The following is an update to pages that are currently under the Using Record Structures book in the DiscernExplorerHelp>Contents Tab>Programming Reference. This material is a complete re-write and will be available in the next release of the DiscernExplorerHelp.exe.

Record Structures

Record structures allow temporarily storing data in memory. Record structures are predominately used by Cerner's middleware to pass information between a frontend application and a backend script program. The frontend creates a request record structure that is sent through the middleware to execute a specific Discern Explorer program. That program will create and populate a reply record structure which is sent back through the middleware to the frontend application. Record structures are also used by Discern Explorer report programs to temporarily store data in memory.

Defining a Record Structure

Use the RECORD command to define a record structure.

Record Command Syntax

```
RECORD record_name (
        {level group_item}
        {level list_item[occurs]}
        {level field_item = data_type}
        {level field_item[occurs] = data_type}
        ) [WITH scope]
```

Note: the square brackets [] surrounding occurs are a part of the syntax. Occurs is required when defining a list_item. Occurs is optional when defining a field_item.

The following simple definition creates a record structure named ENC. This record structure can be used to store four pieces of information; one number that could have a decimal point and three character strings. Each character string could have up to 40 characters. Presumably this record structure would be used to store an encounter id, and the display value of the nursing unit, room and bed location codes.

```
Record ENC
        (1 ENCNTR_ID = F8
         1 NURSE_UNIT_DISP = C40
         1 ROOM_DISP = C40
         1 BED_DISP = C40)
```

A list item is used in a record structure to store and array or list of values. The following definition creates a record structure named ENC. A list item named QUAL is used to allow the record structure to store multiple encounter ids. Any number of encounter ids can be stored in the QUAL list. The actual encounter ids are stored in the ENCOUNTER field item.

```
Record ENC
        (1 QUAL[*]
          2 ENCOUNTER = F8)
```

The following definition creates a record structure named ORD_STAT. Three list items named CANCELLED, COMPLETED, and FUTURE are used to store information about a person's orders broken down by status. For any one person any number of cancelled, completed, and future orders can be stored in the record structure. This record structure is used to allow printing a person's orders in three columns going down a page. When selected the orders are sorted by status and loaded into the record structure list items. Once all of the orders are stored in the record structure, the first cancelled, first completed, and first future orders are printed on the same line and the row advanced.

```
Record ORD_STAT
        (1 PERSON_ID = F8
         1 NAME = VC
         1 CANCELLED[*]
           2 ORDER_ID = F8
           2 MNEMONIC = C40
         1 COMPLETED[*]
           2 ORDER_ID = F8
           2 MNEMONIC = C40
         1 FUTURE[*]
           2 ORDER_ID = F8
           2 MNEMONIC = C40)
```

The following example creates a record structure called TEMP_1 which stores a person's name and ID at level 1. A list item named ORDERS is used at level 1 to store the order ids and mnemonics for the person's group orders. A list item named DETAILS is used at level 2 to store the result ids and display values of the task assay codes for the components that make up the group order. For example a complete blood count (CBC) is a group order. Some of the individual components of the CBC would be a red blood count (RBC), a white blood count (WBC), a hemoglobin (HGB) and a hematocrit (HCT). The TEMP_1 record structure could be used to store any number of group orders for one specific person. For any one group order any number of components could be stored.

```
RECORD TEMP_1 (
        1 PERSON_ID = F8
        1 NAME = VC
        1 ORDERS[*]
          2 ORDER_ID = F8
          2 ORDER_MNE = VC
          2 DETAILS[*]
                3 RESULT_ID = F8
                3 PROC = VC)
```

When defining a list_item, occurs is either an asterisk [*] or an integer. If an asterisk [*] is used, the list will be stored as a linked list in memory. If an integer is used the list will be stored as a contiguous list in memory. Unless you know the exact number of items that will be stored in the list, the recommendation is to use an asterisk [*] when defining a list_item.

Accessing a Record Structure

To refer to or access a level 1 record structure item, use the record structure name, followed by the dash(-), and the greater than sign (>) When used together the -> is referred to as the record connector. After the record connector use the name of the item. For example if the following RECORD command is use to define the Temp_1 record structure:

```
RECORD TEMP_1 (
        1 PERSON_ID = F8
        1 NAME = VC
        1 ORDERS[*]
          2 ORDER_ID = F8
          2 ORDER_MNE = VC
          2 DETAILS[*]
              3 RESULT_ID = F8
              3 PROC = VC)
```

The following would be used to set the field item named PERSON_ID equal to 12345.0

```
SET TEMP_1->PERSON_ID = 12345.0
```

The following would be used in a report writer section to display the value stored in the PERSON_ID field item in column 10 of the output

```
COL 10 TEMP_1->PERSON_ID
```

To reference a list item, use an integer value inside square brackets [] to specify the position of the list. The integer value can be a literal or a variable. For example to refer to the first position of the ORDERS list in the TEMP_1 record structure use:

```
TEMP_1->ORDERS[1]
```

The record connector (->) must be used between the record structure name and a level 1 item. To reference items at subsequent levels either the record connector (->) or a dot (.) can be used. Both of the following examples refer to the ORDER_ID in the first position of the ORDERS list:

```
TEMP_1->ORDERS[1].ORDER_ID
TEMP_1->ORDERS[1]->ORDER_ID
```

To output the ORDER_ID that is in the first three positions of the ORDERS list in a report writer section you could use:

```
FOR(X = 1 TO 3)
        COL 10 TEMP_1->ORDERS[X].ORDER_ID ROW +1
ENDFOR
```

To access the first 4 detail procedures for the first order, you could use:

```
TEMP_1->ORDERS[1].DETAILS[1].PROC
TEMP_1->ORDERS[1]->DETAILS[2].PROC
TEMP_1->ORDERS[1].DETAILS[3]->PROC
TEMP_1->ORDERS[1]->DETAILS[4]->PROC
```

The SET CURALIAS command can be used to create a shortcut to use instead of the list_items when referencing record structure field_items. The following would display the first four items in the TEMP_1->ORDERS[*n*].ORDER_MNE field

```
SET CURALIAS = ORDS TEMP_1->ORDERS[OCNT]
FOR(OCNT = 1 TO 4)
          COL 10 ORDS->ORDER_MNE
         ROW +1
ENDFOR
```

Although it is not good programming practice, Discern Explorer allows omitting the name of a group_item when referring to items that are under that group. For example if the following Record command is used to define a record structure named REC

```
RECORD REC (
 1 GROUP
   2 FIELD = F8
 )
```

Either of the following could be used to refer to the value in the field_item named FIELD.

```
REC->GROUP.FIELD
REC->FIELD
```

Again omitting group_item names is a bad programming practice.

This functionality can also lead to the creation of record structure items that are inaccessible. For example if the following Record command is used to define a record structure named BAD

```
 RECORD BAD (
 1 GROUP
   2 BAD_FIELD = F8
 1 BAD_FIELD = C10
)
```

By default, the level 1 field_item BAD_FIELD is not accessible. Using BAD->BAD_FIELD, would assume that the group_item GROUP was omitted so the level 2 field_item named BAD_FIELD would be referenced. There are two ways for dealing with this inaccessibility issue. First, the name of one of the field_items named BAD_FIELD could be changed to eliminate the ambiguity. Second, the SET MODIFY RECORDGROUP command can be issued to require the use of group_item names when referring to items that are under that group.

Populating or Loading Data Into a Record Structure

To assign a value to a level 1 record structure item, simply set the field_item equal to the value. If the following RECORD command is used to define the Temp_1 record structure:

```
RECORD TEMP_1 (
        1 PERSON_ID = F8
        1 NAME = VC
        1 ORDERS[*]
          2 ORDER_ID = F8
          2 ORDER_MNE = VC
          2 DETAILS[*]
              3 RESULT_ID = F8
              3 PROC = VC)
```

The following would be used to set the field item named PERSON_ID equal to 12345.0

```
SET TEMP_1->PERSON_ID = 12345.0
```

In a report writer section the SET command is not used. For example the following would set the field item named PERSON_ID equal to 12345.0:

```
DETAIL
        TEMP_1->PERSON_ID = 12345.0
```

Prior to setting a field_item within a list_item equal to a value, memory must be allocated to store the list_items. When a RECORD command that defines a list_item using an integer inside the square brackets [] is executed, memory to store that number of positions in the list is allocated. If additional positions are needed for the list, the ALTER() function is used to increase the total size of the list. Since a contiguous block of memory is used, the ALTER() function will allocate a new block of memory that is large enough to hold the items that are currently in the list and the additional items. The items currently in the list are then moved to this new memory location. This process represents a significant memory management operation. Therefore in cases where the exact number of positions that the list will need is known, the recommendation is to allocate memory for that number of positions using an integer inside the square brackets [] in the record command. If the exact number of positions is not known the recommendation is to define the list_item using an asterisk * inside the square brackets [] in the RECORD command and allocate memory for the list using the ALTERLIST() function. The ALTERLIST( ) function allocates a block of memory in a linked list to store information in a record structure list_item that was defined in the RECORD command using an asterisk * inside the [] square brackets. When a RECORD command that defines a list_item using an asterisk * inside the square brackets [] is executed, no memory is allocated to store the positions of the list. The ALTERLIST() function must be used to allocate an initial block of memory prior to assigning values to field_items under the list_item. The general recommendation is to initially allocate a block of memory that is reasonably large enough to store the number of positions that will be needed for the list. After the initial allocation is used, subsequent executions of the ALTERLIST() function should be used to allocate additional memory in reasonable sized blocks. For example if a select statement is used to get a list of person_ids that need to be stored in a record structure and you expect about a thousand person_ids to be returned. It would be reasonable to initially use the ALTERLIST() function to allocate memory to store a thousand positions in the list. If needed, subsequent executions of the ALTERLIST() function could be used to allocate memory for an additional one hundred positions in the list. If you expected the select to return around twenty person_ids, it would be reasonable to initially use the ALTERLIST() function to allocate enough memory to store twenty positions in the list. If needed subsequent executions of the ALTERLIST() function could be used to allocate memory for an additional ten positions in the list. Since it is recommended to allocate memory in reasonable sized blocks, it is very likely that more memory will be allocated than is actually used. Therefore it is also recommended that once a record structure list has been fully populated the ALTERLIST() function be executed one final time to reduce the size of the list down to the actual number of positions that were actually used.

The following example creates a record structure called TEMP_1 which stores a person's name and ID at level 1. A list item named ORDERS is used at level 1 to store the order IDs and mnemonics for the person's group orders. A

list item named DETAILS is used at level 2 to store the result IDs and display values of the task_assay_cds for the components that make up the group order. For example a complete blood count (CBC) is a group order. Some of the individual components of the CBC would be a red blood count (RBC), a white blood count (WBC), a hemoglobin (HGB) and a hematocrit (HCT). The TEMP_1 record structure could be used to store any number of group orders for one specific person. For any one group order any number of components could be stored. The program selects data from the PERSON, ORDERS, and RESULT tables and loads it into the record structure by assigning record structure field items equal to items from the select. The variables, CNT_ORD and CNT_DET, are used to keep track of the number of positions that are needed in the ORDERS and DETAILS list_items. The ALTERLIST() function is used to incrementally increase the size of the list_items as needed.

The program assumes that on average the person will have about 100 group orders. In the HEAD REPORT section, the person's id and name are stored in the TEMP_1 record structure and the ALTERLIST() function is used to allocate memory to store 100 positions in the ORDERS list_item.

The HEAD O.ORDER_ID section is used to increment the CNT_ORD. An IF() statement is used to determine if additional memory needs to be allocated to store more positions in the ORDERS list_item. When the IF() statement is true, the ALTERLIST() function is used to allocate memory to store 10 more positions in the list. The order_id and mnemonic from the orders table are then stored in the respective fields in the ORDERS list_item. The CNT_DET variable is set to zero so it can be used to count the number of components for the current group order. The program assumes that on average each group order will have about ten components. So the ALTERLIST() function is used to allocate memory to store 10 positions in the DETAILS list_item within the current ORDERS list_item.

In the DETAIL section, the CNT_DET variable is incremented. An IF() statement is used to determine if memory needs to be allocated to store more positions in the DETAILS list_item. When the IF() statement is true, the ALTERLIST() function is used to allocate memory to store 10 more positions in the list. The result_id and display value of the task_assay_cd (proc) from the result table are then stored in the respective fields DETAILS list_item.

The FOOT O.ORDER_ID section executes the ALTERLIST() function to deallocate memory that was allocated but not used, to store components of the group order.

The FOOT REPORT section executes the ALTERLIST() function to deallocate memory that was allocated but not used, to store group orders for the person.

```
DROP PROGRAM EX_RECORD GO
CREATE PROGRAM EX_RECORD

SET CNT_ORD = 0
SET CNT_DET = 0

RECORD TEMP_1 (
1       PERSON_ID = F8
1       NAME = VC
1       ORDERS[*]
      2   ORDER_ID = F8
      2   ORDER_MNE = VC
      2   DETAILS[*]
          3   RESULT_ID = F8
          3   PROC = VC)

SELECT INTO "NL:"
        P.PERSON_ID,
        P.NAME_FULL_FORMATTED,
        O.ORDER_ID,
        O.ORDER_MNEMONIC,
        R.RESULT_ID,
        PROC = UAR_GET_CODE_DISPLAY(R.TASK_ASSAY_CD)
FROM  PERSON P,
        ORDERS O,
```

```
        RESULT R
PLAN P WHERE P.PERSON_ID = 12345.0
JOIN O WHERE P.PERSON_ID = O.PERSON_ID
JOIN R WHERE O.ORDER_ID = R.ORDER_ID
ORDER           O.ORDER_ID,
                R.RESULT_ID
HEAD REPORT
        ;store the person_id and name in the record structure
        TEMP_1->PERSON_ID = P.PERSON_ID
        TEMP_1->NAME = P.NAME_FULL_FORMATTED
        ;allocate memory to store information for 100 group orders
        STAT = ALTERLIST(TEMP_1->ORDERS,100)
        ;initialize variables to keep track of the number of group orders and components
        CNT_ORD = 0
        CNT_DET = 0
HEAD O.ORDER_ID
        CNT_ORD = CNT_ORD + 1
        ;if needed, allocate memory to store information for 10 additional group orders
        IF (MOD(CNT_ORD,10) = 1 AND CNT_ORD > 100)
             STAT = ALTERLIST(TEMP_1->ORDERS,CNT_ORD + 9)
        ENDIF
        ;store information for the current group order in the record structure
        TEMP_1->ORDERS[CNT_ORD].ORDER_ID = O.ORDER_ID
        TEMP_1->ORDERS[CNT_ORD].ORDER_MNE = O.ORDER_MNEMONIC
        ;set the component count for the current group order to zero
        CNT_DET = 0
        ;allocate memory to store 10 components for the current group order
        STAT = ALTERLIST(TEMP_1->ORDERS[CNT_ORD].DETAILS,10)
DETAIL
        CNT_DET = CNT_DET + 1
        ;if needed allocate memory to store 10 additional components
        IF (MOD(CNT_DET,10) = 1 AND CNT_DET != 1)
              STAT = ALTERLIST(TEMP_1->ORDERS[CNT_ORD].DETAILS,CNT_DET + 9)
        ENDIF
        ;store information for the current component in the record structure
        TEMP_1->ORDERS[CNT_ORD].DETAILS[CNT_DET].RESULT_ID = R.RESULT_ID
        TEMP_1->ORDERS[CNT_ORD].DETAILS[CNT_DET].PROC = PROC
FOOT O.ORDER_ID
        ;free memory that was allocated but not used
        ;for components of the current group order
        STAT = ALTERLIST(TEMP_1->ORDERS[CNT_ORD].DETAILS,CNT_DET)
FOOT REPORT
        ;free memory that was allocated but not used for group orders
        STAT = ALTERLIST(TEMP_1->ORDERS,CNT_ORD)

END
GO
```

Selecting Information that is Related to Values Currently in a Record Structure

Situations often arise where you need to query for information that is related to values that are currently stored in a record structure. Suppose the following RECORD command was used to define the Temp_1 record structure, and a query had already been executed to load information about a person, their group orders, and the components of those group orders into the Temp_1 record structure. In addition to the information currently loaded in the Temp_1 record structure, the program now needs to get addresses that belong to the person.

```
RECORD TEMP_1 (
        1 PERSON_ID = F8
        1 NAME = VC
        1 ORDERS[*]
          2 ORDER_ID = F8
          2 ORDER_MNE = VC
          2 DETAILS[*]
                3 RESULT_ID = F8
                3 PROC = VC)
```

Since the Person_ID is stored in a level one item in the Temp_1 record structure, it can be referenced directly in a qualification. The following example would select the address information for the person_id that is stored in the Temp_1 record structure:

```
SELECT
          A.ADDRESS_ID
        , A.STREET_ADDR
        , A.CITY
        , A_STATE_DISP = UAR_GET_CODE_DISPLAY(A.STATE_CD)
        , A.ZIPCODE
FROM
        ADDRESS   A
WHERE A.PARENT_ENTITY_ID = TEMP_1->PERSON_ID
        AND A.PARENT_ENTITY_NAME = "PERSON"
WITH NOCOUNTER
```

The same process can be used to reference a specific value that is stored in a list_item. For example

```
SELECT OC.* FROM ORDER_COMMENTS OC WHERE OC.ORDER_ID = TEMP_1->ORDERS[1].ORDER_ID
```

The above could be used to select order comment information related to the order_id that is stored in the ORDER_ID field_item in the first position of the TEMP_1->ORDERS list_item. However, if you needed to select information related to every order_id stored in the TEMP_1->ORDERS list_item, you need a method traverse the entire list and extract each order_id for use in a qualification.

Three common methods are used to traverse record structure list_items to extract values for use in qualifications; a looping statement, the EXPAND() function, and joining to the dummyt table.

Occasionally a query will be placed inside a FOR or WHILE loop. This method is very inefficient and SHOULD NOT be used. With this method the loop is used to increment a variable. The variable is used as the subscript to refer to a position of a record structure list_item in a qualification. Placing a query inside a loop is very inefficient because each iteration of the loop causes Discern Explorer to translate and pass the query to the RDBMS. Again using a query inside a loop is very inefficient and should not be done.

Using the EXPAND() function is the recommended method for qualifying on values in a single level record structure list_item. The EXPAND() function will extract the values from a record structure field_item that is within a list_item, and place them in an IN clause that is passed to the RDBMS query from Discern Explorer. For example suppose you were writing a report that returned information about people, their encounters, and orders. The report also needs to display the person's person aliases. All of this information could be selected in a single query. However, if a single

query is used to select the information, the result set will contain duplicate information. Each encounter a person has will be duplicated for each person alias. A method that is commonly used to eliminate the duplication is to use one select that loads the person, encounter, and orders information into a record structure. Then use a second select to get the person aliases for each person. Report writer clauses are then added to the second select to display the information from the record structure and the information that is returned in the select's result set. The following example demonstrates using this method.

```
DROP PROGRAM CCL_RECORD_EX2 GO
CREATE PROGRAM CCL_RECORD_EX2

prompt
        "Output to File/Printer/MINE " = "MINE"
        , "Enter Last Name" = " "
        , "Enter First Name" = " "

with OUTDEV, Lname, Fname

/*Record structure to store person, encounter, and orders information */
record PERSON_REC (
  1 PLIST [*]                ;person list
    2 PID = f8
    2 NAME = C30
    2 ELIST [*]              ;encounter list
      3 EID = f8
      3 TYPE = C40
      3 OLIST [*]            ;orders list
        4 OID = f8
        4 CAT_DISP = C40
)

/*************************************************************
Declared Variables
*************************************************************/
declare NUM   = I4 with Protect  ;index var for expand()
declare POS   = I4 with Protect  ;position var for locateval()
declare PCNT = I4 with Protect   ;count of people
declare ECNT = I4 with Protect   ;count of encounters
declare OCNT = I4 with Protect   ;count of orders

/*************************************************************
Select person, encounter, orders information and store in record structure
*************************************************************/
SELECT INTO "NL:"
          P.PERSON_ID
        , P.NAME_FULL_FORMATTED
        , E.ENCNTR_ID
        , E_ENCNTR_TYPE_CLASS_DISP = UAR_GET_CODE_DISPLAY(E.ENCNTR_TYPE_CLASS_CD)
        , O.ORDER_ID
        , O_CATALOG_DISP = UAR_GET_CODE_DISPLAY(O.CATALOG_CD)

FROM    PERSON   P
        , ENCOUNTER   E
        , ORDERS   O
;$fname and $lname are passed from prompts
PLAN P WHERE P.NAME_LAST_KEY = $LNAME AND P.NAME_FIRST_KEY = $FNAME
JOIN E WHERE P.PERSON_ID = E.PERSON_ID
JOIN O WHERE E.ENCNTR_ID = O.ENCNTR_ID
```

```
ORDER BY
          P.PERSON_ID
        , E.ENCNTR_ID
        , O.ORDER_ID

HEAD REPORT
        ;allocate memory for 100 people in the person list
        STAT = ALTERLIST(PERSON_REC->PLIST, 100)
HEAD P.PERSON_ID
        ;increment count of people
        PCNT = PCNT +1
        ;check for available memory in the person list
        IF(MOD(PCNT,10) = 1 AND PCNT >100)
                ;if needed allocate memory for 10 more people
                STAT = ALTERLIST(PERSON_REC->PLIST, PCNT +9)
        ENDIF
        ;store the person information in the record structure
        PERSON_REC->PLIST[PCNT].PID = P.PERSON_ID
        PERSON_REC->PLIST[PCNT].NAME = P.NAME_FULL_FORMATTED
        ;reset the count of encounters to zero for this person
        ECNT = 0
HEAD E.ENCNTR_ID
        ;increment count of encounters
        ECNT = ECNT +1
        ;check for available memory in the encounter list
        IF(MOD(ECNT,10) = 1)
                ;if needed allocate memory for 10 more encounters for the current person
                STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST, ECNT +9)
        ENDIF
        ;store the encounter information in the record structure
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].EID = E.ENCNTR_ID
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].TYPE = E_ENCNTR_TYPE_CLASS_DISP
        ;rest the count of orders to zero for this encounter
        OCNT = 0
DETAIL
        ;increment the count of orders
        OCNT = OCNT +1
        ;check for available memory in the encounter list
        IF(MOD(OCNT,10) = 1)
                ;if needed allocate memory for 10 more orders for this encounter
                STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST, OCNT +9)
        ENDIF
        ;store the order information in the record structure
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST[OCNT].OID = O.ORDER_ID
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST[OCNT].CAT_DISP = O_CATALOG_DISP
FOOT E.ENCNTR_ID
        ;free memory that was allocated but not used for orders
        STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST, OCNT)
FOOT P.PERSON_ID
        ;free memory that was allocated but not used for encounters
        STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST, ECNT)
FOOT REPORT
        ;free memory that was allocated but not used for people
        STAT = ALTERLIST(PERSON_REC->PLIST, PCNT)
WITH  NOCOUNTER

/************************************************************
```

```
Select the person aliases for each person_id in the record structure
************************************************************/
SELECT INTO $OUTDEV
          P.PERSON_ID
        , P_PERSON_ALIAS_TYPE_DISP = UAR_GET_CODE_DISPLAY(P.PERSON_ALIAS_TYPE_CD)
        , PA_ALIAS = SUBSTRING(1,30,P.ALIAS)

FROM
        PERSON_ALIAS   P
/*
Using the expand function will cause the following where clause to be translated to
        where p.person_id IN (:1,:2,:3,...:n)
in the query that is passed to the RDMBS when this program is executed.
:1,:2,:3,...:n will be bind variables that are set equal to each of the
values in the person_rec->plist[].pid record structure field_item.
*/
WHERE EXPAND(NUM,1,PCNT,P.PERSON_ID,PERSON_REC->PLIST[NUM].PID)
ORDER BY
        P.PERSON_ID

HEAD P.PERSON_ID
        /* Locatevalsort() will perform a binary search to determine which
        position of person_rec->plist[].pid field is equal to the current p.person_id.
        The variable pos will be set equal to that position.
        Since the person_ids were sorted prior to being stored in the person list (plist) the
        locatevalsort() function can be used.  If the person_ids were not in sorted order in
        person list, locateval() would need to be used to perform a sequential search of the
        person list. */
        POS = LOCATEVALSORT(NUM,1,PCNT,P.PERSON_ID,PERSON_REC->PLIST[NUM].PID)
        ;display the person information from the record structure
        COL  0 PERSON_REC->PLIST[POS].PID
        COL +1 PERSON_REC->PLIST[POS].NAME
DETAIL
        ;display the person aliases from the result set of this query
        COL 70 PA_ALIAS
        COL +1 P_PERSON_ALIAS_TYPE_DISP
        ROW +1
FOOT P.PERSON_ID
        ;display the encounter information from the record structure for this person
        FOR(E_POS = 1 TO SIZE(PERSON_REC->PLIST[POS].ELIST,5))
                COL 10  PERSON_REC->PLIST[POS].ELIST[E_POS].EID
                COL +1  PERSON_REC->PLIST[POS].ELIST[E_POS].TYPE
                ROW +1
                ;display the order information from the record structure for this encounter
                FOR(O_POS = 1 TO SIZE( PERSON_REC->PLIST[POS].ELIST[E_POS].OLIST,5))
                        COL 20 PERSON_REC->PLIST[POS].ELIST[E_POS].OLIST[O_POS].OID
                        COL +1 PERSON_REC->PLIST[POS].ELIST[E_POS].OLIST[O_POS].CAT_DISP
                        ROW +1
                ENDFOR
        ENDFOR
        ROW +2
WITH NOCOUNTER,  MAXCOL = 200

END
GO
```

Using the EXPAND() function is generally the most efficient method for qualifying on values in a single level record structure list_item. Since the EXPAND() function creates an IN clause that is passed in the query to the RDBMS,

there are several factors that need to be considered. First, how much variance will be in the number of items that are stored in the record structure list_item. Second, will the number of items that are passed in the IN clause be too large for the RDBMS optimizer to efficiently retrieve the related data. Finally, will the number of items in the record structure list_item be greater than 200.

When a Discern Explorer query is executed, the query is translated and passed to the RDBMS. The RDBMS determines a plan for executing that query and caches that plan. If the RDBMS receives the exact same query again, it will execute the query using the cached plan. A finite number of plans can be kept in the cache. If the RDBMS receives a new query that does not have a cached execution plan, it will determine a plan for the new query, push the plan for the query with the oldest most recent execution out of the cache, and place the plan for the new query in the cache. The cache will always contain the plans for the most recently executed queries. Any variance in a query, including the number of items in the IN clause or the value of the items in the IN will cause the RDBMS to cache a new plan for the query. To prevent that, Discern Explorer assigns the values from the record structure list_item to bind variables and passes the bind variables in the IN clause to the RDBMS. Discern Explorer will also pad the number of items passed in the IN clause up to the nearest multiple of 20. If you had the following values when using the person list from the previous example:

PERSON_REC->PLIST[1].PID = 11.0
PERSON_REC->PLIST[2].PID = 22.0
PERSON_REC->PLIST[3].PID = 33.0
PERSON_REC->PLIST[4].PID = 44.0
PERSON_REC->PLIST[5].PID = 55.0

The IN clause that would be passed to the RDBMS from using

```
WHERE EXPAND(NUM,1,PCNT,P.PERSON_ID,PERSON_REC->PLIST[NUM].PID)
```

Would be

```
WHERE P.PERSON_ID IN( :1, :2, :3, :4, :5, :6, :7, :8, :9, :10, :11, :12, :13, :14, :15, :16, :17, :18, :19, :20)
```

In the above, the bind variables would be set a follows:

:1 = 11.0
:2 = 22.0
:3 = 33.0
:4 = 44.0
:5 = 55.0
:6 through :20 = 55.0

If a sixth value was added to the PERSON_REC->PLIST[6].PID, the bind variables :6 through :20 would be set equal to that value. If twenty one values were placed in the person list, forty bind variables would be passed in the IN clause to the RDBMS. Bind variables :21 through :40 would all be set equal to the same value. Using bind variables and padding the number of items in the IN up to the next multiple of 20 reduces the number of plans that could be stored in the cache for this program. Without this method any variance in the number of items in the person list would cause a new plan to be generated and cached. However, if the number of items that need to be passed in the IN clause is greater than 200, Discern Explorer will pass them as literal values instead of using bind variables. Using the literal values will cause a new plan to be cached for each execution of the program since most likely one or more of the literal values will be different for multiple executions of the program. Often when the number of items in an IN clause is greater than 200 the RDBMS will have trouble efficiently processing the query. To prevent that the dummyt table can be used to break the number of items that will be passed in the IN clause into groups of 200 and execute the query at the RDBMS level multiple times. The following example uses this method.

The example initializes the following variables that are used to control the number of elements that are passed in the IN clause generated by the Expand() function. These variables must be an integer data type.

Actual_Size is the actual size of the record structure list_item.

Expand_Total is the actual_size padded up to the next multiple of 200.

Expand_Start will be passed to the Expand() function and will be used as the starting position of the record structure list_item. Expand_Start must be initialized to value greater than 0 (zero). It is generally initialized to 1 (one) and is then reset using the Assign() function in the join to the Dummyt. The internal processing that Discern Explorer uses to build the IN clause and reset Expand_Start requires that Expand_Start be initialized to the starting value prior to execution of the select that uses it.

Expand_Stop will be passed to the Expand() function and will be used as the stop position of the record structure list. Expand_Stop must be initialized to value greater than 0 (zero). It is generally initialized to 200 and is then reset using the Assign() function in the join to the Dummyt. The internal processing that Discern Explorer uses to build the IN clause and reset Expand_Stop requires that Expand_Stop be initialized to the stop value prior execution of the select.

```
DROP PROGRAM CCL_RECORD_EX3 GO
CREATE PROGRAM CCL_RECORD_EX3

prompt
            "Output to File/Printer/MINE " = "MINE"
          , "Enter Last Name" = " "
          , "Enter First Name" = " "

with OUTDEV, Lname, Fname

/*Record structure to store person and encounter information */
record PERSON_REC (
  1 PLIST [*]                ;person list
    2 PID = f8
    2 NAME = C30
    2 ELIST [*]              ;encounter list
      3 EID = f8
      3 TYPE = C40
)

/*************************************************************
Declared Variables
*************************************************************/
declare NUM  = I4 with Protect ;index var for expand()
declare POS  = I4 with Protect ;position var for locateval()
declare PCNT = I4 with Protect ;count of people
declare ECNT = I4 with Protect ;count of encounters
;the following variables will be used to always pass 200 items in the IN clause when using expand()
declare ACTUAL_SIZE   = I4 with Protect
declare EXPAND_TOTAL  = I4 with Protect
declare EXPAND_START  = I4 with NoConstant(1), Protect
declare EXPAND_STOP   = I4 with NoConstant(200), Protect

/*************************************************************
Select person and encounter information and store in record structure
*************************************************************/

SELECT INTO "NL:"
            P.PERSON_ID
          , P.NAME_FULL_FORMATTED
          , E.ENCNTR_ID
          , E_ENCNTR_TYPE_CLASS_DISP = UAR_GET_CODE_DISPLAY(E.ENCNTR_TYPE_CLASS_CD)

FROM
            PERSON   P
          , ENCOUNTER   E
;$fname and $lname are passed from prompts
```

```
PLAN P WHERE P.NAME_LAST_KEY = $LNAME AND P.NAME_FIRST_KEY = $FNAME
JOIN E WHERE P.PERSON_ID = E.PERSON_ID

ORDER BY
          P.PERSON_ID
         , E.ENCNTR_ID

HEAD REPORT
         ;allocate memory for 100 people in the person list
         STAT = ALTERLIST(PERSON_REC->PLIST, 100)
HEAD P.PERSON_ID
         ;increment count of people
         PCNT = PCNT +1
         ;check for available memory in the person list
         IF(MOD(PCNT,10) = 1 AND PCNT >100)
                  ;if needed allocate memory for 10 more people
                  STAT = ALTERLIST(PERSON_REC->PLIST, PCNT +9)
         ENDIF
         ;store the person information in the record structure
         PERSON_REC->PLIST[PCNT].PID = P.PERSON_ID
         PERSON_REC->PLIST[PCNT].NAME = P.NAME_FULL_FORMATTED
         ;reset the count of encounters to zero for this person
         ECNT = 0
DETAIL
         ;increment count of encounters
         ECNT = ECNT +1
         ;check for available memory in the encounter list
         IF(MOD(ECNT,10) = 1)
                  ;if needed allocate memory for 10 more encounters for the current person
                  STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST, ECNT +9)
         ENDIF
         ;store the encounter information in the record structure
         PERSON_REC->PLIST[PCNT].ELIST[ECNT].EID = E.ENCNTR_ID
         PERSON_REC->PLIST[PCNT].ELIST[ECNT].TYPE = E_ENCNTR_TYPE_CLASS_DISP
FOOT P.PERSON_ID
         ;free memory that was allocated but not used for encounters
         STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST, ECNT)
FOOT REPORT
         ;free memory that was allocated but not used for people
         STAT = ALTERLIST(PERSON_REC->PLIST, PCNT)
WITH NOCOUNTER


/************************************************************
Select the person aliases for each person_id in the record structure
using the dummyt method to keep the size of the IN clause = 200 items
************************************************************/

;actual_size is the actual number of person_ids in the person_rec->plist
SET ACTUAL_SIZE  = SIZE(PERSON_REC->PLIST,5)
;if actual_size is not a multiple of 200, increase the size of the list to
;the next multiple of 200 to allow always passing 200 bind variables in
;the IN clause that is generated by the expand()
IF(MOD(ACTUAL_SIZE,200) != 0)
         ;set expand_total to the next multiple of 200
         SET EXPAND_TOTAL = ACTUAL_SIZE +(200 - MOD(ACTUAL_SIZE,200))
         ;increase the list size to the next multiple of 200
         SET STAT = ALTERLIST(PERSON_REC->PLIST, EXPAND_TOTAL)
```

```
        ;set the added positions of list equal to the last person_id in the list
        FOR(IDX = ACTUAL_SIZE+1 TO EXPAND_TOTAL)
                SET PERSON_REC->PLIST[IDX].PID = PERSON_REC->PLIST[ACTUAL_SIZE].PID
        ENDFOR
ELSE
        SET EXPAND_TOTAL = ACTUAL_SIZE
ENDIF


SELECT INTO $OUTDEV
          P.PERSON_ID
        , P_PERSON_ALIAS_TYPE_DISP = UAR_GET_CODE_DISPLAY(P.PERSON_ALIAS_TYPE_CD)
        , PA_ALIAS = SUBSTRING(1,30,P.ALIAS)

FROM
        PERSON_ALIAS   P
        ,(DUMMYT D WITH SEQ = VALUE(EXPAND_TOTAL/200))
/*Use dummyt to execute the query with 200 items in the IN clause.
Expand_total/200 returns the number of times the query will be executed at the RDBMS level.
Expand_start and expand_stop are set using assign().  The first time the query is executed at the
RDBMS level, expand_start will be set to 1 and expand_stop will be set to 200.  The second time the
query is executed at the RDBMS level expand_start will be set to 201 and expand_stop will be set to 400.
Expand_start and expand_stop will continue to be incremented in blocks of 200 until all of the person_ids
in the person_list have been passed in the IN clause of the query at the RDBMS level. */

PLAN D WHERE ASSIGN(EXPAND_START,EVALUATE(D.SEQ,1,1,EXPAND_START+200))
                AND ASSIGN(EXPAND_STOP,EXPAND_START+(199))
JOIN P WHERE
        EXPAND(NUM,EXPAND_START,EXPAND_STOP,P.PERSON_ID,PERSON_REC->PLIST[NUM].PID)
ORDER BY
        P.PERSON_ID

HEAD P.PERSON_ID
        /* Locatevalsort() will perform a binary search to determine which
        position of person_rec->plist[].pid field is equal to the current p.person_id.
        The variable pos will be set equal to that position.
        Since the person_ids were sorted prior to being stored in the person list (plist) the
        locatevalsort() function can be used.  If the person_ids were not in sorted order in the
        person list, locateval() would need to be used to perform a sequential search of the
        person list. */
        POS = LOCATEVALSORT(NUM,1,PCNT,P.PERSON_ID,PERSON_REC->PLIST[NUM].PID)
        ;display the person information from the record structure
        COL  0 PERSON_REC->PLIST[POS].PID
        COL +1 PERSON_REC->PLIST[POS].NAME
DETAIL
        ;display the person aliases from the result set of this query
        COL 70 PA_ALIAS
        COL +1 P_PERSON_ALIAS_TYPE_DISP
        ROW +1
FOOT P.PERSON_ID
        ;display the encounter information from the record structure for this person
        FOR(E_POS = 1 TO SIZE(PERSON_REC->PLIST[POS].ELIST,5))
                COL 10  PERSON_REC->PLIST[POS].ELIST[E_POS].EID
                COL +1  PERSON_REC->PLIST[POS].ELIST[E_POS].TYPE
                ROW +1
        ENDFOR
        ROW +2
WITH NOCOUNTER, SEPARATOR=" ", FORMAT, MAXCOL = 200
```

```
END
GO
```

Using the EXPAND() function is the most efficient method of selecting information that is related to values that are currently in a single level record structure list. However, the EXPAND() function cannot be used when working with multi-level record structure lists. Instead the DUMMYT table and MAXREC() function should be used to traverse multi-level record structure lists. For example suppose you were writing a report that returned information about people, their encounters, and orders. The report also needed to display the person's encounter aliases. All of this information could be selected in a single query. However, if a single query was used to select the information, the result set would contain duplicate information. Each order a person has would be duplicated for each encounter alias. A method that is commonly used to eliminate the duplication is to use one select that loads the person, encounter, and orders information into a record structure. Then use a second select to get the encounter aliases for each encounter. Report writer clauses are then added to the second select to display the information from the record structure and the information that is returned in the select's result set. The following example uses the SEQ Pseudo Column and the DUMMYT table to demonstrate how to select information that is related to values in a multi-level record structure list.

```
DROP PROGRAM CCL_RECORD_EX4 GO
CREATE PROGRAM CCL_RECORD_EX4

prompt
        "Output to File/Printer/MINE" = "MINE"
        , "Enter Last Name" = " "
        , "Enter First Name" = " "

with OUTDEV, LNAME, FNAME

/*Record structure to store person, encounter, and orders information */
record PERSON_REC (
  1 PLIST [*]                ;person list
    2 PID = f8
    2 NAME = C30
    2 ELIST [*]              ;encounter list
      3 EID = f8
      3 TYPE = C40
      3 OLIST [*]            ;orders list
        4 OID = f8
        4 CAT_DISP = C40
)

/*************************************************************
Declared Variables
*************************************************************/
declare PCNT = I4 with Protect ;count of people
declare ECNT = I4 with Protect ;count of encounters
declare OCNT = I4 with Protect ;count of orders

/*************************************************************
Select person, encounter, orders information and store in record structure
*************************************************************/
SELECT INTO "NL:"
          P.PERSON_ID
        , P.NAME_FULL_FORMATTED
        , E.ENCNTR_ID
        , E_ENCNTR_TYPE_CLASS_DISP = UAR_GET_CODE_DISPLAY(E.ENCNTR_TYPE_CLASS_CD)
```

```
        , O.ORDER_ID
        , O_CATALOG_DISP = UAR_GET_CODE_DISPLAY(O.CATALOG_CD)

FROM    PERSON   P
        , ENCOUNTER   E
        , ORDERS   O
;$fname and $lname are passed from prompts
PLAN P WHERE P.NAME_LAST_KEY = $LNAME AND P.NAME_FIRST_KEY = $FNAME
JOIN E WHERE P.PERSON_ID = E.PERSON_ID
JOIN O WHERE E.ENCNTR_ID = O.ENCNTR_ID

ORDER BY
         P.PERSON_ID
        , E.ENCNTR_ID
        , O.ORDER_ID

HEAD REPORT
        ;allocate memory for 100 people in the person list
        STAT = ALTERLIST(PERSON_REC->PLIST, 100)
HEAD P.PERSON_ID
        ;increment count of people
        PCNT = PCNT +1
        ;check for available memory in the person list
        IF(MOD(PCNT,10) = 1 AND PCNT >100)
                ;if needed allocate memory for 10 more people
                STAT = ALTERLIST(PERSON_REC->PLIST, PCNT +9)
        ENDIF
        ;store the person information in the record structure
        PERSON_REC->PLIST[PCNT].PID = P.PERSON_ID
        PERSON_REC->PLIST[PCNT].NAME = P.NAME_FULL_FORMATTED
        ;reset the count of encounters to zero for this person
        ECNT = 0
HEAD E.ENCNTR_ID
        ;increment count of encounters
        ECNT = ECNT +1
        ;check for available memory in the encounter list
        IF(MOD(ECNT,10) = 1)
                ;if needed allocate memory for 10 more encounters for the current person
                STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST, ECNT +9)
        ENDIF
        ;store the encounter information in the record structure
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].EID = E.ENCNTR_ID
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].TYPE = E_ENCNTR_TYPE_CLASS_DISP
        ;rest the count of orders to zero for this encounter
        OCNT = 0
DETAIL
        ;increment the count of orders
        OCNT = OCNT +1
        ;check for available memory in the encounter list
        IF(MOD(OCNT,10) = 1)
                ;if needed allocate memory for 10 more orders for this encounter
                STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST, OCNT +9)
        ENDIF
        ;store the order information in the record structure
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST[OCNT].OID = O.ORDER_ID
        PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST[OCNT].CAT_DISP = O_CATALOG_DISP
FOOT E.ENCNTR_ID
        ;free memory that was allocated but not used for orders
```

```
        STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST[ECNT].OLIST, OCNT)
FOOT P.PERSON_ID
        ;free memory that was allocated but not used for encounters
        STAT = ALTERLIST(PERSON_REC->PLIST[PCNT].ELIST, ECNT)
FOOT REPORT
        ;free memory that was allocated but not used for people
        STAT = ALTERLIST(PERSON_REC->PLIST, PCNT)
WITH  NOCOUNTER

/************************************************************
Select the encounter aliases for each encntr_id in the record structure

Defining the dummyt d1 with SEQ equal the size of the plist in the FROM
clause, and planning on d1 causes Discern Explorer to set up an internal
loop that sets the pseudo field d1.seq equal to 1 and increments it by 1
until it is equal to the number of positions in the plist.

Using the MaxRec() function and joining to d2 causes the Discern Explorer
internal loop to look at each position of the plist and set up an internal
child loop that sets the pseudo field d2.seq equal to 1 and increments it
by 1 until it is equal to the number of positions in the elist for this person.

Joining to the encntr_alias table and using d1.seq and d2.seq as subscripts
in the record structure lists, causes Discern Explorer to pull each encntr_id
out of the elists.  That encntr_id is assigned to a bind variable in a query
that is passed to the RDBMS to get the information from the encntr_alias table
for that specific encntr_id.  Using this method, the query will be executed one
time at the RDBMS level for each encntr_id that is stored in one of the elists.
Discern Explorer gets the result set from each execution of the query and combines
them into a single result set.

Since Discern Explorer is using internal looping to increment the pseudo fields
d1.seq and d2.seq, these values can also be used to reference positions in the
record structure lists that are related to the encntr_alias fields from the result
set of the query.
************************************************************/

SELECT INTO $OUTDEV
          PLIST_PID  = PERSON_REC->PLIST[D1.SEQ].PID
        , PLIST_NAME = PERSON_REC->PLIST[D1.SEQ].NAME
        , ELIST_EID  = PERSON_REC->PLIST[D1.SEQ].ELIST[D2.SEQ].EID
        , ELIST_TYPE = PERSON_REC->PLIST[D1.SEQ].ELIST[D2.SEQ].TYPE
        , E.ENCNTR_ID
        , E.ALIAS
        , E_ENCNTR_ALIAS_TYPE_DISP = UAR_GET_CODE_DISPLAY(E.ENCNTR_ALIAS_TYPE_CD)

FROM
        ENCNTR_ALIAS   E
        , (DUMMYT   D1  WITH SEQ = VALUE(SIZE(PERSON_REC->PLIST, 5)))
        , (DUMMYT   D2  WITH SEQ = 1)
PLAN D1 WHERE MAXREC(D2, SIZE(PERSON_REC->PLIST[D1.SEQ].ELIST, 5))
JOIN D2
JOIN E WHERE E.ENCNTR_ID = PERSON_REC->PLIST[D1.SEQ].ELIST[D2.SEQ].EID
ORDER BY
        PLIST_PID,
        E.ENCNTR_ID
HEAD PLIST_PID
        ;display person information from the record structure
```

```
        ;the select expressions PLIST_PID and PLIST_NAME were created in the select list above
        COL  0 PLIST_PID
        COL +1 PLIST_NAME
        ;you could reference the record structure field items here as well i.e.
        ;col  0 person_rec->plist[d1.seq].pid
        ;col +1 person_rec->plist[d1.seq].name
        ROW +1
HEAD E.ENCNTR_ID
        ;display encounter information from the record structure
        ;the select expressions ELIST_PID and ELIST_TYPE were created in the select list above
        COL 10 ELIST_EID
        COL +1 ELIST_TYPE
        ROW +1
DETAIL
        ;display the encounter alias information from the result set
        ALIAS = SUBSTRING(1, 30, E.ALIAS)
        COL 20 ALIAS
        COL +1 E_ENCNTR_ALIAS_TYPE_DISP
        ROW +1
FOOT E.ENCNTR_ID
        ;display the orders for each encounter
        FOR(O_POS = 1 TO SIZE(PERSON_REC->PLIST[D1.SEQ].ELIST[D2.SEQ].OLIST,5))
                COL 30          PERSON_REC->PLIST[D1.SEQ].ELIST[D2.SEQ].OLIST[O_POS].OID
                COL +1          PERSON_REC->PLIST[D1.SEQ].ELIST[D2.SEQ].OLIST[O_POS].CAT_DISP
                ROW +1
        ENDFOR
FOOT PLIST_PID
        ROW +1
WITH NOCOUNTER, SEPARATOR=" ", FORMAT

END
GO
```

Capturing Record Structure Status

The STATUS( ) control option will capture the status of each operation while processing through a record structure list on an insert, update, or delete. The STATUS( ) control option will set the record item equal to 0 if the operation failed or 1 if it is successful.

The program below attempts to insert two rows into a custom RDBMS table named TEST_TABLE. If both inserts are successful, R1->QUAL[1].STAT and R1->QUAL[2].STAT will set equal to 1. If there was a unique index on the TEST_TABLE.ID field, attempting to set the TEST_TABLE.ID to the same value on two different rows would cause a unique constraint violation causing the second attempt to fail. In this case R1->QUAL[1].STAT would be set equal to 1 and R1->QUAL[2].STAT would be set equal to 0.

```
DROP PROGRAM TEST_INS GO
CREATE PROGRAM TEST_INS

RECORD R1(
    1 LOAD_CNT   = I4
    1 TOTAL_QUAL = I4
    1 QUAL[10]
        2 STAT  = I1     ;=0 the insert failed, =1 the insert succeeded
        2 ID  = I4
        2 PNAME  = C40
        )
    SET R1->QUAL[1].ID = 55
    SET R1->QUAL[1].PNAME = "TEST55"
    SET R1->QUAL[2].ID = 55
    SET R1->QUAL[2].PNAME = "TEST56"
    SET R1->LOAD_CNT = 2

INSERT FROM
    TEST_TABLE M,
    (DUMMYT D WITH SEQ=VALUE(R1->LOAD_CNT))

SET
    M.ID = R1->QUAL[D.SEQ].ID,
    M.PNAME = R1->QUAL[D.SEQ].PNAMEL[D.SEQ].DATE2)
PLAN D
JOIN M
WITH   NOCOUNTER, STATUS(R1->QUAL[D.SEQ].STAT) ;CAPTURING STATUS OF INSERT

END
GO
```

Related Topics

You can copy one record structure to another by defining each structure, setting the record items, and then setting one structure equal to the other.

```
RECORD TEMP_1
( 1 TEST_ID = F8 ) GO

RECORD TEMP_2
( 1 TEST_ID = F8 ) GO

SET TEMP_1->TEST_ID = 1 GO
SET TEMP_2 = TEMP_1 GO
```

The MOVEREC() and MOVERECLIST() functions can be used to move values from one record structure to another.

The LOCATEVAL() and LOCATEVALSORT() functions can be used to search a record structure looking for a specific value.

Final Thoughts

There are multiple purposes for using record structures and we have provided a basic framework to help you understand how to: define a record structure, load the record structure, and select information that is related to values in a record structure.

Cerner Millennium solutions use standard record structures to exchange information between a front end application and the RDBMS. For example a front end application will create and populate a request record structure. The request is passed through the middleware and executes an associated Discern Explorer program (script). That program may execute a select command using the information in the request. The information returned by the select is loaded into a reply record structure that is returned by the middleware to the front end application. The front end application extracts the information from the reply and displays it to the user. Many solutions provide functionality that allows leveraging the request and reply structures in customizable Discern Explorer programs. Working with request and reply record structures is identical to working with custom record structures using the concepts discussed in the preceding sections.