer 2. If a valid binary relationship record is found, we then examine if the target (second) file of this record is already in the thread. If it is not, it is added to the end of the thread, to become the thread-end. If it is, the next binary relationship record is read using pointer 2. The search ends when the end of Binary Relationship File 24 is reached using pointer 2. The formation of a single thread is then complete and is stored in Knowledge Base 14 as follows:

```
Thread-Head: . file name
             . file item  to link to next file
Thread-Body: . file name
                  . file item to link to the previous
                    file
                  . relationship to the previous file

             . file item to link to the next file
Thread-End:  . file name
                  . file item to link to the previous
                    file
             . relationship to the previous file
```

The thread-head contains the first file and the first file item. It is linked to the next file using the thread-body or thread-end. The thread-body is made up of a file, its item which has the same domain as the item of the previous file it is linked to and the relationship to that file. It also contains another item which is used to link to the next file. The thread-end is similar to the thread-body except that it does not have the item to link to the next file.

The following example briefly shows how Knowledge Base 14 is built up with basic knowledge threads using the above procedure. Assume that Binary Relationship File 24 contains the following records:



This Sheet Has Been Replaced



| Source file | Item | Target file | Item | Relationship |
|---|---|---|---|---|
| 1. EMPLOYEES | Br-no | BRANCHES | Br-no | NU |
| 2. BRANCHES | Br-no | EXPENSES | Br-no | UR |
| 3. EMPLOYEES | Emp-no | BILLINGS | Emp-no | UR |
| 4. BILLINGS | Emp-no | EMPLOYEES | Emp-no | RU |
| 5. PROJECTS | Proj-no | BILLINGS | Proj-no | UR |
| 6. BILLINGS | Proj-no | PROJECTS | Proj-no | RU |

The method of this example is then as follows:

| Pointer 1 at | Thread Formed |
|---|---|
| 1st record of above file | Use the first binary relationship record EMPLOYEES link to BRANCHES as the valid binary relationship record to form the initial thread. Next, examine whether this thread can be extended by searching through the same Binary Relationship File, this time using pointer 2. It can be linked to the second binary relationship record, namely BRANCHES link to EXPENSES to form the final thread:<br><br>EMPLOYEES → BRANCHES → EXPENSES |
| 2nd record | Similarly, using the next binary relationship BRANCHES link to EXPENSES record, form the initial thread:<br><br>BRANCHES → EXPENSES<br><br>This thread cannot be extended as the EXPENSES file cannot be linked to other binary relationship records. |



This Sheet Has Been Replaced



| | | |
|---|---|---|
| 3rd record | Using the 3rd binary relationship the third thread EMPLOYEES → BILLINGS → PROJECTS is formed as follows:<br><br>EMPLOYEES links to BILLINGS via Emp-no<br>BILLINGS links to PROJECTS via Proj-no<br><br>This thread uses different items, namely Emp-no and Proj-no to link the two binary relationship records. |
| 4th record | No thread is formed as relationship is 'RU'. |
| 5th record | Thread formed is PROJECTS → BILLINGS → EMPLOYEES as follows:<br><br>PROJECTS links to BILLINGS via Proj-no<br>BILLINGS links to EMPLOYEES via Emp-no<br><br>This thread uses different items, namely Proj-no and Emp-no, to link the two binary relationship records. |
| 6th record | No thread is formed as relationship is 'RU'. |

Each thread is stored in Knowledge Base 14 as follows, using the third thread above as example:

```
Thread-Head file = EMPLOYEES
            item = Emp-no
Thread-Body file = BILLINGS
          item-1 = Emp-no
    relationship = UR
          item-2 = Proj-no
```



This Sheet Has Been Replaced



```
Thread-End  file = PROJECTS
            item = Proj-no
    relationship = RU
```

Information Scout

Figure 4 is a flow chart of one method suitable for use as Information Scout 15 of Fig. 1. Report Item Selector 16 first guides the user to select the items to be reported. Inference Engine 17 then infers the access path based on the items selected.

Report Item Selector

The user is prompted for a keyword or presented with a list of all the words in Keyword Library 13 to choose from. When the user chooses to provide a keyword, Report Item Selector 16 lists all the items that match the keyword provided. For example, the keyword DATE could return the following list of items containing the word DATE.

```
DATE BIRTH
DATE JOINED
DELIVERY DATE
LAST DATE UPDATE
```

The user then selects the desired item from the list. In one embodiment, an explanation of each item is also displayed. This explanation is either extracted from the data dictionary of an application system by Data Dictionary Analyzer 21 or entered by a programmer analyst maintaining the application system.

Using the above method, the user selects all the items to be reported. For each item, the file containing it is automatically identified and added to the file list using Generate File List Step 27. When two or more items are selected from the same file, only one entry is made in the file list. This file list is sorted such that the first file is the file which has the highest number of items selected and the other files are in descending order of items



This Sheet Has Been Replaced



selected. When there is only one file in the list (that is, all the items selected come from the same file), no search of access paths is required. Otherwise, inference engine 17 is invoked to infer the access path.

Inference Engine

The first step involves searching for the optimal knowledge thread in Knowledge Base 14 to be used to generate an access path. If no optimal knowledge thread is found, the next step is to infer new knowledge threads, one of which is used to generate an access path (See Figure 4). However, if no new threads can be inferred, then the user is informed that there is no solution.

Knowledge base 14 comprises two sections: basic and acquired knowledge threads. The basic knowledge section contains the knowledge threads that are generated by Semantics Extractor 12. The acquired knowledge section also contains knowledge threads but these are knowledge threads that have been inferred by Inference Engine 17. The process of acquiring these knowledge threads is explained later.

In this embodiment, the search is made on the acquired knowledge section first. If a thread is found where all the files in the file list using step 27 of Figure 4 exists on the thread, this thread used to generate the access path. When the search is unsuccessful, i.e. no acquired knowledge thread is found which can match all the files in the file list, the search proceeds using the basic knowledge section. Figure 5 shows a flow chart of one method of performing the search for a knowledge thread from the basic knowledge section to be used to generate an access path.

The following example illustrates this search. Assume the file list from Step 27 in Figure 4 has been build up from the items the user has selected and it contains two files as follows:

File list:

EMPLOYEES

BRANCHES



This Sheet Has Been Replaced



Assume that the basic knowledge section has the following three basic knowledge threads with EMPLOYEES as the thread-head:

Thread 1: EMPLOYEES → BRANCHES → EXPENSES
Thread 2: EMPLOYEES → PAY
Thread 3: EMPLOYEES → BILLINGS

We define the number of files accessed as the number of files in the thread which exists in the file list starting from the thread-head.

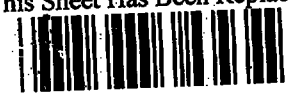
Thread 1 has files which match all those in the file list, namely EMPLOYEES and BRANCHES. The number of files accessed in this case is two (EMPLOYEES → BRANCHES) and is equal to the number of files in the file list. This thread is considered an "optimal thread" as all the files in the file list exist contiguously on the thread starting from the thread-head. Thread 2 and 3 are invalid because BRANCHES is not found on the thread.

When an optimal thread is found, the search ceases and the optimal thread is used to generate the access path. In the above example, the thread EMPLOYEES → BRANCHES → EXPENSES is used to generate the access path but only up to the BRANCHES file with the EXPENSES file ignored or "trimmed out" as it is not required in this example of the user query. Otherwise, Inference Engine 17 attempts to infer new threads by joining two or more threads together in parallel (see Figure 4). Those new threads that are found to be optimal, i.e. they have the number of files accessed equal to the number of files in the file list from step 27 of Figure 4, are then classified as acquired knowledge threads and stored in the acquired knowledge section of Knowledge Base 14.

The following section describes the process of deriving acquired knowledge threads. Before inferring new threads, Inference Engine 17 must first generate a list of knowledge threads consisting of basic knowledge threads which have at least one file matching any file in the file list from step 27 of Figure 4. This thread list is then sorted in



This Sheet Has Been Replaced



descending order of the number of files in each thread matching those in the file list. Within this sorted list of knowledge threads, if there are more than one thread with the same number of files matching those in the file list, the threads are then sorted in ascending order of the number of files in each thread. This results in a knowledge thread that has the most number of files matching those in the file list but has the least number of files on the thread being at the top of the list. This forms the sorted thread-list. This list is then used to form new threads by joining in parallel two or more threads. Figures 6a and 6b form a flow chart of how these new threads are inferred from the sorted thread-list.

For ease of explanation of how the above procedure works, assume that the file list from step 27 of Figure 4 contains the following three files:

EMPLOYEES
BRANCHES
PAY

and the basic knowledge section of the Knowledge Base 14 contains the following three threads as before:

Thread 1: EMPLOYEES → BRANCHES → EXPENSES
Thread 2: EMPLOYEES → PAY
Thread 3: EMPLOYEES → BILLINGS

Thus there is no basic thread that contains all three files in the file list. Inference Engine 17 next employs parallel join inferencing. The basic knowledge threads which have at least one file found in the file list is extracted and sorted as follows:



This Sheet Has Been Replaced

| Thread-list: | No. of files in thread matching those in file list | Thread-length |
|---|---|---|
| 1. EMPLOYEES → PAY | 2 | 2 |
| 2. EMPLOYEES → BRANCHES → EXPENSES | 2 | 3 |
| 3. EMPLOYEES → BILLINGS | 1 | 2 |

The flow chart of Figures 6a and 6b is then applied. First, the thread EMPLOYEES → PAY is added to the parallel list. Next, the file EMPLOYEES is read from the file list. Since EMPLOYEES exists on this thread it is removed from the file list. The next file from the file list is then read and examined whether it exists in the same knowledge thread. Since it does, it is also removed. However, the next file from the file list, namely BRANCHES, does not exist in the knowledge thread. It therefore remains in the file list. The next step is to retrieve from Knowledge Base 14 the basic knowledge threads whose thread head (first file) is EMPLOYEES.

As stated earlier, the Knowledge Base contains the following basic knowledge threads:

1. EMPLOYEES → BRANCHES → EXPENSES
2. EMPLOYEES → PAY
3. EMPLOYEES → BILLINGS

The basic thread EMPLOYEES → BRANCHES → EXPENSES has the BRANCHES file and is therefore added to the parallel list. But before it is added, the EXPENSES file is removed as it is not a file in the file list. The parallel list now contains the following threads:

Parallel list:

EMPLOYEES → PAY (obtained from the sorted thread list)
EMPLOYEES → BRANCHES (obtained from the basic knowledge threads)



This Sheet Has Been Replaced



The above knowledge threads have a parallel relationship through the common file EMPLOYEES and form the parallel thread as follows:

EMPLOYEES → BRANCHES
→ PAY

Next, the optimality test is applied. As the number of files accessed (which is earlier defined as the number of files in the thread starting from the thread-head) is equal to the number of files in the file list, an optimal solution has been found. This optimal thread is then added to Knowledge Base 14 as an acquired knowledge thread. However, if the optimality test fails, the above process to search for new parallel relationships is then repeated using the next thread on the sorted thread-list. If there are any new parallel relationships found, the optimality test is again applied.

In the event the sorted thread-list has been exhausted with no optimal acquired knowledge thread formed from the parallel relationships found, the user is prompted to select one of the parallel relationships found if there are more than one, or else the single parallel relationship found is used to generate the access path.

In cases where there are more than one parallel relationships, there may exist parallel relationships as follows:

a. EMPLOYEE → PAY
→ BRANCHES
b. EMPLOYEE → BRANCHES
→ PAY

As both these parallel relationships are semantically the same, one is redundant and is thus removed.

There may also be a case whereby there are no parallel relationships found after the sorted thread list has been



This Sheet Has Been Replaced



exhausted. In such a case, there is no solution to the end-user query.

The inference method as illustrated in the flow chart of Figures 6a and 6b also takes into account inference of relationships using more than one file as common files. For example, the method is able to infer the following parallel relationships, whereby EMPLOYEES and PAY are the two common files:

```
EMPLOYEES → PAY → file1
EMPLOYEES → PAY → file2
```

These then form the parallel thread:

```
EMPLOYEES → PAY → file1
                → file2
```

Program Generator

Based on the path inferred by Inference Engine 17, the corresponding QUIZ ACCESS statement is generated. In 'QUIZ', the file linkage is specified using the ACCESS statement with the following syntax:

```
ACCESS file [     LINK   item OF file TO item OF file ]
     [ [ { AND  } item OF file TO item OF file ] .. ]
           { LINK }
```

where ACCESS, LINK, AND, OF, TO are part of the 'QUIZ' syntax,

file refers to the file name,

item refers to the item name in the file to be linked,

[ ] means optional statement,

{ } means choose one of the options i.e. AND or LINK,

'.. means repeats one or more times

In QUIZ, there are two ways of defining a linkage between a number of files: hierarchical and parallel. A hierarchical linkage is defined with the "LINK ... TO" option



This Sheet Has Been Replaced



of the ACCESS statement. A parallel linkage is defined with the "AND ... TO" option of the access statement.

When a single thread is used to generate the access path, the hierarchical link is used. When a combination of two or more threads are used a parallel link is used. From the earlier examples,

1. EMPLOYEES → BRANCHES
   uses the hierarchical linkage to generate
   ACCESS EMPLOYEES LINK Br-no OF EMPLOYEES
   TO Br-no OF BRANCHES
2. EMPLOYEES → BRANCHES
   → PAY
   uses the parallel linkage to generate
   ACCESS EMPLOYEES LINK Br-no OF EMPLOYEES
   TO Br-no OF BRANCHES
   AND Emp-no OF EMPLOYEES
   TO Emp-no OF PAY

It should be noted that for a different language implementation, Inference Engine 17 and Knowledge Base 14 design need not change. Only the access path needs to be rewritten using the designated language.

Compiler/Executor

After Source Program 19 is generated, Compiler/Executor 20 is used to compile Source Program 19 into executable code. The compiled program is then executed to produce the report. In this embodiment, Compiler/Executor 20 is the QUIZ part of POWERHOUSE fourth generation language.

ALTERNATIVE EMBODIMENTS

Several additional embodiments are also taught, as will now be described.

In one embodiment, a Model Purifier 26 is included which allows a user to add or alter the key type and binary relationships in Knowledge Base 14 (Figure 7). The Model



This Sheet Has Been Replaced

Purifier interfaces with Semantics Extractor 12 to change Knowledge Base 14 accordingly.

In another embodiment, the functionality of Knowledge Base 14 is extended in order to store items that are derived from items in the database model. In this embodiment, the Model Purifier allows a user to input the specifications for the derived items. Derived items can also be obtained from source code of applications programs that access the database, in which case Semantics Extractor 12 serves to extract the derived item specifications from these programs. These derived items and their related database items together form a pseudo database file. Semantics Extractor 12 then uses this pseudo file to derive new binary relationships with normal database files and build new basic knowledge threads. The new binary relationships and new knowledge threads are then stored in Knowledge Base 14. In this embodiment, the functionality of Program Generator 18 is also extended so that after Inference Engine 17 has generated the access path which may contain pseudo database files as well as normal database files, Program Generator 18 uses this access path to generate source programs to obtain information from the normal database files and the pseudo database files.

In another embodiment, another component called the Security Model Specifier 29 is included to allow a user to input a security model which specifies the items of the database model that the user can access. This is called item security. The security model also specifies the range of values within an item that the user can access. This is called value security. To support this security, the functionality of Knowledge Base 14 is extended to store the security model. Functionality of Report Item Selector 16 and Program Generator 18 are also extended to use the security model in the Knowledge Base so that the information obtained from the database at query time meets the security model specification.

In another embodiment, the functionality of the Model Purifier is extended to allow a user to specify multiple domains for a data item and the aliases for a file containing



This Sheet Has Been Replaced



this item. In this embodiment, the functionality of Program Generator 18 is also extended to generate the appropriate file alias statements in the source program to access the database to satisfy the user query.

Figure 7 depicts one embodiment of this invention in which Model Purifier 26 is used. Model Purifier 26 serves to allow a user to add or alter the key type of items in Knowledge Base 14, e.g. the key type of an item can be changed from unique to repeating key. Model Purifier 26 is also used to allow a user to alter the binary relationships between database files located within Knowledge Base 14.

The rationale for the use of Model Purifier 26 in accordance with this embodiment of the present invention is that in some applications the database model or the application programs that access the database may not be rich enough for Semantics Extractor 12 to extract the necessary semantics including file linkages for the user to perform certain queries. Model Purifier 26 allow the user to input the additional semantics to satisfy these queries.

Figure 8 shows one embodiment of this invention in which Semantics Extractor 12 is extended to interface with Model Purifier 26. If the key types are altered, Model Purifier 26 activates Semantics Extractor 26 to re-derive the binary relationships and to rebuild the knowledge threads in Knowledge Base 14. If the binary relationships are altered, Model Purifier 26 activates Semantics Extractor 14 to rebuild the knowledge threads.

A user may specify an item or items to be reported that may not be found in Database Model 10. Examples of such items are as follows, which we called derived items as they are obtained by defining using the items of the database files:

| | Derived Item | Defined as |
|---|---|---|
| (i) | Employee-Name | Firstname of Employee file + Lastname of Employee file |

This Sheet Has Been Replaced

Note: Firstname & Lastname are items of database file Employee

| | | |
|---|---|---|
| (ii) | Sales-Commission | Sales-Amount of Invoice file x Commission Rate of Commission Table file |

Note: Sales-Account and Commission Rate are items of database file Invoice and Commission-Table, respectively

| | | |
|---|---|---|
| (iii) | Total-Sales | sum of Sales-Amount of Invoice file for each month of the year |
| (iv) | Ratio-of-Jan-Sales to-Total-Sales | (Sales-Amount of Invoice file for January)÷(Total-Sales derived from (iii) above) |

These derived items can be obtained from direct user input into Knowledge Base 14 or from Source Code of Application Programs 27 that access the database. To obtain such derived items from direct user input, the functionality of Model Purifier 26 is extended to meet this requirement. To obtain such derived items from Applications Programs 27, the functionality of Semantics Extractor 12 is extended with Program Analyzer 28 (see Fig. 9) to extract the derived items from these programs. In addition to the definitions of the derived items, the data type, size, and format of these derived items are also extracted from Applications Programs 27 by Program Analyzer 28 of Semantics Extractor 12. These derived items and their related database items together form a pseudo database file which is included in File Definition List 22 (see Figure 9). Semantics Extractor 12 then uses this pseudo file in File Definition List 22 to derive new binary relationships with normal database files in File Definition List 22 and build new basic knowledge threads. The new binary relationships and new knowledge threads are



This Sheet Has Been Replaced



stored in Knowledge Base 14. Besides the Model Purifier 26 and Semantics Extractor 12, the Program Generator 18 is also extended so that after the Inference Engine has generated the access path which may contain pseudo database files as well as normal database files, the Program Generator 18 uses this access path to generate source programs to obtain information from the normal database files and pseudo database files.

The following is a description of one embodiment of an algorithm suitable for use by Program Analyzer 28 of Semantic Extractor 12 to extract the derived items in accordance with the teachings of this invention:

a. From a single pass application program

A single pass application program is a program that contains only one database access statement.

```
Step 1:     Extract the database files that are accessed in
            the application program.

                For each database file accessed
                        Extract the list of items of database
                        file,
                        Store this list and the database file
                        name as a pseudo database file in the
                        knowledge base.
                Next database file

Step 2:     Extract the list of derived items and their
            definitions from the application program

                For each derived item
                        Scan for the data type, size and
                        format,
                        Store the derived item name, its
                        definition, data type, size and format
                        in the same pseudo database file
                        obtained from step 1 above.
```



This Sheet Has Been Replaced



Next derived item

b. From a multiple pass application program

A multiple pass application program consists of many single pass application program "stringed" together with the output of the previous pass being used as input by the current pass.

Step 1: Create a new pseudo database file 1 for the first pass (i=1) using the steps of the single pass application program above.

Step 2: For each of the remaining passes i.e. i=2 to n with n being the last pass,

Create a pseudo database file i

Store the location of the previous database pseudo file (i-1) in pseudo database file i.

These derived items are then presented to the user together with the items from the database model for the user to select from in order to satisfy the user query.

In order to generate a program from the access path inferred by Inference Engine 17 that may contain pseudo database files, in addition to the above described extension of the functionality of Model Purifier 26 and Semantics Extractor 12, the functionality of Program Generator 18 is also extended as follows:

Step 1: Examine the access path generated by the Inference Engine 17.



This Sheet Has Been Replaced



Step 2: For every file in the access path:

If it is a pseudo database file,

Then (i) generate source program to produce data for the pseudo database file using the derived item definitions; and

(ii) generate source program to extract the information from both the database and the data of the pseudo database file produced by the program of step 2 (i) above.

Next file

In another embodiment, a security model specifies the items a user can access (item security) and the range of values within an item the user can access (value security). In item security, a user is assigned access rights to a subset of the list of items available in Knowledge Base 14. In value security, a user is assigned access rights to a range or ranges of values within an item in Knowledge Base 14. For example, the access rights could be DEPT-NO = 101, SALARY <2000, or SALARY of PAY file if EMP-GRADE of EMPLOYEE file >8.

Figure 10 shows the Main Flowchart of one embodiment of the present invention which supports a security model. As shown in Figure 10, Security Model Specifier 29 is provided for the user to input the security model into Knowledge Base 14. Also, in accordance with this embodiment, the functionality of Report Item Selector 16 and Generate File List 27 module (Figure 4) of Information Scout 15, as well as Program Generator 18 are extended to support the security model as is now explained.

In one embodiment, the functionality of Report Item Selector 16 is extended such that only the items defined in



This Sheet Has Been Replaced



the security model that are accessible by the user will be presented to the user for selection at query time.

In one embodiment, the functionality of Generate File List module 27 is extended to perform the following:

```
Step 1:     For each query item selected
                Retrieve the file(s) containing the query
                item.
            Next query item

Step 2:     For each file retrieved in step 1
                For each item in the file
                        Retrieve any value security defined
                Next item
            Next file

Step 3:     For each value security retrieved in steps 1 and
            2
                If the security definition involves value
                from another file
                Then add the file to the file list for
                inferring the access path
            Next value security
```

In one embodiment of this invention, the functionality of Program Generator 18 is extended to perform the following:

```
Step 1:     For each query item selected
                Retrieve any value security defined on the
                item.
                Retrieve the file(s) containing the query
                item.
                Next query item

Step 2:     For each file retrieved in step 1
                For each item in the file
                        Retrieve any value security defined
```



This Sheet Has Been Replaced



```
                    Next item
               Next file

Step 3:        For each value security retrieved in step 2
                    Join the value security definition using
                    the AND condition.
               Next value security
```

In yet another embodiment of this invention, the functionality of the Model Purifier is extended to specify multiple domains for a data item. An example of an item with multiple domains is Code Id of a database file called Master Code which may contain 2 items, namely Code Id and Code Description. The data for the Master Code file could, by way of example, be as follows:

| Code Id | Code Description |
|---|---|
| RACE01 | Chinese |
| RACE02 | Caucasian |
| . | |
| . | |
| . | |
| . | |
| CITY01 | San Francisco |
| CITY02 | Singapore |
| . | |
| . | |
| . | |

In this example, this file is used to store codes for races and cities. However, races and cities are two separate domains. To break the codes into the two separate domains, the Code Id must be redefine as Race Code and City Code. The Master Code file will then have the following file aliases:

```
Race Master Code to contain Race Code Id
City Master Code to contain City Code Id
```



This Sheet Has Been Replaced



To support this multiple domain for an item, we need to extend the Model Purifier to allow a user to specify the multiple domains and the corresponding file aliases. Figures 12 and 13 depict one embodiment of a main flowchart and Semantics Extractor 12 of the present invention which are capable of supporting multiple domains. The redefined items, eg Race Code Id and City Code Id, are stored in Keyword Library 13 and the alias file(s) are stored in Knowledge Base 14. In this embodiment, the functionality of Program Generator 18 is extended to identify the alias file(s) in the access path generated by Inference Engine 17 and to generate the necessary alias file(s) statements in the source program.

There are also a number of alternative embodiments of this invention related to how the Knowledge Base is physically implemented. The various forms of implementations can be considered in the following manner:

a. Precreated versus Run-time Creation
b. Permanent Storage versus Temporary Storage
c. Outside the Database Model versus Inside the Database Model

In precreated mode, the Knowledge Base is created once and used for every query over a period of time. Should the Database Model change, then the Knowledge Base is recreated to properly reflect the changed Database Model.

In run-time mode, the Knowledge Base is created at query time for every query, regardless of whether the Database Model has or has not changed since a previous query.

In permanent storage, the Knowledge Base is usually implemented in secondary storage devices such as a disk drives. In temporary storage the Knowledge Base is implemented in memory such as RAM (random access memory). A Precreated Knowledge Base is preferably (although not necessarily) implemented in permanent storage while a Run-Time Knowledge Base is preferably (although not necessarily) implemented in temporary storage. The reason is that as precreated knowledge is created once and reused many



This Sheet Has Been Replaced



times, it would be beneficial to keep it permanent so that even if the electrical power to the computer system is switched off, the Knowledge Base is retained and can be used again once power is restored. In run-time mode, as the Knowledge Base is created at query time and not reused for the next query, it is not necessary to implement the Knowledge Base in permanent storage.

In POWERHOUSE fourth generation language, the database model is implemented in the data dictionary. In SQL database language, the database model is implemented in the system catalog.

In one embodiment of the present invention, the Knowledge Base is implemented outside the data dictionary or system catalog. In an alternative embodiment of the present invention, the Knowledge Base is implemented to reside partially or fully inside the data dictionary or system catalog, as the data dictionary or system catalog is a form of storage that can contain the Knowledge Base.

GENERATION OF CLASSES AND THEIR TRANSLATION INTO ENTITY-RELATIONSHIP MODELS

An additional embodiment is taught as will now be described. The purpose of this embodiment is to provide another way for a user to interface with Report Item Selector 16 to formulate this query. Instead of a long list of items for a user to pick as described earlier the user is initially prompted for a class to pick out of a number of classes that make up his application. An entity-relationship (E-R) model of the selected class is then presented to the user for selection of the desired attributes. The desired attributes then constitute a designation of the information to be extracted from the database. It is also possible for the user to formulate the query by selecting the desired attributes from two different classes.

This invention introduces a way of selectively grouping the database files into what we called classes and translating each class definition into an E-R model. These classes represent the different types of high-level objects



This Sheet Has Been Replaced



that make up the application, e.g. employees, customers, orders, invoices, etc.

This embodiment, as shown in Figure 11, involves the following:

- the extension of Semantics Extractor 12 to extract more semantics from Database Model 10
- the addition of Class Generator 30
- the addition of E-R Model Translator 31
- the addition of E-R Model of Classes File 32

Extension of Semantics Extractor 12

Figure 12 is a flowchart of one embodiment of this invention utilizing Semantics Extractor 12 which is extended to support the extraction of additional semantics from the Database Model 10. There are several additional parts to this embodiment.

Classify Binary Relationships (BR)

Classify Binary Relationships step 33 classifies each binary relationship in Binary Relationships File 24 into one of the following types, in this embodiment:

- has_children
- has_wards
- has_subtype

We shall now describe these three different types of binary relationships and explain how they can be identified from the key types of the items of the database files.

Earlier we have described six kinds of file linkages, namely

UR (unique key to repeating key)
RU (repeating key to unique key)
NU (non-key to unique key)
UU (unique key to unique key)



This Sheet Has Been Replaced



RR (Repeating key to repeating key)
NR (non-key to repeating key)

Of these six kinds, both the RR and NR combination should not exist in a normalized data model as they represent a bad file design.

Let us now consider the remaining four kinds of file linkages. For the UR kind, there are two possible types of binary relationships that can exist, namely

- has_children
- has_wards

Let us explain the meaning of these two types with examples. Suppose we have an EMPLOYEE file with Emp_No as a unique key and another file called SKILLS to contain the skills of every employee. The SKILLS file has a repeating key item called Emp_No and a non-key item called Skill but no unique key. The binary relationship between these two files is as follows:

| Source File | Item | Target File | Item | Relationship |
|---|---|---|---|---|
| EMPLOYEE | Emp_No | SKILLS | Emp_No | UR |

This binary relationships is a "has_children" type because EMPLOYEE is not only related to SKILLS but the relationship is one where EMPLOYEE considers SKILLS as its "children". This is because the records of SKILLS can only be created if their corresponding EMPLOYEE record exists. This "has_children" type of UR binary relationship can be identified by the fact that the source file of the binary relationship has a unique key but the target file has a repeating key but no unique key.

Let us consider another example involving 3 files as follows:



This Sheet Has Been Replaced



| File | Unique Key | Repeating Key |
|---|---|---|
| CHAPTER | Chap_No | - |
| SECTION | Chap_No, Sect_No | Chap_No |
| PARAGRAPH | Chap_No, Sect_No, Para_No | Chap_No, Sect_No |

In this case CHAPTER has a unique key called Chap_No and no repeating key. SECTION has a composite unique key and a repeating key within the composite unique key. These files have the following binary relationships:

| Source File | Item | Target File | Item | Relationship |
|---|---|---|---|---|
| CHAPTER | Chap_No | SECTION | Chap_No | UR |
| SECTION | Chap_No, Sect_No | PARAGRAPH | Chap_No, Sect_No | UR |

Both these binary relationships are "has_children" type because records of SECTION can only EXIST if the corresponding record of CHAPTER exists. Similarly, records of PARAGRAPH can only exist if the corresponding record of SECTION exist. These "has_children" type of UR binary relationships can be identified by the fact that the source file and target file have a unique key and the target item is a repeating key within the unique key of the target file as well as the binary relationship being UR.

Consider another example where a CUSTOMER file has Cust_No as its unique key and another file INVOICES has Inv_No as its unique key and Cust_No as its repeating key. They both have a binary relationship as follows:

| Source File | Item | Target File | Item | Relationship |
|---|---|---|---|---|
| CUSTOMERS | Cust_No | INVOICES | Cust_No | UR |

This binary relationship is a "has_wards" type. In this case, we cannot consider CUSTOMER as having INVOICES as its "children" since INVOICES have their own identity through their own unique key, namely Inv_No. Instead, we could consider CUSTOMER as a "guardian" having INVOICES as its "wards" because INVOICES belong to their respective CUSTOMER.

A "has_wards" type can be identified by the fact that both the source and target files have a unique key and the



This Sheet Has Been Replaced



target item is not part of the target file unique key as well as the binary relationship being UR.

Let us next consider the NU and RU binary relationships. The RU binary relationships are the inverse of the UR Binary relationships. We therefore classify them as either the "inverse_of_has_children" or the "inverse_of_ has_wards". As for the NU binary relationships, we create an inverse of it and assign to this inverse a "has_wards" type. We then store it in Knowledge Base 14. For example, suppose we have an EMPLOYEE file having a NU binary relationship with a BRANCHES file using the Branch_Code item from both files as follows:

| Source File | Item | Target File | Item | Relationship |
|---|---|---|---|---|
| EMPLOYEE | Branch_Code | BRANCHES | Branch-Code | NU |

We create an inverse as follows with the type specified as "has_wards":

| Source File | Item | Target File | Item | Type |
|---|---|---|---|---|
| BRANCHES | Branch_Code | EMPLOYEE | Branch_Code | has_wards |

This binary relationship is then stored in Knowledge Base 14.

Lastly let us consider the UU binary relationship. Such a binary relationship is called a "has_subtype" type. Consider for example, a file called EMPLOYEE with Emp_No as its unique key and two other files MONTHLY_RATED_EMP and DAILY_RATED_EMP both of which also have Emp_No as their unique key. As an example, an employee is either a monthly rated employee or a daily rated employee but not both. We consider this as EMPLOYEE having MONTHLY-RATED_EMP and DAILY_RATED_EMP as its subtypes. The following binary relationships between EMPLOYEE and the two other files with EMPLOYEE as the source file reflect this "has_subtype" type of binary relationships:

| Source File | Item | Target File | Item | Type |
|---|---|---|---|---|
| EMPLOYEE | Emp_No | MONTHLY_RATED_EMP | Emp_No | UU |
| EMPLOYEE | Emp_No | DAILY_RATED_EMP | Emp_No | UU |



This Sheet Has Been Replaced



If a file has more than one subtype it is possible to automatically identify which of the two opposite UU binary relationships is a "has_subtype" and which the "inverse_of_has_subtype" by comparing them with another pair of opposite UU binary relationships. The ones with the same source file are then the "has_subtype". However, it is not possible to do so if a file has only one subtype. In a later section of this patent description, we explain that a user will have to input this knowledge using Model Purifier 26. (Note: In the SQL language using primary keys and foreign keys, it is possible to automatically identify which of the two opposite binary relationships is a "has_subtype" even if a file has only one subtype.)

We earlier describe the extension of Model Purifier 26 to allow an user to manually specify multiple domain and corresponding file aliases. In this embodiment, this process can now be automated using the binary relationships. We shall now describe how the procedure Derive Binary Relationship 23 can be further extended to identify files that requires file aliases to be defined and to define these file aliases. A file that require file aliases to be defined can be identified by the fact that there are more than one binary relationships of the NU or RU kind which has the same target file and item. For example, suppose there are two NU binary relationships as follows:

| Source File | Item | Target File | Item | Type |
|---|---|---|---|---|
| EMPLOYEES | Race_Code | MASTER_CODE | Code | NU |
| EMPLOYEES | Citizenship_Code | MASTER_CODE | Code | NU |

The target file, namely MASTER_CODE, and target item, namely Code, is the same for both binary relationships. For each such binary relationships, we create a file alias of the target file and replace the target file of the binary relationship with the file alias, e.g. for the above two binary relationships, we create the following file aliases for the MASTER_CODE file and store them in the Knowledge Base 14:



This Sheet Has Been Replaced



EMPLOYEE_Race_Code
EMPLOYEE_Citizenship_Code

and change the binary relationships to:

| Source File | Item | Target File | Item | Type |
|---|---|---|---|---|
| EMPLOYEE | Race_Code | EMPLOYEES_Race_Code | Code | NU |
| EMPLOYEE | Citizenship_Code | EMPLOYEES_Citizenship_Code | Code | NU |

Identify Entity Type of Each File

In Identify Entity Type of Each File step 34 a database file is classified as one of the following types:

- kernel entity
- characteristic entity
- associative entity
- subtype entity
- pure lookup entity

Kernel entities are entities that have independent existence, they are "what the database is really all about". In other words, kernels are entities that are neither characteristic nor associative, e.g. suppliers, parts, employees, orders, etc. are all kernel entities.

A characteristic entity is one whose primary purpose is to describe or "characterize" some other entity. For example, the file SKILLS which contains the SKILLS an employee has is a characteristic entity of the EMPLOYEE entity. Characteristic entities are existence-dependent on the entity they described which can be kernel, characteristic, or associative.

An associative entity is an entity whose function is to represent a many-to-many (or many-to-many-to-many, etc.) relationship among two or more other entities. For example, a shipment is an association between a supplier and a part. The entities associated may each be kernel, characteristic, or associative.



This Sheet Has Been Replaced



A subtype is a specialization of its supertype. For example, as described earlier MONTHLY_RATED_EMP and DAILY_RATE_EMP are subtypes of EMPLOYEE.

Lastly, we have entities that look like kernel entities but should not be classified as one because their purpose is solely for lookup of code description.

Figure 13a describes the procedure to identify kernel and pure lookup entities. It first identifies those database files that are either kernel or pure lookup entities using the following rule:

1) Identify those database files that have a unique key.
2) Of these database files, eliminate those that are used as a target file in any "has_children" or "has_subtype" binary relationships.

These database files are either kernel or pure lookup entities. To distinguish between the two it next uses the following rule:

1) IF such a database file has no "children" or "subtype", i.e. it is not a source file in any "has_children" or "has_subtype" binary relationship; AND
2) IF it is not a "ward" i.e., it is not a target file in any "has_wards" binary relationship;
3) THEN it is a pure lookup entity;
4) OTHERWISE it is a kernel entity.

Figure 13b describes one embodiment of a procedure to identify characteristic and associative entities. It uses the following rule:

1) IF a database file appears more than once as a target file in "has_children" binary relationships, THEN it is an associative entity;



This Sheet Has Been Replaced



2) OTHERWISE, IF it appears only once, THEN it is a characteristic entity.

Subtype entities are easily identified as they are the target files in "has_subtype" binary relationships.

Note that the SYSFILES as used in Figure 13a is a part of File Definition List 22 which has been redefined as consisting of two files, namely

- SYSFILES
- SYSFILEITEMS

SYSFILES is used to store the following:

- file name
- indicator whether it is an alias file or a real file having alias files
- if alias file, its real file name
- entity type

SYSFILEITEMS is used to store the following:

- file name
- item name
- item type (e.g. character, numeric, date)
- keytype of item, e.g. unique key, repeating key, or non-key

Reclassify Certain Entities and Binary Relationship 35

Even though we have earlier identified some files as kernel entities, some of these kernel entities should be reclassified as pure lookup entities. Consider for example a kernel entity EMPLOYEE having an associative entity LANGUAGE_SPOKEN whose other kernel entity is LANGUAGE. LANGUAGE_SPOKEN has two items, namely a repeating key called Emp_No and another repeating key called Language_Code. The LANGUAGE file has only two items, namely a unique key item called Language_Code and a non-key item called Language_Desc.



This Sheet Has Been Replaced



Even though we earlier identified LANGUAGE as a kernel entity since it has LANGUAGE_SPOKEN as its "children", LANGUAGE should be reclassified as a pure lookup entity since it is used solely by the LANGUAGE_SPOKEN file to obtain the description of the Language_Code.

Figure 14 is a flow chart depicting one embodiment of a procedure to identify these kernel entities and modify them to pure lookup entities. It uses the following rule to do this:

1) IF a kernel entity has only associative entities and no characteristic entities; and
2) IF it is not a target file in any "has_ward" binary relationships;
3) THEN modify the kernel entity into a pure lookup entity. Also, modify the associative entities of this kernal entity into characteristic entities if the associative entities has only one other entity that associates it.

Next we access all those "has_children" and "has_wards" binary relationships whose source file is one of these pure lookup entities. We then modify them into a new type called "inverse_of_pure lookup" type. We use the word "inverse" as the lookup direction is not from source file to target file but from target file to source file.

Class Generator 30

Class Generator 30 (Fig. 11) generates a definition of a class for each kernel entity in the database and stores this definition as a Class Definition File (CDF) in Knowledge Base 14. A class is a cluster of files whose structure is a tree. The root of the tree is a kernel entity which defines the core attributes of the class. The tree has the following main branches:

(i) a branch for each of the subtypes of the root kernel entity



This Sheet Has Been Replaced



(ii) a branch for each of the "wards" of the root kernel entity

(iii) a branch for each of the characteristic entities of the root kernel entity

(iv) a branch for each of the associative entities of the root kernel entity

These branches are derived using the "has_subtype", "has_wards", and "has_children" binary relationships with the root kernel entity being the source file.

Each of the above subtype, characteristic, and associative entities could also have their own branches which are their characteristic or associative entities. The latter characteristic or associative entities could also have their own characteristic or associative entities, and so forth. These branches are derived using the "has_children" binary relationship with the target files of these binary relationships forming the new branches. The procedure of Figure 15a and 15b together with the sub-procedure Include_File (list) of Figure 16a and 16b are used to derive the above branches which are then stored as a set of lists in Class Definition File A (CDF A). An example of such a list is:

```
File      =   EMPLOYEE
item      =   Emp_No
File      =   BILLINGS
item      =   Emp_No
BR Type   =   "has_children"
item      =   Proj_No
File      =   PROJECTS
item      =   Proj_No
BR type   =   "inv_of_has_children"
```

This list contains a file EMPLOYEE, linked to a file BILLINGS, which is linked to file PROJECTS. The binary relationship (BR) type from EMPLOYEE to BILLINGS is "has_children" using the item Emp_No from both files and the



This Sheet Has Been Replaced



BR type from BILLINGS to PROJECTS is "inv_of_has_children" using the item Proj_No from both files.

Let us now describe how a list in CDF A is produced using this procedure. It starts by initializing a list to the first kernel entity in SYSFILES. This kernel entity forms the root kernel entity of a class to be generated. Next it looks for a subtype entity of this kernel entity using a sorted SYSFILES and the Binary Relationships File 24. This file has been sorted in descending order of subtypes, kernels, associative, characteristics, and pure lookups. If one subtype entity is found, it is added to the list together with the name of the corresponding binary relationship type, which in this case is "has_subtype" and the names of the items used. It then calls on a sub-procedure Include_File (list), for example as depicted in Figure 16a and 16b, to find associative and characteristic entities of the subtype entity. If a characteristic entity is found, it is added to the list together with the name of the corresponding binary relationship type which in this case is "has_children" and the names of the items used. The sub-procedure Include_File (list) is then called again, this time to find other associative entities or characteristic entities of the characteristic entity. If no such entity can be found the list is then written to the Class Definition File A (CDF A).

Besides these branches, the tree of a class also has what we called "lookup" branches originating from each node in the above branches. These "lookup" branches are derived using the "inv_of_has_wards" and "pure_lookup" binary relationships, with the node being the source file and the target file forming new branches. Furthermore, the new branches could also have their own new "lookup" branches and so forth. These subsequent "lookup" branches are formed using not only the "inv_of_has_wards" and "pure_lookup" binary relationships but also the "inv_of_has_children" binary relationships with the target file forming the new "lookup" branches. The procedure of Figure 17 together with the sub-procedure Process_File of Figure 18a and 18b are



This Sheet Has Been Replaced



suitable for use to derive these branches, which are then stored as a set of lists in Class Definition File B (CDF B).

Let us describe one example of how a list in CDF B is produced using this procedure. It starts by reading in the first record of CDF A. A list is then initialized to the first file in this CDF A record. A check is made to see if this file has been processed before. Since it is not, the sub-procedure Process_File (list), for example as depicted in Figure 18a and 18b, is then called to look for a "lookup" file for this file. If there is such a file, it is added to the lists together with the corresponding binary relationship type and the sub-procedure is called again. If no further "lookup" file can be found, the list is written to CDF B.

Let us now apply the above procedures on the exemplary personnel system below to generate the classes for this system.

| File | Items | Item Type | Key Type |
|---|---|---|---|
| EMPLOYEE | Emp_No | Numeric | Unique key |
| | Emp_Name | Character | Non-key |
| | Branch_No | Numeric | Non-key |
| | Race_Code | Character | Non-key |
| | Address | Character | Non-key |
| | Salary | Numeric | Non-key |
| MANAGER | Emp_No | Numeric | Unique key |
| | Co_Car_No | Character | Non-key |
| NON_MANAGER | Emp_No | Numeric | Unique key |
| | Union_M_No | Character | Non-key |
| BRANCH | Branch_No | Numeric | Unique key |
| | Branch_Name | Character | Non-key |
| | Br_Tot_Expenses | Numeric | Non-key |
| | Country_No | Numeric | Non-key |
| RACE_CODE | Race_Code | Character | Unique key |
| | Race_Desc | Character | Non-key |
| SKILLS | Emp_No | Numeric | Repeating key |
| | Skill | Character | Non-key |



This Sheet Has Been Replaced



| | | | | |
|---|---|---|---|---|
| 1 | PROJECT | Proj_No | Numeric | Unique key |
| 2 | | Proj_Name | Character | Non key |
| 3 | | Cust_No | Numeric | Non-key |
| 4 | | | | |
| 5 | BILLINGS | Emp_No | Numeric | Repeating key |
| 6 | | Proj_No | Numeric | Repeating key |
| 7 | | Month | Date | Non-key |
| 8 | | Amount | Numeric | Non-key |
| 9 | | | | |
| 10 | CUSTOMER | Cust_No | Numeric | Unique key |
| 11 | | Cust_Name | Character | Non-key |
| 12 | | | | |
| 13 | EXPENSES | Branch_No | Numeric | Repeating key |
| 14 | | Month | Date | Non-key |
| 15 | | Adv_Exp | Numeric | Non-key |
| 16 | | Pers_Exp | Numeric | Non-key |
| 17 | | | | |
| 18 | COUNTRY | Country_No | Numeric | Unique key |
| 19 | | Country_Name | Character | Non-key |

20

21 Using extended Semantics Extractor 12 of Figures 11 and 12,
22 the following binary relationships are derived:

23

| 24 Source File | Item | Target File | Item | Type |
|---|---|---|---|---|
| 25 EMPLOYEE | Emp_No | SKILLS | Emp_No | has_children |
| 26 EMPLOYEE | Emp_No | BILLINGS | Emp_No | has_children |
| 27 EMPLOYEE | Emp_No | MANAGER | Emp_No | has_subtype |
| 28 EMPLOYEE | Emp_No | NON_MANAGER | Emp_No | has_subtype |
| 29 BRANCH | Branch_No | EMPLOYEE | Branch_No | has_ward |
| 30 BRANCH | Branch_No | EXPENSES | Branch_No | has_children |
| 31 COUNTRY | Country_No | BRANCH | Country_No | inv. of pure_lookup |
| 32 RACE_CODE | Race_Code | EMPLOYEE | Race_Code | inv. of pure_lookup |
| 33 PROJECT | Proj_No | BILLINGS | Proj_No | has_children |
| 34 CUSTOMER | Cust_No | PROJECT | Cust_No | inv. of pure_lookup |

35

36 Also, the entity type of each file in SYSFILES is as follows:

37

| 38 File | Entity Type | Short Name of File |
|---|---|---|
| 39 EMPLOYEE | Kernel | K1 |
| 40 MANAGER | Subtype | S1 |
| 41 NON-MANAGER | Subtype | S2 |
| 42 BRANCH | Kernel | K2 |
| 43 RACE__CODE | Pure Lookup | L1 |



This Sheet Has Been Replaced



| | | |
|---|---|---|
| SKILLS | Characteristic | C1 |
| PROJECT | Kernel | K3 |
| BILLINGS | Associative | A1 |
| CUSTOMER | Pure Lookup | L2 |
| EXPENSES | Characteristic | C2 |
| COUNTRY | Pure Lookup | L3 |

Let us first apply the exemplary procedure of Figures 15a and 15b on the personnel system example. It first initializes a list to the first kernel entity in SYSFILES which is EMPLOYEE (K1). This means that it is going to generate Class Definition File A for the EMPLOYEE class. Next it uses a sorted SYSFILES and the Binary Relationship File 24 to find other entities to add to this list. The sorted SYSFILES contains files in descending order of subtypes, kernels, associatives, characteristics and pure lookups. The next entity added to the list is S1 which is MANAGER, with the items being Emp_No and the BR type being "has_subtype". This list is then written to CDF A.

The list is initialized again to K1 and S2 added to it, with the items being Emp_No and the BR type being "has_subtype", after which it is written to CDF A.

After the subtypes have been processed, the procedure initializes list K1 and search for those kernel entities that are "wards" of K1. However, K1 has no "wards" and so there are no such entities to add to the list.

Next the procedure searches for associative entities of K1. K1 has one associative entity, namely BILLINGS (A1) so A1 is added to the list, with the items being Emp_No and the BR type being "has_children". Next the procedure includes PROJECTS (K3) in this list as it constitutes the other entity that associates A1, with the items being Proj_No and the BR type being "inv_of_has_children". This list containing K1, A1, K3 is then written to CDF A.

After this, the procedure initializes the list to K1 again and search for characteristic entities of K1. K1 has one characteristic entity, namely SKILLS (C1), so C1 is added

This Sheet Has Been Replaced

to the list with the items being Emp_No and the BR type being "has_children". The list is then written to CDF A.

At this point CDF A contains the following lists:

- a list having K1, S1
- a list having K1, S2
- a list having K1, A1, K3
- a list having K1, C1

The procedure next generates the lists for the next kernel entity, namely BRANCHES (K2). K2 has no subtype and associative entities but it has (EMPLOYEE) K1 as its "ward". This produces the list K2, K1, with the items being Branch_No and the BR type being "has_wards". This list is written to CDF A. It also has EXPENSES (C2) as its characteristic entity. This produces the list K2, C2, with the items being Branch_No and the BR type being "has_children", which is also written to CDF A.

Finally, the list for the third and last kernel entity, namely PROJECTS (K3), is produced. However, there is only one list, namely

- a list having K3, A1, K1

since PROJECTS has an associative entity only which is BRANCH (A1), with K1 being the other entity that associates A1. In this list the items for K3 and A1 is Proj_No with the BR type being "has_children". The items for A1 and K1 is Emp_No with the BR type being "inv_of_has_children".

Let us next apply the exemplary procedure of Figure 17 on the personnel system example. It uses the lists of CDF A derived earlier to generate the lists for CDF B. Using the same personnel system it first initializes a list to the first entity in the first list of CDF A, namely K1. It then adds to this list entities that are lookup entities to K1. A lookup entity is a target file in a "pure_lookup" or "inv_of_has_ward" binary relationships, with K1 as the source file. K1 looks up on BRANCH (K2), so K2 is added to the list



This Sheet Has Been Replaced



with the items being Branch_No and the BR type being "inv_of_has_wards". Next a check is made on K2 to see if it too has lookup entities. K2 in fact has one, namely COUNTRY (L3). L3 is therefore added to the list with the items being Country_No and the BR type being "pure_lookup". At this point the list contains K1, K2, L3. Since L3 has no lookup entities, this list is written to the CDF B. Next the procedure returns to K2 to see if K2 has other lookup entities. Since it does not, the procedure returns to K1. It finds that K1 has another lookup entity, namely RACE_CODE (L1). The list containing K1, L3 with the items being Race_Code and the BR type being "pure_lookup", is then written to CDF B.

The next entity in the first list of CDF A is next processed. This entity is MANAGER (S1). It, however, does not have any lookup entities and so it is ignored.

As there is no more entity on the first list of CDF A, the first entity in the second list of CDF A is considered for processing. A check is made first to see if this entity has already been processed earlier. This entity is K1 which has been processed earlier and so it is ignored. The next entity is NON_MANAGER (S2) which has not being processed. However, it does not have any lookup entities and so it is ignored.

The above procedure is again applied for the next list of CDF A, which contains K1, A1, K3. As K1 has already been processed and A1 has no lookup entity, no list is produced for either of them. However, K3 (PROJECT) has a lookup entity, namely CUSTOMERS (L2). So the list K3, L2 is produced with the items being Cust_No and the BR type being "pure_lookup". It is written to CDF B. The next list of CDF A is K1, C1. However, since K1 has already been processed and C1 (SKILLS) has no lookup entities, both are ignored.

The above procedure is applied to the remaining lists in CDF A. In this example, only one list is produced, containing K2, L3 with the items being Country_No and the BR type being "pure_lookup".

This Sheet Has Been Replaced



Besides CDF A and CDF B, the Class Definition File also includes CDF C. CDF C includes a single list which contains all the pure lookup entities.

The CDF C for the personnel system example contains L1, L2, L3.

E-R Model Translator

E-R Model Translator 31 (Figure 11) is used to produce an Entity-Relationship (E-R) model for each class using CDF A, CDF B and Binary Relationships File 24. Many different embodiments of an E-R model can be produced. The following describes one example of a procedure to produce one embodiment of an entity-relationship (E-R) model of a class for all the classes except the class containing the pure lookup entities. This procedure begins by creating another file identical to SYSFILEITEMS. This duplicate file is called TEMPFILEITEMS. Next, for each "inverse_of_pure_lookup" binary relationships in Binary Relationship File 24, it inserts all the items of the file used as source file in this binary relationship except those items of the file that are used as source items into this TEMPFILEITEMS at the point which corresponds to the target items of this binary relationship. Next, for each "has_children", "has_wards", or "has_subtype" type of binary relationships in Binary Relationship File 24, it deletes those items in TEMPFILEITEMS that are used as target items in such binary relationships.

The procedure then gradually builds the E-R model for each class making use of the resultant TEMPFILEITEMS. It first include all the items of the root kernel entity of the class into the E-R model of the class. These items are obtained from TEMPFILEITEMS. Next it applies the following Decision Tables 1 and 2 on the CDF A and CDF B of the class to determine the relationship names between two adjacent entities in the class which are not pure lookup entities.

This Sheet Has Been Replaced

| Decision Table 1 | | | | |
|---|---|---|---|---|
| To define relationship names between two adjacent entities in the CDF A lists | | | | |
| Rule | From Entity | To Entity | BR Type | Relationship Name |
| 1 | K | S | has_subtype | K is a S |
| 2 | K | S | has_wards | K has S |
| 3 | K | K' | has_wards | K has K' |
| 4 | X | C | has_children | X has C |
| 5 | X | A | has_children | a. X has Y<br>b. X and Y have A<br>(Y is another entity that associates A) |

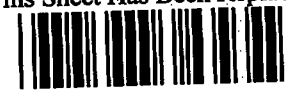
| Decision Table 2 | | | | |
|---|---|---|---|---|
| To define relationship names between two adjacent entities in the CDF B lists | | | | |
| Rule | From Entity | To Entity | BR Type | Relationship Name |
| 1 | X | C or A | inv_of_has_wards | X references C or A |
| 2 | K' | K | inv_of_has_wards | K' belongs to K |
| 3 | C or A | K' | inv_of_has_wards | C or A references K |
| 4 | C or A | K | inv_of_has_children | C or A belongs to K |

Legend:
- X - kernel, subtype, characteristic, or associative entity
- C - characteristic entity
- A - associative entity
- K,K' - kernel entities
- Y - other entity that associates associative entity

For each relationship name identified using the above tables, the relationship name and the items of the file corresponding to the second entity of the two adjacent entities that establish this relationship name are included



This Sheet Has Been Replaced



in the E-R model of the class. The items included are obtained from TEMPFILEITEMS. This E-R model is then stored in the E-R Model of Class File 32.

Let us now illustrate this procedure by applying it on the exemplary personnel system described earlier. We know at this stage that this personnel system has three classes which are derived from the kernel entity of EMPLOYEE, BRANCH, and PROJECT. Let us call these three classes:

- ABOUT EMPLOYEE
- ABOUT BRANCH
- ABOUT PROJECT

If we were to apply the procedure, we should get the following E-R models for the three classes:

ABOUT EMPLOYEE

Emp_No
Emp_Name
Race_Code
Race_Desc
Address
Salary
<EMPLOYEE belongs to BRANCH>
    Branch_No
    Branch_Name
    Br_Tot_Expenses
    Country_Code
    Country_Name
<EMPLOYEE is a MANAGER>
    Co_Car_No
<EMPLOYEE is a NON_MANAGER>
    Union_M_No
<EMPLOYEE has PROJECT>
    Proj_No
    Proj_Name
    Cust_No
    Cust_Name



This Sheet Has Been Replaced



```
                <EMPLOYEE and PROJECT have BILLINGS>
                    Month
                    Amount
                <EMPLOYEE has SKILLS>
                    Skill

            ABOUT BRANCH
                Branch_No
                Branch_Name
                BR_Tot_Expenses
                Country_Code
                Country_Name
                <BRANCH has EMPLOYEE>
                    Emp_No
                    Emp_Name
                    Race_Code
                    Race_Desc
                    Address
                    Salary
                <BRANCH has EXPENSES>
                    Month
                    Adv_Exp
                    Pers_Exp

            ABOUT PROJECT
                Proj_No
                Proj_Name
                Cust_No
                Cust_Name
                <PROJECT has EMPLOYEE>
                    Emp_No
                    Emp_Name
                    Race_Code
                    Race_Desc
                    Address
                    Salary
                <EMPLOYEE belongs to BRANCH>
                        Branch_No
```



This Sheet Has Been Replaced



Branch_Name
Br_Tot_Expenses
Country_Code
Country_Name
<PROJECT and EMPLOYEE have BILLINGS>
Month
Amount

Let us now explain how these E-R models are produced when the procedure is applied. First another file identical to SYSFILEITEMS is created. This file is called TEMPFILEITEMS. For each "inverse_of_pure_lookup" binary relationship in Binary Relationship File 24, the procedure inserts in TEMPFILEITEMS, at the point corresponding to the target item, all the items of the source file except the source item itself, e.g. there is an "inverse_of_pure_lookup" binary relationship as follows.

| Source File | Item | Target File | Item |
|---|---|---|---|
| RACE_CODE | Race_Code | EMPLOYEE | Race_Code |

The items of the source file, namely RACE_CODE, are Race_Code and Race_Desc. The procedure inserts only the item Race_Desc (leaving out Race_Code as it is a source item) at Race_Code of EMPLOYEE of TEMPFILEITEMS, so that the items of EMPLOYEE in TEMPFILEITEMS becomes:

Emp_No
Emp_Name
Branch_No
Race_Code
Race_Desc
Address
Salary

Next the procedure deletes those items in TEMPFILEITEMS that correspond to the target item in "has_children", "has_wards" and "has_subtype" binary relationships in Binary



This Sheet Has Been Replaced

Relationship File 24. For example, Branch_No of EMPLOYEE in the exemplary personnel system is a target item in a "has_wards" binary relationship with BRANCH. Thus, this item in the TEMPFILEITEMS file is deleted.

The resultant TEMPFILEITEMS file is then used together with the two decision tables earlier described to produce the above E-R model for each class which is then stored in the E-R Model of Classes File 32.

Let us show how the procedure produces the E-R model for the ABOUT EMPLOYEE class. First all items of EMPLOYEE (K1) obtained from TEMPFILEITEMS are included into the E-R model of this class. Next it reads CDF B to find records having K1 as the first file. The first such record is a list containing K1, K2, L3. The next file in this list is BRANCH (K2) which has a BR type of "inv_of_has_wards" with EMPLOYEE (K1). As K1 and K2 are kernel entities with the BR type of "inv_of_has_wards" the procedure applies rule 2 of Decision Table 2 to derive the following relationship name:

<EMPLOYEE belongs to BRANCH>

This relationship name together with the items of BRANCH obtained from TEMPFILEITEMS are then included into the E-R model of ABOUT EMPLOYEE.

The next file after K2 in the above CDF B record is L3. As L3 is a pure lookup entity the procedure ignores it and proceeds to read in the next record of CDF B having K1 as the first file. The record is a list containing K1, L1. However, since the next file in this list, namely RACE_CODE (L1) is a pure lookup entity, the procedure ignores it.

As there are no more records in CDF B having K1 as the first file, the procedure starts to read the CDF A file. The first record of CDF A is a list containing K1, S1. As S1 is a subtype entity with a BR type of "has_subtype" to K1, the procedure applies rule 1 of Decision Table 1 to derive the relationship name:

<EMPLOYEE is a MANAGER>



This Sheet Has Been Replaced



This relationship name together with the items of MANAGER obtained from TEMPFILEITEMS are then included in the E-R model of ABOUT EMPLOYEE class.

Next, the procedure reads the CDF B file to look for records containing S1 as the first file. However, there are no such records. It then proceeds to read in the next record of CDF A. This record contains K1, S2. Using the same rule as applied to S1 above, the procedure derives the following relationship name:

<EMPLOYEE is a NON-MANAGER>

This relationship name together with the items of NON-MANAGER obtained from TEMPFILEITEMS are then included in the E-R model of ABOUT EMPLOYEE class.

The procedure next reads the CDF B file to look for records containing S2 as the first file. However, there are no such records. It proceeds to read in the next record of CDF A. This record contains K1, A1, K3. As A1 (BILLINGS) is an associative entity and K3 (PROJECTS) is the other entity that associates A1, the procedure first applies rule 5a of Decision Table 1 using K1 and K3 to derive the following relationship name:

<EMPLOYEE has PROJECTS>

This relationship name together with the items of PROJECTS obtained from TEMPFILEITEMS are then included into the E-R model of ABOUT EMPLOYEE class. Next the procedure reads CDF B file to look for records with K3 as the first file. There is one such record, namely K3, L2. However, as L2 (CUSTOMER) is a pure lookup entity, the record is ignored. The procedure then applies rule 5b of Decision Table 1 on the current CDF A record, namely the list containing K1, A1, K3, to derive the following relationship name:

<EMPLOYEE and PROJECT have BILLINGS>



This Sheet Has Been Replaced



This relationship name together with items of BILLINGS obtained from TEMPFILEITEMS are then included in the E-R model of ABOUT EMPLOYEE Class.

The procedure next reads the CDF B file to look for records with A1 as the first file. However, there is no such file. It then proceeds to read in the next CDF A record. This record contains K1, C1. Applying rule 4 of Decision Table 1, the procedure derives the following relationship name

<EMPLOYEE has SKILLS)

This relationship name together with the items of SKILLS obtained from the TEMPFILEITEMS file are then included in the E-R model of ABOUT EMPLOYEE class.

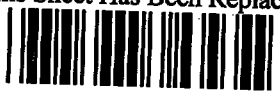
The procedure next reads the CDF B file to look for records having C1 as the first file. However, there are none and so it proceeds to read the next CDF A record. However, there are no more CDF A records. This ends the E-R model translation for the ABOUT EMPLOYEE class from its class definition.

Besides the procedure just described, the E-R Model Translator also has another procedure which creates a class using CDF C to contain all the pure lookup entities. For the personnel system example, this procedure creates the following class:

```
ABOUT PURE LOOKUP ENTITIES
        <RACE_CODE>
        Race_Code
        Race_Desc
        <CUSTOMER>
        Cust_No
        Cust_Name
        <COUNTRY>
        Country_No
        Country_Name
```



This Sheet Has Been Replaced



Extension To Existing Modules

Besides extending Semantics Extractor 12, in one embodiment Model Purifier 26, and/or Report Item Selector 16 are also extended. The extension of Model Purifier 26 provides an interface to allow a user to specify, for two opposite binary relationships of UU (unique key to unique key), which binary relationship is the "has_subtype" and which is the "inverse of has_subtype" type, if a file has only one sub-type. However, this is not necessary for embodiments using the SQL language because it is possible to know this from the primary and foreign keys.

The extension of Report Item Selector 16 provides an interface that displays all or selected classes and their E-R model, as well as allowing a user to formulate a query by picking the desired class attributes from the E-R model of the classes.

Model purifier 26 and Security Model Specifier 29 are optional modules to this embodiment.

In conclusion, this embodiment provides an alternative way of presenting the database items for end-users to select to formulate their queries. This presentation of using classes and their E-R models is more intuitive and meaningful than a single list of items. As a result, this embodiment makes it easier for end-users to formulate relational database queries.

INTEGRATION WITH ELECTRONIC MAIL

In one embodiment of this invention, the end-user query facility is integrated with an electronic mail (E-mail) system to allow a user located at a site with no on-line access to the application database but with connection to an E-mail system to still be able to make a query of the application database. The E-mail system provides the delivery mechanism for the user's query to be transmitted to the site where the query will be processed, and for the query results to be returned to the user after processing. The integration with the E-mail system also provides one convenient method for all queries made via the E-mail system



This Sheet Has Been Replaced



to be logged into a log file. Information such as user id, items selected by user, the number of records extracted from database, the time of query, and the time taken to process the query can all be logged.

Figure 19 shows one example of this embodiment. User A at a site with no on-line access to the Application Database 39 interfaces with Report Item Selector 16 of Information Scout 15 to select the items to be reported. Note that in this embodiment, the Information Scout consists only of the Report Item Selector which interfaces with the Send-Mail Agent instead of the Inference Engine 17. The Send-Mail Agent 41 then takes the items selected and stores them into a file which is then attached to an E-mail. This E-mail is then sent to the Query Mailbox 40. The Open-Mail Agent 42 periodically checks for E-mail in the Query Mailbox 40. If there is an E-mail, it reads the E-mail to get the file attached. This file contains the items selected by the User. A. It then passes the items selected to the Inference Engine. Note that in this embodiment the Inference Engine 17 interfaces with the Open-Mail Agent 42 instead of the Report Item Selector 16. The Inference Engine then identifies one or more database files which contain the desired information and searches the Knowledge Base 14 to determine the linkages connecting the identified files. Program Generator 18 then generates a Source Program 19 based on the linkages inferred. Source Program 19 is then compiled and executed against the Application Database by the Compiler/Executor 20. The query result obtained is then composed into an E-mail by the Open-Mail Agent 42 and sent to User A Mailbox 43. User A obtains his query result by reading this E-mail.

The following describes in greater detail the Send-Mail Agent 41 and the Open-Mail Agent 42 based on the integration with Microsoft Mail, a product of Microsoft Corporation, USA.

The procedure for the Send-Mail Agent 41 is given as a representative section of code in Program Listing 1. It is written in Microsoft's Visual Basic under Microsoft's Windows operating system and uses the Microsoft's Electronic Forms Designer as well as the Microsoft's Messaging Application



This Sheet Has Been Replaced



Program Interface (MAPI) to interface with the Microsoft Mail E-Mail System.

The procedure works as follows. After User A has selected his items using the Report Item Selector 39 which has been implemented using Microsoft's Electronic Forms Designer, the Send-Mail Agent 41 is invoked. This Agent first stores the items selected into a temporary file called query.dat which is then assigned to another file called FName. Next a function called WriteMessage is called to attach this file to an E-Mail created by Report Item Selector 39. The WriteMessage function is one of the functions provided by the Microsoft's Electronic Forms Designer. This function uses the command MEFAddAttachment ( ) to attach the FName file to the E-Mail. This E-Mail is then posted to the Query Mailbox 40 by using the command MAPISendMail. This completes the description of Send-Mail Agent 41.

The procedure for Open-Mail Agent 42 is given as a representative section of code in Program Listing 2. It is written in C language under Microsoft's Windows operating system and also uses Microsoft Mail's Messaging Application Program Interface (MAPI) to interface with Microsoft Mail E-Mail system.

This procedure works as follows. It periodically checks Query Mailbox 40 of the E-Mail system using the subroutine vTimerServeProc( ) running under subroutine WinProc( ). VTimerServeProc( ) in turn uses the MAPI command MAPIFindNext to find out whether there is any E-Mail in the mailbox. If there is, it then issues the MAPI command MAPIReadMail to read the E-Mail. At the same time it logs the time the message is read into the server log file (which is one of the two files under Log File 44, the other being queue log file) by calling the sub-routine vServerLog. It next processes the attachment in the E-Mail. The attachment is a file where the items selected by a user as his query is stored. This file is called query.dat. vTimerServerProc next calls vServeAttach to process this attachment. vServeAttach first logs the user name, the subject and the date received into the server log file. It then calls Inference Engine 17 which



This Sheet Has Been Replaced



is implemented as a sub-routine called 1AppPHEng. This sub-routine also contains Program Generator 18 to generate the Powerhouse Quiz source program based on the linkages determined by Inference Engine 17. It next uses Compiler/Executor 20 to get the query result from the database. Compiler/Executor 20 is implemented as a sub-routine called 1AppPHRet. The query result is stored in a file called RESULT.XXX where XXX can be XLS, MDB, TXT, DBF which are the file extensions for the different file formats available. Next the vServeAttach logs the number of rows of records retrieved from the database that make up the query result. It also logs the selection filter used in the user query as well as the items selected into the queue log file of Log File 44 using the sub-routine vQueueLog. Lastly, it composes an E-Mail to contain the query result and then use the MAPISendMail command to send this E-Mail to User A Mailbox 43. This completes the description of the Open-Mail Agent.

Thus by integrating the query facility to an E-Mail system, a user who has no on-line access to his application database is still able to make his query using the E-Mail system. Another advantage of this embodiment is that all queries made can be logged for historical analysis. Furthermore, the embodiment allows queries to be processed in batch mode during off-peak hours whenever there is excessive load on the database system during peak hours.

ALTERNATIVE WAY OF PRESENTING QUERY RESULTS TO HELP PREVENT MISINTERPRETATION

This alternative embodiment provides an alternative way in which query results can be presented to help prevent users from misinterpreting their query results. Query results can possibly be misinterpreted by users because of their complexity, which is in turn due to the queries being complex. A complex query is one which is made up of many basic queries with each basic query having its own distinct result. What a user gets when he makes a complex query is a report in which these distinct basic query results have been



This Sheet Has Been Replaced



compounded. This compound report is not always easy for the user to understand or interpret. This embodiment breaks down a complex query into its basic components and processes each basic component (which is a basic query) separately so that what a user finally receives as a result is a disjoint set of distinct basic query results which the user can easily understand or interpret.

This alternative embodiment involves adding a new module called the Knowledge Thread Analyser 50 to the query facility of Figure 14 described earlier. An exemplary main flowchart for this alternative embodiment is shown in Figure 40.

Knowledge Thread Analyser

Before we describe the function of the Knowledge Thread Analyser 50, let us explain how query results can be misinterpreted by illustrating two different cases.

**Case A:**

Suppose we have two database files as follows:

| **File** | **Item** | **Key** |
|---|---|---|
| INVOICE | Inv-No | Unique key is Inv-No |
| | Inv-Amt | |
| | Customer | |
| INVOICE_LINES | Inv-No | Repeating key is Inv-No |
| | Part-No | |
| | Qty | |
| | Price | |

Suppose also that INVOICE has only one record as follows:

| **Inv-No** | **Inv-Amt** |
|---|---|
| 1 | $1000 |

This Sheet Has Been Replaced

and INVOICE_LINES has 3 records as follows:

| Inv-No | Part-No | Qty | Price |
|---|---|---|---|
| 1 | A | 2 | $ 50 |
| 1 | B | 3 | $100 |
| 1 | C | 7 | $175 |

Suppose a user picks the items Inv-No, Inv-Amt, Part-No, Qty and Price as his query. The following query result would be produced after the query is processed.

| Inv-No | Customer | Inv-Amt | Part-No | Qty | Price |
|---|---|---|---|---|---|
| 1 | IBM | $1000 | A | 2 | $ 50 |
| 1 | IBM | $1000 | B | 3 | $100 |
| 1 | IBM | $1000 | C | 7 | $175 |

This result is correct but can be misinterpreted by a user because the Inv-Amt of $1,000 is repeated for every record of INVOICE_LINES. A user may total up the Inv-Amt to get what he thinks is the total invoice amount, namely $3000. This is wrong as there is only one invoice amount which is $1000.

This query is complex and can be broken down into two basic queries. One basic query is about finding out the amount for each invoice for each customer. The items Inv-No, Customer and Inv-Amt from the INVOICE file would provide this information. The other basic query is about finding out the details of each invoice, which details can be obtained from the items Inv-No, Customer, Part-No, Qty, and Price. The above query result is a compound report of these two basic queries. However, if both these basic queries were processed separately we would get a disjoint set of basic query results as follows:

This Sheet Has Been Replaced

| Inv-No | Customer | Inv-Amt | Part-No | Qty | Price |
|---|---|---|---|---|---|
| 1 | IBM | $1000 | | | |
| 1 | IBM | | A | 2 | $ 50 |
| 1 | IBM | | B | 3 | $100 |
| 1 | IBM | | C | 7 | $175 |

Notice that with this disjoint set of basic query results the Inv-Amt is now correctly presented as a single value of $1000 and not a repeating value of $1000.

This case can be identified by the fact that the query involves two files in which they have a "one to many" relationship. Alternatively, we can say that their binary relationships is a UR (unique to repeating) or UN (unique to non-key). We can also say that their binary relationship is a "has_children" or "has_wards" type. Also, one of the items selected is from the file whose unique key is used in the binary relationship, this item being non-key and numeric. For the example above, INVOICE is the file whose unique key Inv-No is used in the binary relationship with INVOICE_LINES. Inv-Amt is one of the items selected as part of the query and it is non-key and numeric.

Consider another case which can also lead to misinterpretation by users.

**Case B:**

Suppose we have three database files as follows:

| File | Item | Key |
|---|---|---|
| EMPLOYEE | Emp_No | Unique key is Emp_No |
| | Emp_Name | |
| SKILLS | Emp_No | Repeating key is Emp_No |
| | Skill | |

This Sheet Has Been Replaced



| | | | |
|---|---|---|---|
| PAYS | Emp_No | Repeating key is Emp_No | |
| | Month | | |
| | Salary | | |

Suppose also that these three files contain the following records:

| | | | |
|---|---|---|---|
| **EMPLOYEE:** | **Emp_No** | **Emp_Name** | |
| | 1 | John | |
| | 2 | Sally | |
| **SKILLS:** | **Emp_No** | **Skill** | |
| | 1 | COBOL | |
| | 1 | Fortran | |
| | 1 | Ada | |
| | 2 | C++ | |
| | 2 | COBOL | |
| **PAYS:** | **Emp_No** | **Month** | **Salary** |
| | 1 | Jan | 1000 |
| | 1 | Feb | 1000 |
| | 2 | Jan | 2000 |
| | 2 | Feb | 2000 |

Suppose a query is formulated by picking the items Emp_Name, Skill, Month, and Salary. Let us first consider the case of using PowerHouse to process this query, followed by the case of using SQL to process this query. The reason for doing this is that both PowerHouse and SQL give different results, but each may still lead to misinterpretation by users. The following result would be produced after this query is processed by PowerHouse.

| **Emp_Name** | **Skill** | **Month** | **Salary** |
|---|---|---|---|
| John | COBOL | Jan | 1000 |
| John | Fortran | Feb | 1000 |
| John | Ada | | |
| Sally | C++ | Jan | 2000 |



This Sheet Has Been Replaced



| | | | |
|---|---|---|---|
| Sally | COBOL | Feb | 2000 |

A user may then use this result to find out the all Salary records of each employee with COBOL skill. As the first and fifth row of the query result contain COBOL, he would think that the Salary records come from these two rows only, namely:

| Emp_Name | Month | Salary |
|---|---|---|
| John | Jan | 1000 |
| Sally | Feb | 2000 |

However, this is not totally correct as the correct result should include all the Salary records of John and Sally both of whom have COBOL skill. The correct result is as follows:

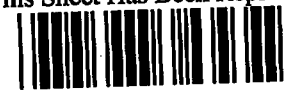
| Emp_Name | Month | Salary |
|---|---|---|
| John | Jan | 1000 |
| John | Feb | 1000 |
| Sally | Jan | 2000 |
| Sally | Feb | 2000 |

Let us now consider the case of using SQL to process the same query involving Emp_Name, Skill, Month and Salary. The following result would be produced by SQL:

| Emp_Name | Skill | Month | Salary |
|---|---|---|---|
| John | COBOL | Jan | 1000 |
| John | COBOL | Feb | 1000 |
| John | Fortran | Jan | 1000 |
| John | Fortran | Feb | 1000 |
| John | Ada | Jan | 1000 |
| John | Ada | Feb | 1000 |
| Sally | C++ | Jan | 2000 |
| Sally | C++ | Feb | 2000 |
| Sally | COBOL | Jan | 2000 |



This Sheet Has Been Replaced



| | | | |
|---|---|---|---|
| Sally | COBOL | Feb | 2000 |

This result could be misinterpreted by a user because the Salary of John and Sally is repeated many times. He may total up the Salary column to find the total Salary for John and Sally. What he would get respectively is 6000 and 8000. This is incorrect. The correct result is 2000 and 4000, respectively since John's salary is 1000 each in Jan and Feb and Sally's salary is 2000 each in Jan and Feb.

This query is complex and involves two multi-valued dependencies of EMPLOYEE, namely SKILLS and PAYS. It can be broken down into two basic queries. One basic query is about employees and their skills and the other basic query is about employees and their pays. The different query results by PowerHouse and SQL shown above are compound reports of these two basic queries. On the other hand the disjoint set of results for these two basic queries are:

| Emp_Name | Skill | Month | Salary |
|---|---|---|---|
| John | COBOL | | |
| John | Fortran | | |
| John | Ada | | |
| Sally | C++ | | |
| Sally | COBOL | | |
| | | | |
| John | | Jan | 1000 |
| John | | Feb | 1000 |
| Sally | | Jan | 2000 |
| Sally | | Feb | 2000 |

Notice now that with this report a user could easily find out correctly all salary records of employees who have COBOL skill. Also, with these two disjointed basic reports, the user would not incorrectly total up the salary of John and Sally.

This case can be identified by the fact that the query involves more than one multi-valued dependencies (MVD). For



This Sheet Has Been Replaced



example, in the above case the attribute or item SKILL is multi-dependent on the attribute or item Emp_No, i.e. each employee has a well-defined set of skills. Similarly, the attributes or items Month and Salary are also multi-dependent on Emp_No. Alternatively, we can say that it involves at least three files in which one file has a "one-to-many" relationship or a UR (unique-to-repeating) or a UN (unique to non-key) binary relationship with each of the other files or in other words, one file has a "has_children" or a "has_wards" binary relationship with each of the other files.

Let us now describe Knowledge Thread Analyser 50. It takes as its input the knowledge thread (as defined earlier) determined by Inference Engine 17 based on the items selected by the user from a single class. Since the items selected by a user are obtained from a single class, since every knowledge thread determined by Inference Engine 17 has as its thread-head the root kernel entity of the class from which the items are selected. Knowledge Thread Analyser 50 then analyses this knowledge thread, breaking it down first into simple knowledge threads to resolve the Case B type of complex query problems, and second for each simple knowledge thread derived breaking it down further into smaller simple knowledge threads to resolve the Case A type of complex query problems. The new knowledge threads produced as a result of this analysis are then passed to Program Generator 18 to generate the corresponding source programs which are then compiled and executed to produce the query results.

Before we describe in detail this exemplary procedure, let us recap what a knowledge thread is by giving an example, because it will help in understanding this procedure. Suppose a user makes a query on the ABOUT EMPLOYEE class described earlier by selecting the items Emp_Name, Salary, Branch_Name, Br_Tot_Expenses, Country_Name, Skill, Month and Amount. The knowledge thread determined by Inference Engine 17 is as follows:

This Sheet Has Been Replaced



EMPLOYEE → BRANCH (NU) → COUNTRY (NU)
→ SKILLS (UR)
→ BILLINGS (UR)

Legend: NU - non-key to unique key
UR - unique key to repeating key

This knowledge thread has one thread-head but many thread-ends, with the thread-head being the root kernel entity of the class. What we want to do is to breakdown this complex thread into many simple threads, where each simple thread has one thread-head and only one thread-end. This complex knowledge thread can be broken down into three simple knowledge threads as follows:

- EMPLOYEE → BRANCH (NU) → COUNTRY (NU)
- EMPLOYEE → SKILLS (UR)
- EMPLOYEE → BILLINGS (UR)

Two different types of simple threads can be derived from a complex thread determined by Inference Engine 17. One type is the NU or RU threads in which all the binary relationships between two adjacent files in the thread is NU or RU. Another type is the UR threads in which all the binary relationships between two adjacent files is UR. One characteristic of the UR simple threads is that their thread-head is same as the thread-head of the complex thread. As for the NU or RU simple threads, their thread-head is the file on the complex thread which starts off the NU or RU relationship.

For example, the above complex knowledge thread has one NU simple thread, namely

- EMPLOYEE → BRANCH (NU) → COUNTRY (NU)

This Sheet Has Been Replaced



and two UR simple threads, namely

- EMPLOYEE → SKILLS (UR)

- EMPLOYEE → BILLINGS (UR)

We shall now describe an exemplary embodiment of Knowledge Thread Analyser 50 in detail. It comprises the following steps:

Step 1: Analyze the knowledge thread determined by Inference Engine 17 and derive the NU or RU simple knowledge threads and the UR simple knowledge threads. Store the NU or RU simple knowledge threads in NU-RU Thread File and the UR simple knowledge threads in UR Thread File. This resolves the Case B type of complex query problem.

Step 2: Access the first simple knowledge thread in NU-RU Thread File. Starting from the thread-end analyze each pair of adjacent files for Case A type of complex query. If a pair has a Case A type of complex query, generate a new knowledge thread comprising files from the thread-end through to the first file in the pair of adjacent files being analyzed, this first file forming the thread-head of the new knowledge thread. Assign to this new knowledge thread its respective set of user selected items such that only its thread-head has user selected items which are both non-key and numeric. Store this new knowledge thread in New Thread File. Repeat this step for each of the remaining simple threads in NU-RU Thread File.



This Sheet Has Been Replaced



Step 3: Access the first simple knowledge thread in UR Thread File. Starting from the thread-head analyze each pair of adjacent files for Case A type of complex query. If a pair has a Case A type of complex query, generate a new knowledge thread comprising files from the thread-head through to the first file in the pair of adjacent files being analyzed, with the thread-head of the new knowledge thread being the same as the thread-head of the knowledge thread being analyzed. Assign to this new knowledge thread its respective set of user-selected items such that only its thread-end has user selected items which are both non-key and numeric. Store this new knowledge thread in New Thread File. Repeat this step for each of the remaining simple threads in UR Thread File.

Step 4: For each simple knowledge thread in UR Thread File combine it with those NU or RU simple threads in the NU-RU Thread File which have their thread-head matching a file in the UR simple knowledge thread. This complex thread becomes a new knowledge thread. Assign this new knowledge thread its respective set of user selected items such that only its UR thread-end has user-selected items which are both non-key and numeric. Store this new knowledge thread in New Thread File.

Step 5: Eliminate duplicate knowledge threads in the New Thread File.

Let us apply this procedure on the earlier example of a query on the ABOUT EMPLOYEE class to show how the above procedure works. When we apply step 1, we get the following NU simple knowledge threads



This Sheet Has Been Replaced



. EMPLOYEE → BRANCH (NU) → COUNTRY (NU)

and the following UR simple knowledge threads:

. EMPLOYEE → SKILLS (UR)
. EMPLOYEE → BILLINGS (UR)

These simple threads are stored in the NU-RU Thread File and the UR Thread File, respectively. Next we use step 2 on the NU simple knowledge thread. The first pair of adjacent files starting from the thread-end are COUNTRY and BRANCH. Since the user-selected item from COUNTRY file is Country_Name which is not numeric, there is no Case A type of complex query in this pair of adjacent files. So no new knowledge thread is generated for this pair. The next pair comprises BRANCH and EMPLOYEE. Since one of the user-selected items from BRANCH is Br_Tot_Expenses which is both non-key and numeric, there is a Case A type of complex query in this pair. The following new knowledge thread is generated as a result:

. BRANCH → COUNTRY (NU)

The user-selected items assigned to it are Country_Name, Branch_Name, and Br_Tot_Expenses. Notice that only the thread-end, namely BRANCH, has the item which is both non-key and numeric, namely Br_Tot_Expenses. This new Knowledge Thread is then stored in the New Thread File.

As there are no more NU simple knowledge thread, we next apply step 3 on the UR simple knowledge threads. The first UR simple knowledge thread is

. EMPLOYEE → SKILLS (UR)

There is only pair of adjacent files in this thread. Since one of the user-selected items from EMPLOYEE is Salary which is both non-key and numeric, there is a Case A type of



This Sheet Has Been Replaced



complex query in this pair. The following new knowledge thread is generated as a result:

. EMPLOYEE

The user-selected items assigned to it are Emp_Name and Salary. This new knowledge thread is then stored in the New Thread File.

The next UR simple knowledge thread is

. EMPLOYEE → BILLINGS (UR)

For this pair of files, the following new knowledge thread is also generated:

. EMPLOYEE

with Emp_Name and Salary being assigned to it. It is then stored in the New Thread File.

Next we apply step 4. We combine the NU simple thread EMPLOYEE → BRANCH (NU) → COUNTRY (NU) with the UR simple knowledge thread EMPLOYEE → SKILLS (UR) to derive the following new knowledge thread:

EMPLOYEE → BRANCH (NU) → COUNTRY (NU)
→ SKILLS (UR)

The user-selected items assigned to it are Emp_Name, Branch_Name, Country_Name and Skill. This new knowledge thread is then stored in the New Thread File.

Another new knowledge thread is also formed as follows:

EMPLOYEE → BRANCH (NU) → COUNTRY (NU)
→ BILLINGS (UR)



This Sheet Has Been Replaced



The user-selected items assigned to it are Emp_Name, Branch_Name, Country_Name, Month, and Amount. This new knowledge thread is then stored in the New Thread File. Notice that only the UR thread-end, namely BILLINGS, has the user-selected item which is both non-key and numeric, namely Amount.

Step 5 is applied next. In this step the duplicate thread, namely EMPLOYEE, in the New Thread File is eliminated. This completes the analysis of the complex knowledge thread determined by the Inference Engine 17 by the Knowledge Thread Analyser 50.

In this embodiment Model Purifier 26 and Security Model Specifier 29 are optional modules.

Thus this embodiment ensures that end-users will not misinterpret their query results whenever they make complex queries as it ensures that the query results produced are presented as a disjointed set of basic query results.

The invention now being fully described, it will be apparent to one of ordinary skill in the art that many changes and modifications can be made thereto without departing from the spirit or scope of the appended claims.



This Sheet Has Been Replaced



This Sheet Has Been Replaced

WHAT IS CLAIMED IS

1. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of classes and a set of linkages of the database model;

a class generator for reading said database model and generating said set of classes and said set of linkages of the database model and which stores in said knowledge base said set of classes and said set of linkages;

an information scout for interfacing with a user to obtain from the user choices based on said classes as a designation of the information to be extracted from said database;

an inference engine which, based upon said designation of information to be extracted from said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files; and

a program generator which accesses the linkages obtained by said inference engine and generates a program to extract said desired information from said database.

2. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of classes and a set of linkages of the database model;

a class generator for reading said database model and generating said set of classes and said set of linkages of the database model and which stores in said knowledge base said set of classes and said set of linkages;

an information scout for interfacing with a user to obtain from the user choices based on said classes



* C L M S *

as a designation of the information to be extracted from said database;

an inference engine which, based upon said designation of information to be extracted from said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files;

a knowledge thread analyzer receiving as its input said linkages determined by said inference engine, and which breaks down said linkage(s) into simple linkage(s); and

a program generator which accesses said simple linkages obtained by said knowledge analyzer and generates programs, one for each said simple linkage, to extract said desired information from said database as a plurality of simple query results, one for each generated program.

3. An end user query facility for accessing a database having a plurality of database files formed using a database model, comprising:

a knowledge base which stores a set of linkages of the database model;

a semantics extractor for reading said database model and extracting the semantics of said database model, and which stores in said knowledge base said set of linkages,

an information scout for interfacing with a user to obtain from the user a designation of the information to be extracted from said database,

an inference engine which, based upon said designation of information to be extracted from said database, identifies one or more of said database files which contain the desired information and searches said knowledge base to determine the linkage(s) connecting said one or more identified files;

a program generator which accesses the linkages obtained by said inference engine and generates a program to extract said desired information from said database;

This Sheet Has Been Replaced







a first e-mail agent for said information scout to interface with an e-mail system, said first e-mail agent being used to post said designation of information to be extracted from said database to the mailbox of said inference engine; and

a second e-mail agent for the said inference engine to interface with said e-mail system, said second e-mail agent being used to access said mailbox of said inference engine to obtain said designation of information to be extracted from said database.

4. An end-user query facility as in claim 42 wherein said security model comprises:

item security which specifies a set of database files and items selected from said database which a user or class of users can access; and

value security which specifies a set of conditions to restrict said user or class of users to a certain range(s) of values of said database.

5. An end user query facility as in claim 1 wherein each said class comprises a data model of a subset of database files of said database model, said data model comprising a tree.

6. An end user query facility as in claim 1 wherein said tree of said class has a root comprising a database file of kernel entity type and a plurality of branches each comprising a linkage of one or more database files from said subset of database files

7. An end-user query facility as in claim 6 wherein said class generator comprises:

a semantics extractor to derive a set of binary relationships and to derive the entity type of each database file of said database model;

This Sheet Has Been Replaced



* C L M S *

a means to derive said set of classes using said set of binary relationships and said entity type of each database file of said database model.

8. An end-user query facility as in claim 7 wherein said entity type is selected from a set of entity types which comprise a kernel entity type, a subtype entity type, a characteristic entity type, an associative entity type or a pure lookup entity type.

9. An end-user query facility as in claim 8 wherein said binary relationship comprises a linkage between two files in said database model in which one file has a repeating or foreign key that references the unique or primary key of the other file.

10. An end user query facility as in claim 9 wherein said set of binary relationships comprises "has_children", "has_wards", "inverse_ of_pure_lookup" or "has_subtype" binary relationships.

11. An end user query facility as in claim 10 wherein said set of binary relationships further comprises "inverse_of_"has_children", "inverse_of_has_wards", "pure_lookup" or "inverse_of_has_subtype" binary relationships.

12. An end user query facility as in claim 10 wherein said "has_children" binary relationship is one in which the source file is not a pure lookup entity type and has a one-to-many relationship with the target file, and the target file does not have its own independent unique or primary key.

13. An end user query facility as in claim 11 wherein said "inverse_of_has_children" binary relationship is one in which the target file is not a pure_lookup entity type and has a one-to-many relationship with the source file, and the

This Sheet Has Been Replaced







source file does not have its own independent unique or primary key.

14. An end user query facility as in claim 10 wherein said "has_wards" binary relationship is one in which its source file is not a pure lookup entity type and has a one-to-many relationship with its target file, and its target file has its own independent unique key or primary key.

15. An end user query facility as in claim 11 wherein said "inverse_of_has_wards" binary relationship is one which the target file is not a pure lookup entity type and has a one-to-many relationship with the source file, and the source file has its own independent unique or primary key.

16. An end user query facility as in claim 10 wherein said "has_subtype" binary relationship is one in which its source file has a one-to-one relationship with its target file and the target file is a subtype entity of the source file.

17. An end user query facility as in claim 11 wherein said "inverse_of_has_subtype" binary relationship is one in which its target file has a one-to-one relationship with its source file and the source file is a subtype entity of the target file.

18. An end-user query facility as in claim 1 or 2 wherein said information scout comprises an interface that displays all or selected said classes and that allows a user to formulate a query by picking the desired class attributes from said class as a designation of the information to be extracted from said database.

19. An end-user query facility as in claim 2 wherein a said simple query results is one that does not contain repeated numeric values in any of the column(s) that corresponds to a non-key numeric item of said database.

This Sheet Has Been Repla



* C L M S *

20. An end-user query facility as in claim 2 wherein a said simple query results is one that does not contain two or more columns belonging to items whose database files are multi-valued dependencies of another database file in said database.

21. An end-user query facility as in claims 1, 2, or 3 which further comprises a model purifier for a user to alter said database model by defining new keys for database files or by altering existing keys of database files of said database model.

22. An end-user query facility as in claims 1, 2, or 3 which further comprises a model purifier for a user to alter said database model by defining new item(s), new key(s) or new file(s) using items of database files of said database model.

23. An end-user query facility as in claim 21 or 22 wherein said keys comprise unique or repeating keys.

24. An end-user query facility as in claim 21 or 22 wherein said keys comprise primary or foreign keys.

25. An end-user query facility as in claims 1, 2, or 3 which further comprises a program analyzer for analyzing and deriving new item(s), new key(s) or new file(s) from source code of applications that access said database model, and for altering said database model using said new item(s), new key(s) or new files.

26. An end-user query facility as in claims 22 or 25 wherein said program generator comprises means to generate source program that produce data from said new items of said new files and wherein said end-user query facility further comprises a compiler for compiling said source program.

This Sheet Has Been Replaced







27. An end-user query facility as in claims 1, 2, or 3 which further comprises a security model specifier to allow a user to input a security model which specifies restrictions on the information a user can obtain from said database.

28. An end user query facility as in claims 1, 2, or 3 wherein said program comprises a source code program.

29. An end user query facility as in claims 1, 2, or 3 wherein said information scout comprises:

means for obtaining from a user a user supplied keyword indicative of the data desired to be extracted from said database;

means for determining all keywords in said database model having a predefined relationship with said user supplied keyword; and

means for causing a user to select one or more of said keywords having a predefined relationship, said one or more keywords thus selected serving as said designation of the information to be extracted from said database.

30. An end-user query facility as in claim 7 wherein said semantics extractor further comprises means for deriving basic knowledge threads comprising a set of linkages of said database model and storing said basic knowledge threads in said knowledge base.

31. An end-user query facility extractor as in claim 30 wherein each said basic knowledge thread comprises a set of two or more of said database files serially linked together such that one file is linked to the next file through an item that has the same domain as the item in the next file and that this same item is a unique or repeating key of the next file.

32. An end-user query facility extractor as in claim 30 wherein each said basic knowledge thread comprises a set of two or more of said database files serially linked together

This Sheet Has Been Replace



* C L M S *

such that one file has a repeating or foreign key that references the unique or primary key of the next file.

33. An end-user query facility as in claims 1, 2, or 3 wherein said inference engine comprises:

means for inferring new acquired knowledge threads and storing said new acquired knowledge threads in said knowledge base; and

means for determining an access path to said database using either said basic knowledge threads or said acquired knowledge threads as required to meet the user query.

34. An end-user query facility as in claim 33 wherein said acquired knowledge thread is derived through a combination in parallel of two or more of said basic knowledge threads such that one of the said basic knowledge threads has one or more of its consecutive files in common with the corresponding number of consecutive files starting from the thread head of another of said basic knowledge threads.

35. An end-user query facility as in claim 34 wherein said thread head is a first file on a said basic knowledge thread.

36. An end-user query facility as in claims 1, 2, or 3 wherein said knowledge base comprises pre-created knowledge base.

37. An end-user query facility as in claims 1, 2, or 3 wherein said knowledge base comprises a run-time created knowledge base.

38. An end-user query facility as in claims 1, 2, or 3 wherein said knowledge base comprises a persistent knowledge base.

This Sheet Has Been Replace







39. An end-user query facility as in claims 1, 2, or 3 wherein said knowledge base comprises a transient knowledge base.

40. An end-user query facility as in claims 1, 2, or 3 wherein said knowledge base is implemented in data dictionary of said database.

41. An end-user query facility as in claims 1, 2, or 3 wherein said knowledge base is implemented in system catalog of said database.

42. An end-user query facility as in claim 10 wherein said "inverse_of_pure_lookup" binary relationship is a binary relationship whose source file is a pure_lookup entity.

43. An end-user query facility as in claim 11 wherein said "pure_lookup" binary relationship is a binary relationship whose target file is a pure lookup entity.

44. An end-user query facility as in claim 6 wherein said class further comprises a set of relationship names each of which specifies the nature of the relationship between two adjacent database files in each branch of said tree of said class.

45. An end-user query facility as in claim 44 wherein said class generator comprises:

a semantics extractor to derive a set of binary relationships and to identify the entity type of each database file of said database model;

a means to derive said set of classes and their said set of relationship names using said set of binary relationships and said entity type of each database file of said database model.

This Sheet Has Been Replaced







46. An end-user query facility as in claim 45 wherein said class generator further comprises a means for a user to modify said relationship names in the said set of classes.

47. An end-user query facility as in claim 21, 22, or 25 wherein said class generator further comprises a means to read said altered database model to regenerate said set of classes and said set of linkages.

48. An end-user query facility as in claim 1 or 2 which further comprises:

a first e-mail agent for said information scout to interface with an e-mail system, the sail first e-mail agent being used to post said designation of information to be extracted from said database to the mailbox of said inference engine:

a second e-mail agent for the said inference engine to interface with said e-mail system, the said second e-mail agent being used to access said mailbox of said inference engine to obtain said designation of information to be extracted from said database.

49. An end-user query facility as in claims 3 or 48 wherein said second e-mail agent further comprises means to post said desired information extracted from said database to the user mailbox.

50. An end-user query facility as in claim 28 which further comprises a compiler to compile said source code program.

This Sheet Has Been Replaced



* C L M S *