Discern Explorer
Program Efficiency Checklist

Introduction

The following checklist can be used as a guide to help ensure that the queries in your *Discern Explorer* (CCL) programs will execute in an efficient manner. A detailed description of each item on the checklist is provided on the subsequent pages.

- The query performs indexed reads on tables. Full table scans are only used when the full table scan is more efficient than an indexed read.
- The query qualifies using the best index on each table that it reads.
- If needed, the query uses +0 or a function to control index usage.
- The query does not use orahints in an attempt to control index usage.
- No unnecessary queries are performed. The desired data should be retrieved in as few queries as possible.
- The program uses UARs instead of joins to the Code_Value table or selecting from the Code_Value table.
- UARs to get code values are called directly in the program instead of executing CPM_GET_CD_FOR_CDF
- *Cerner Millennium* production programs (scripts) take advantage of functionality provided by servers to get code values and textual values associated with code values
- Use the Expand() function to qualify on the values in a record structure list instead of using the Dummyt table with seq for the same purposes. When using the Expand() function to qualify on the values in a record structure list, consider using the Dummyt table to ensure that bind variables are used in the IN clause of the query is passed to the RDBMS.
- If the values in a record structure list are sorted in ascending order, use the LocateValsort() function to find a specific value in the list. If the values in a record structure list are not sorted in ascending order use the LocateVal() function to find a specific value in the record structure list.
- If the query uses the Dummyt table, it is either used as the first table in the Plan clause or in the last Join clause. The Dummyt table is not used in the middle of the Plan/Join clauses.
- The query uses the Outerjoin function in the Where clause instead of the Outerjoin or DontCare control options in the With clause when performing an outerjoin on an RDBMS table.
- Use Where Not Exists and a nested Select or Outerjoin and field = null to create an exception query on an RDBMS table. Only use the Outerjoin, DontExist control options to create an exception query on a non-RDBMS table.
- Avoid dynamically building a query using call parser() or the parser() function. Use Select IF or conditional statements to execute a compiled query instead of using parser functionality.
- Ensure proper index usage for all possible execution paths, if parser functionality is used.
- The From clause or Plan/Join clauses are structured so the tables in the query are read in the most efficient order.
- Orjoins are avoided wherever possible.
- Values are placed on the right-hand side of the operator in qualification clauses.

- Create lists in record structures using the asterisk [*] and use the alterlist() function to initialize memory instead of using a fixed [number] value and the alter() function.
- Use the VC data type when creating character items in a record structure if the average length of the character data is greater than or equal to 40.
- The binary size of the program object is less than the cache size of the server that will execute the program object.
- Set Trace RecPersist is not used. If needed With PersistScript or With Persist is used to create objects in subprograms that are made known to the calling program

The query performs indexed reads on tables. Full table scans are used only when the full table scan is more efficient than an indexed read.

Use CCLQUERY to verify that full table scans are performed by the RDBMS only when the query is expected to return over 20% of the rows from the table. The CCLQUERY program can be used to review how the RDBMS will process the queries contained in a *Discern Explorer* program. For more information on CCLQUERY refer to DiscernExplorerHelp.exe>System Reference>Useful CCL Utilities. The CCLQUERY program can be executed from the command line, from the Reports menu in DiscernVisualDeveloper.exe, added as a program to the ExplorerMenu, or executed from the Tools-Execute Program menu in Visual Explorer.

If a program named CCL_TEST is created by including the following source code:

```
drop program ccl_test go
create program ccl_test

select o.order_mnemonic
from orders o,
person p
plan o where o.order_mnemonic = "BUN"
join p where p.person_id = o.person_idwith counter
end go
```

And CCLQUERY is executed to analyze CCL_TEST, the output will contain the following lines:

```
PLAN STATEMENT FOR D_TEST7:1252586901:R006:Q01
    0    1)SELECT STATEMENT
    1     2)NESTED LOOPS
    2      3)TABLE ACCESS                    FULL           ORDERS
    3      3)TABLE ACCESS                    BY ROWID       PERSON
    4       4)INDEX                           UNIQUE SCAN    XPKPERSON
```

```
                >>> UNIQUE    PERSON_ID
```

This output from CCLQUERY shows that a full table scan was performed when reading the Orders table.

If the following source code is used to qualify using the catalog_cd field instead of the order_mnemonic field:

```
drop program ccl_test go
create program ccl_testdeclare bun_var = f8
set bun_var = uar_get_code_by("DISPLAYKEY", 200, "BUN")
select o.order_mnemonic,
p.name_full_formatted
from orders o,
person p
plan o where o.catalog_cd = BUN_VAR
join p where p.person_id = o.person_id
with counter
end go
```

CCLQUERY will show that the Orders table is read using the XIE1ORDERS index (a non-unique index on the Catalog_CD field).

```
PLAN STATEMENT FOR D_TEST7:1303528401:R006:Q01
    0    1)SELECT STATEMENT
    1     2)NESTED LOOPS
    2      3)TABLE ACCESS                    BY ROWID       ORDERS
    3       4)INDEX                           RANGE SCAN     XIE1ORDERS
              >>> NONUNIQUE CATALOG_CD
    4      3)TABLE ACCESS                    BY ROWID       PERSON
    5       4)INDEX                           UNIQUE SCAN    XPKPERSON
              >>> UNIQUE    PERSON_ID
```

Generally it is more efficient to read a table using an index than it is to perform a full table scan. However in some situations a full table scan may be more efficient. A few simple checks can be performed to determine if a full table scan may be more efficient than an indexed read. First, execute a query to determine the total number or rows on a table. Then execute a query to return the desired rows. If the number of desired rows divided by the total number of rows on the table is greater than .20, then a full table scan may be the more efficient method. This will typically occur when qualifying on a table using an _IND or _FLAG field or a _CD field with a low cardinality. For example, to determine if it is more efficient to select all COMPLETED orders from the orders table using a full table scan or an indexed scan, suppose:

```
select count(*) from orders returns 7293806

select count(*) from orders where order_status_cd = completed_var
returns 3514165
```

For this query a full table scan may be more efficient. Since 514165 / 7293806 = .48, it might be more efficient to perform a full table scan than it would be to use the index on order_status_cd when reading the Orders table.

Full table scans read the data in sequential pages. So a full table scan will get and read all records on page 1. It will then get and read all records on page 2, then page 3 and so on, until all records have been read. Indexed reads get a specific page and read a specific record from that page using an ordered set of values. So if the first value in a non-unique index is in record 9 on page 2 and also in record 15 on page 30, the index would get page 2, read record 9, then get page 30 and read record 15. If the second value in the index is in record 12 on page 2, the index would then need to go back and get page 2 and read record 12. Since the index jumps from one page where a specific value is found to another page where a specific value is found the index may read some pages multiple times. As the percentage of rows that qualify compared to the total number of rows on the table increases, the likelihood of reading pages multiple times also increases. With each page that is read multiple times the efficiency of using the index decreases. Eventually you reach a point where it becomes more efficient to read all of the pages once. At that point a full table scan will be more efficient than an indexed read. However it is generally more efficient to do an indexed read than a full table scan.

The query qualifies using the best index on each table that it reads.

Use CCLQUERY to verify that the RDBMS optimizer is using the best indexes to read the tables when the query is processed. For more information on CCLQUERY refer to DiscernExplorerHelp.exe>System Reference>Useful CCL Utilities. The CCLQUERY program can be executed from the command line, from the Reports menu in DiscernVisualDeveloper.exe, added as a program to the ExplorerMenu, or executed from the Tools>Execute Program menu in Visual Explorer.

When querying tables, the goal is to read the least number of pages and rows in order to retrieve all of the requested data. When creating a qualification, review all available indexes on the table, paying close attention to the column position (Col Pos) of the columns that form the index. To read the table using a given index, you must qualify on the columns that form the index according to their column position.

For example, the Orders table contains the following indexes:

Index Name	Unique Index Col	Col Pos
XIE2ORDERS	NONUNIQUE ENCNTR_ID	1
XIE1ORDERS	NONUNIQUE CATALOG_CD	1
XIE7ORDERS	NONUNIQUE PERSON_ID	1

	ENCNTR_ID	2
	CATALOG_CD	3
	CATALOG_TYPE_CD	4
	ACTIVITY_TYPE_CD	5

If you want to select all CBC orders for a given encounter, and you only know the catalog_cd associated with the CBC and the encntr_id associated with the encounter:

- You do not want the RDBMS to read the Orders table using the XIE1ORDERS index. Doing so will find all existing CBC orders regardless of which encounter the CBC belongs to. Over time, as new CBC orders are entered into the system, the rows associated with this index will grow and your program's performance will degrade.
- You cannot use the XIE7ORDERS index. To use the XIE7ORDERS index you must qualify on the person_id because person_id is in the first position of the index. However, if the person_id were available, the XIE7ORDERS index would be the best choice.
- In this scenario, the best index to use when reading the Orders table is the XIE2ORDERS index. Because you know the encntr_id, you can use the index to only read the rows on the Orders table where the encntr_id is equal to the encounter you are looking for. From that subset of orders, you can further qualify to only select the CBC orders. Because the total number of orders for the encounter will be less than the total number of CBC orders for all encounters, you will read fewer pages and records using the XIE2ORDERS index than you would using the XIE1ORDERS index.

If needed, the query uses +0 or a function to control index usage.

If CCLQUERY shows that an index other than the best index is used when the table is read, you can add zero to an indexed numeric field in the qualification or wrap a function around an indexed character field in the qualification to ensure the RDBMS is using the best index.

If you want to select CBC orders for a specific encounter, the qualification would look similar to the following example:

```
where o.encntr_id = encounter_var and
o.catalog_cd = cbc_var
```

Given this qualification, the RDBMS optimizer will read the Orders table either using the index on the catalog_cd field XIE10RDERS or the index on the encntr_id field XIE2ORDERS. Because you do not want to read the Orders table using the catalog_cd index, you can add zero to the catalog_cd field in the qualification to prevent that index from being used. Your qualification would look similar to the following example:

```
where o.encntr_id = encounter_var and
o.catalog_cd + 0 = cbc_var
```

To prevent an index on a character field from being used to read the table, wrap the Trim() function around the field in the qualification, as in the following example:

```
where Trim(p.name_last_key) = Name_var and
p.updt_dt_tm > cnvtdatetime(curdate -7, 0)
```

Using the Trim() function in this manner will prevent the index on name_last_key from being used, and will force the optimizer to use the updt_dt_tm index to read the Person table.

The query does not use Orahints in an attempt to control index usage.

In most cases, you should use +0 or the Trim() function instead of Orahints to control index usage. The control option, Orahint, can be added to the With clause in an attempt tell the optimizer which index to use when reading a table. However, using Orahints forces the optimizer to use a cost based approach to optimization. For cost based optimization to be effective, the Oracle Analyze command must be executed on the table on a regular basis. Because the Analyze command is not executed on a regular basis, the statistics needed by cost based optimization are often inaccurate. Using inaccurate statistics causes the optimizer to choose the wrong index or perform a full table scan. Therefore, using an Orahint could cause your query to be less efficient.

No unnecessary queries are performed. The desired data should be retrieved in as few queries as possible.

In most cases, one larger query will be more efficient than several smaller queries used to perform the same task. The following examples get encounter and person information for person records that have been updated in the past 30 days. Both of the examples get the exact same data. The first example uses two selects to get the data while the second example uses one select to get the same data. During testing, the system resources used by the first example were on average about 40% greater than the system resources used by the second example.

The following example uses two queries to get a list of persons and their encounters. Even though this example follows many of the efficiency guidelines in each of the queries, it is still an inefficient example because a single query can be used to get the same data using fewer system resources.

```
;create index variable
```

```
declare num = i4 with protect
;create record structure to store person and encounter info
record person_enc (
1 plist [*]
2 person_id = f8
2 name = vc
2 enc [*]
3 encntr_id = f8
3 encntr_type = vc
)
;select person info and place in record structure
select into "nl:"
p.person_id,
p.name_full_formatted
from
person p
where p.updt_dt_tm >= cnvtdatetime(curdate-30,0)
;ensure person has an encounter
and exists (select e.encntr_id
from encounter e
where p.person_id = e.person_id)
order by p.person_id
head report
pcnt = 0
detail
pcnt = pcnt +1
if(size(person_enc->plist,5) < pcnt)
stat = alterlist(person_enc->plist, pcnt + 49)
endif
person_enc->plist[pcnt].person_id = p.person_id
person_enc->plist[pcnt].name = p.name_full_formatted
foot report
stat = alterlist(person_enc->plist, pcnt)
with nocounter, separator=" ", format
;get encounter info for people selected above
select into "nl:"
e.encntr_id,
encntr_disp = uar_get_code_display(e.encntr_type_class_cd )
from
encounter e
;use expand function to build an in clause
;where e.person_id in (the person ids stored in the record above)
where expand(num,1,size(person_enc->plist,5),
e.person_id, person_enc->plist[num].person_id )
order by e.person_id
head e.person_id
ecnt = 0
pos = locateval(num,1,size(person_enc->plist,5),
```

```
e.person_id,person_enc->plist[num].person_id )
detail
ecnt = ecnt +1
if(size(person_enc->plist[pos].enc,5) < ecnt)
stat = alterlist(person_enc->plist[pos].enc, ecnt +9)
endif
person_enc->plist[pos].enc[ecnt].encntr_id = e.encntr_id
person_enc->plist[pos].enc[ecnt].encntr_type = encntr_disp
foot e.person_id
stat = alterlist(person_enc->plist[pos].enc, ecnt)
with nocounter, separator=" ", format
```

The following example is a more efficient example. Using a single select that joins the person and encounter tables gets the data using significantly fewer system resources.

```
;create record structure to store person and encounter info record
person_enc (
1 plist [*]
2 person_id = f8
2 name = vc
2 enc [*]
3 encntr_id = f8
3 encntr_type = vc
)
;get person and encounter info and place in record sturcture
select into "nl:"
p.person_id,
p.name_full_formatted,
e.encntr_id,
encntr_disp = uar_get_code_display(e.encntr_type_class_cd )
from
person p,
encounter e
plan p where p.updt_dt_tm >= cnvtdatetime(curdate-30,0)
join e where p.person_id = e.person_id
order by p.person_id
head report
pcnt = 0
head p.person_id
pcnt = pcnt +1
if(size(person_enc->plist,5) < pcnt)
stat = alterlist(person_enc->plist, pcnt + 49)
endif
person_enc->plist[pcnt].person_id = p.person_id
person_enc->plist[pcnt].name = p.name_full_formatted
ecnt = 0
detail
ecnt = ecnt +1
if(size(person_enc->plist[pcnt].enc,5) < ecnt)
```

```
stat = alterlist(person_enc->plist[pcnt].enc, ecnt +9)
endif
person_enc->plist[pcnt].enc[ecnt].encntr_id = e.encntr_id
person_enc->plist[pcnt].enc[ecnt].encntr_type = encntr_disp
foot p.person_id
stat = alterlist(person_enc->plist[pcnt].enc, ecnt)
foot report
stat = alterlist(person_enc->plist, pcnt)
with nocounter, separator=" ", format
```

The program uses UARs instead of joins to the Code_Value table or selecting from the Code_Value table.

Joins to the Code_Value table and selects from the Code_Value table should be avoided. Instead, you should use UARs that get information from cached memory.

For example, use UAR_GET_CODE_DISPLAY, UAR_GET_CODE_MEANING, or UAR_GET_CODE_DISCRIPTION to get the text string that is associated with a coded field instead of joining to the Code_Value table. The following code segment demonstrates how these UARs are used:

```
select disp = UAR_GET_CODE_DISPLAY(p.sex_cd),
desc = UAR_GET_CODE_DESCRIPTION(p.sex_cd),
mean = UAR_GET_CODE_MEANING(p.sex_cd),
p.sex_cd
from
person p
where p.sex_cd >0.0
```

Use UAR_GET_MEANING_BY_CODESET or UAR_GET_CODE_BY to set a variable equal to a code value that is used in a qualification. The following code segments show how to use the UARs to set a variable (fvar) equal to the code value for the order status code of FUTURE and then use it (fvar) in a qualification in the select:

```
;example 1
declare fvar = f8 with
constant(UAR_GET_CODE_BY("MEANING", 6004, "FUTURE")),protect
select into $1
o.order_id,
o_order_status_disp =
UAR_GET_CODE_DISPLAY( o.order_status_cd ),
o_catalog_disp = UAR_GET_CODE_DISPLAY( o.catalog_cd )
from orders o
where o.order_status_cd = fvar
```

```
;example 2
declare fvar = f8
declare stat = i4
set stat = UAR_GET_MEANING_BY_CODESET(6004, "FUTURE", 1, fvar)
select into $1
o.order_id,
o_order_status_disp =
UAR_GET_CODE_DISPLAY( o.order_status_cd ),
o_catalog_disp = UAR_GET_CODE_DISPLAY( o.catalog_cd )
from orders o
where o.order_status_cd = fvar
```

In most cases, setting a variable using one of the UARs, and qualifying where the coded field is equal to the variable, is the most efficient method. However, in some cases it may be more efficient to use a nested select that reads the code_value table. For example, if you needed to get information about people with encounters, where the location code for the encounter is a nursing unit and the display value of the nursing unit begins with 2W, you cannot use the UARs. The UARs do not allow you to get a list of code_values where the cdf_meaning is NURSEUNIT and the display_key is 2W*. You could use UAR_GET_MEAING_BY_CODESET to get a list of code_values where the cdf_meaning is NURSEUNIT, but it does not allow you to do the additional qualification on display_key. For this specific type of issue, a nested select that reads the Code_Value table might be the most efficient way to qualify on the information you are looking for. The following example shows how to use a nested select to get information about people with encounters where the location code for the encounter is a nursing unit and the display value of the nursing unit begins with 2W:

```
select into "nl:"
p.person_id,
p.name_full_formatted,
e_loc_disp = UAR_GET_CODE_DISPLAY( e.location_cd ),
e_encntr_disp = UAR_GET_CODE_DISPLAY(e.encntr_type_cd)
from encounter e,
person p
plan e where exists
(select cv1.code_value
from code_value cv1
where cv1.code_value = e.location_cd and
cv1.code_set = 220
and trim(cv1.cdf_meaning) = "NURSEUNIT"
and cv1.active_ind = 1
and cv1.begin_effective_dt_tm<=cnvtdatetime(curdate,curtime3)
and cv1.end_effective_dt_tm>= cnvtdatetime(curdate,curtime3)
and cv1.display_key = "2W*")
join p where p.person_id = e.person_id
with counter
```

Situations where a direct select from the Code_Value table is needed should be rare. In most circumstances, reads against the Code_Value table can be replaced by using a UAR. Generally, using a UAR instead of reading the Code_Value table will improve performance.

UARs to get code values are called directly in the program instead of executing CPM_GET_CD_FOR_CDF

Generally it is more efficient to call the UARs directly in the program than it is to execute CPM_GET_CD_FOR_CDF.

For example use:

```
declare mrn_var = f8 with Constant(UAR_GET_CODE_BY("MEANING",4,"MRN"))
instead of:
set cdf_meaning = MRN
set code_value = 0.0
set code_set = 4
execute CPM_GET_CD_FOR_CDF
set mrn_var = code_value
```

Cerner Millennium production programs (scripts) take advantage of functionality provided by servers to get code values and textual values associated with code values

This subject only applies to *Cerner Millennium* production programs that are executed via the middleware servers.

Many servers create and populate the ReqData record structure. This record structure contains commonly used code_values and should be used instead of UAR calls or selects from the Code_Value table. Call EchoRecord(ReqData) can be used to examine the contents of the ReqData record structure.

Cerner Millennium production programs should use the auto populate feature of the Reply record to have the middleware provide the textual values associated with a code _value instead of UAR calls or selects from the Code_Value table when the textual value is only needed by the front end application. The middleware will automatically populate the textual values associated with a code_value when it encounters a specific pattern in the reply definition. When the reply contains an f8 field with a name that ends in _CD, followed by a c40 field with a name that ends in _DISP, the middleware will automatically get the display text for the code_value in the _CD field and assign it to the _DISP field. If the _DISP field is followed by a c60 field with a name that ends in _DESC, the description will be assigned to the _DESC field. If the _DESC field is followed by a c12 field with a name that ends in _MEAN, the cdf_meaning will be assigned

to the _MEAN field. This functionality of the middleware allows the *Discern Explorer* program to simply define the reply and assign a code_value to the _CD field in the reply.

In the following example the middleware will assign the display value, description, and cdf_meaning for the sex_cd to the appropriate fields in the reply.

```
record reply (
1 sex_cd = f8
1 sex_disp = c40
1 sex_desc = c60
1 sex_mean = c12 )
select p.sex_cd
from person p
detail
reply->sex_cd = p.sex_cd
```

Using the above definition of the reply, the program only needs to set the reply->sex_cd field. The middleware populates the remaining reply items.

The reply must follow the specified pattern from the top down. If the _MEAN or _DESC are not needed, they could be eliminated from the reply definition. But if the _MEAN is needed the _DISP and _DESC must be included in the reply definition.

Use the Expand() function to qualify on the values in a record structure list instead of using the Dummyt table with seq for the same purposes. When using the Expand() function to qualify on the values in a record structure list, consider using the Dummyt table to ensure that bind variables are used in the IN clause of the query that is passed to the RDBMS.

Situations often arise where you need to select information that is related to one of the values stored in a record structure list. Two methods are commonly used pull the values out of a record structure list and use them in the qualifications of a select command. The older method defines the Dummyt table in the

From clause of the select using With Seq = *number of items in the list* option. The newer method uses the Expand() function.

The following example shows a segment of code using the Dummyt table With Seq = number of items in the list method.

```
select into "nl:"
from
(dummyt d with seq =size(input->entity_qual_cnt,5)),
encounter e
plan d join e where e.encntr_id =
input->entity_qual_cnt[d.seq].entity_id
with nocounter
```

The problem with the Dummyt table With Seq = *number of items in the list* method is that it executes a query at the RDBMS level one time for every value in the record structure list. The item from the record structure list will be passed as a bind variable in the RDBMS query. If in the above example, the input->entity_qual_cnt list contained 4 entity_ids, the following query would be executed by the RDBMS 4 times:

```
select e.encntr_id from encounter e where e.encntr_id = :1
```

Each time the query was executed, :1 would be assigned one of the entity_ids. The RDBMS query is exactly the same each time that it is executed, allowing the RDBMS to execute the query using the cached execution plan. However, executing the RDBMS query one time for each value in the record structure list generally uses more system resources than would be used if the query were executed once and selected all rows that matched one of the values in the list.

Using the Expand() function method causes *Discern Explorer* to only pass the query to the RDBMS one time. The Expand() is converted to an IN clause. To take advantage of RDBMS caching, *Discern Explorer* will assign the values from the record structure list to bind variables. The IN clause that is created will then use the bind variables. *Discern Explorer* will also pad the number of bind variables that are passed in the IN clause. If the record structure list contains 5 or less values, the number of bind variables passed to the IN clause will be padded to 5. If the record structure list contains more than 5 values, and a single search expression is used, the number of bind variables passed to the IN clause will be padded up to the next multiple of 20. If multiple search expressions are used, the number of bind variables passed to the IN clause will be padded up to the next multiple of 10. The padded bind variables will be set equal to the value of the last item in the record structure list. *Discern Explorer* limits the number of bind variables that will be passed in the IN clauses generated by the Expand() function to 200. If more than 200 items will be placed in the IN clauses generated by the Expand() function, *Discern Explorer* will pass the literal values instead of using bind variables.

The following example shows a segment of code that uses the Expand() method

```
select into "nl:"
from encounter e
where expand(num,1, size(input->entity_qual_cnt,5),
e.encntr_id, input->entity_qual_cnt[num].entity_id)
with nocounter
```

If in the above example, the input->entity_qual_cnt list contained 5 or less entity_ids, the Expand() would be converted to:

```
select e.encntr_id from encounter e
where e.encntr_id IN (:1, :2, :3, :4, :5)
```

If the input->entity_qual_cnt list contained 4 entity_ids, bind variables :1 through :4 would be assigned one of those entity_ids. Bind variable :5 would be assigned the same value as bind variable :4. This allows the same execution plan to be used by the RDBMS when the input->entity_qual_cnt list contains from 1 to 5 entity_ids.

If in the above example, the input->entity_qual_cnt list contained 6 to 20 entity_ids, the Expand() would be converted to:

```
select e.encntr_id from encounter e
where e.encntr_id IN
(:1, :2, :3, :4, :5, :6, :7, :8, :9, :10,
:11, :12, :13, :14, :15, :16, :17, :18, :19, :20)
```

If in the above example, the input->entity_qual_cnt list contained 21 to 40 entity_ids, the Expand() would be converted to:

```
select e.encntr_id from encounter e where
e.encntr_id IN
(:1, :2, :3, :4, :5, :6, :7, :8, :9, :10,
:11, :12, :13, :14, :15, :16, :17, :18, :19, :20,
:21, :22, :23, :24, :25, :26, :27, :28, :29, :30,
:31, :32, :33, :34, :35, :36, :37, :38, :39, :40)
```

In the above example, if the input->entity_qual_cnt list contained 35 entity_ids, bind variables :1 through :35 would be assigned one of the values from the list. Bind variables :36 through :40 would set equal to the same value as bind variable :35.

Using bind variables and padding the number of bind variables that are passed in the IN clause takes advantage of the RDBMS caching of execution plans. If these methods were not used, the RDBMS would have to create and cache a new execution plan almost every time a query that used the Expand() function was executed.

Discern Explorer will continue to use bind variables and pad the number of bind variables passed in the IN clause to the next multiple of 10 or 20 until the total number of bind variables in the IN clause exceeds 200. If the total number of bind variables in the IN clause exceeds 200, literals will be passed instead of bind variables. Using literals in the IN clause instead of bind variables will almost always prevent the RDBMS from using a cached execution plan when performing the query. In situations where the number of items in the record structure list may exceed the 200 bind variable limit, consider using the Dummyt table to force multiple executions of the query at the RDBMS where each execution will use 200 or fewer bind variables in the IN clause. Using the dummyt table in this manner will allow each execution of the query at the RDBMS level to take advantage of cached execution plans. When a query is executed frequently enough to keep the execution plan in the cache ensuring the query that is passed to the RDBMS always contains the same number of bind variables will allow the RDBMS to use the cached version of the optimized query and prevent several queries that are nearly identical from taking up space in the cache.

The following example uses the Expand() function along with a join to the Dummyt table to ensure that bind variables are used instead of literals and the number of bind variables passed in the IN clause will always be the same. Using a combination of the Expand() function and the Dummyt table to ensure the same query is always passed to the RDBMS is a preferred and recommended method. However, this method cannot be combined with the NOT operator to create a NOT IN qualification.

The following example assumes that a process has already completed that defined the Person_Enc record structure and placed a set of person ids and names in the plist list.

```
record person_enc (
1 plist [*]
2 person_id = f8
2 name = vc
2 enc [*]
3 encntr_id = f8
3 encntr_type = vc
)
```

The example will get the encounter id and encounter type for each person id and place them in the record structure as well.

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.

Expand_Size is the number of positions of the record structure list that values will be pulled from.

- The values that are pulled from the record structure list are assigned to bind variables.
- There can be up to a total of 200 bind variables in the IN clause created by the Expand() function.
- 200 should be evenly divisible by Expand_Size.
- If a single search expression is used in the Expand() use 200 as the Expand_Size.
- If two search expressions are used in the Expand() use 100 as the Expand_Size.
- If three or four search expressions are used in the Expand() use 50 as the Expand_Size.
- If five search expressions are used in the Expand() use 40 as the Expand_Size.
- If six to ten search expressions are used in the Expand() use 20 as the Expand_Size.

Expand_Total is the padded size of the record structure list and should be a multiple of expand_size.

Expand_Start

- Will be passed to the Expand() function and will be used as the starting position of the record structure list.
- Expand_start must be initialized to value greater than 0 (zero).
- It is generally initialized to 1 (one) and is then reset in the join to the Dummyt.
- The internal processing that *Discern Explorer* uses to build the IN clause and reset expand_start in the join to Dummyt requires that expand_start be initialized to the starting value prior to 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 intitialized to the expand_size and is then reset in the join to the Dummyt or calculated in the Expand() function call.

- The internal processing that *Discern Explorer* uses to build the IN clause and reset expand_stop in the join to Dummyt requires that expand_stop be initialized to the stop value prior to the select that uses it.

```
set actual_size = size(person_enc->plist,5)
set expand_size = 200
set expand_total = actual_size +
(expand_size - mod(actual_size,expand_size))
set expand_start = 1
set expand_stop = 200
set num = 0
;increase the size of the list to expand_total
;increasing the list size allows always passing the
;same number of elements in the IN clause
;that is generated by the expand()
set stat = alterlist(person_enc->plist, expand_total)
;set the added positions of list equal to the last
;item in the list
for(idx = actual_size+1 to expand_total)
set person_enc->plist[idx].person_id =
person_enc->plist[actual_size].person_id
endfor
select into "nl:"
e.encntr_id,
encntr_disp = uar_get_code_display(e.encntr_type_class_cd)
from
encounter e,
(dummyt d with seq = value(expand_total/expand_size))
;Use dummyt to execute query with expand_size number of
;elements in the IN clause.
;Expand_total/expand_size returns the number of times
;the query will be executed.
;Expand_start and expand_stop are reset using assign()
plan d where
assign(expand_start,evaluate(d.seq,1,1,expand_start+expand_size))
and assign(expand_stop,expand_start+(expand_size-1))
join e where expand(num,expand_start,expand_stop,
e.person_id, person_enc->plist[num].person_id )
order by e.person_id
head report
;remove the padded positions from the list
stat = alterlist(person_enc->plist, actual_size)
head e.person_id
ecnt = 0
;find the position of the person_id in the record structure pos = locateval(num,1,actua
 e.person_id,person_enc->plist[num].person_id )
;if the person_ids in the record structure are in sorted
;order, use locatevalsort() instead of locateval()
 e.person_id,person_enc->plist[num].person_id )
```

```
detail
ecnt = ecnt +1
if(size(person_enc->plist[pos].enc,5) < ecnt)
stat = alterlist(person_enc->plist[pos].enc, ecnt +9)
endif
;place the encounter information in the record structure
person_enc->plist[pos].enc[ecnt].encntr_id = e.encntr_id
person_enc->plist[pos].enc[ecnt].encntr_type = encntr_disp
foot e.person_id
stat = alterlist(person_enc->plist[pos].enc, ecnt)
foot report
row +0
with nocounter
```

Using the Expand() and Dummyt method shown above would pass the following query to the RDBMS:

```
SELECT /*+ CCL<SAME_EXPAND_SIZE_EXAM:S9999:O1> */
E.PERSON_ID,E.ENCNTR_ID, E.ENCNTR_TYPE_CLASS_CD
FROM ENCOUNTER E WHERE ((E.PERSON_ID IN
(:1,  :2,  :3,  :4,  :5,  :6,  :7,  :8,  :9,  :10,
 :11, :12, :13, :14, :15, :16, :17, :18, :19, :20,
 :21, :22, :23, :24, :25, :26, :27, :28, :29, :30,
 :31, :32, :33, :34, :35, :36, :37, :38, :39, :40,
 :41, :42, :43, :44, :45, :46, :47, :48, :49, :50,
 :51, :52, :53, :54, :55, :56, :57, :58, :59, :60,
 :61, :62, :63, :64, :65, :66, :67, :68, :69, :70,
 :71, :72, :73, :74, :75, :76, :77, :78, :79, :80,
 :81, :82, :83, :84, :85, :86, :87, :88, :89, :90,
 :91, :92, :93, :94, :95, :96, :97, :98, :99, :100,
 :101,:102,:103,:104,:105,:106,:107,:108,:109,:110,
 :111,:112,:113,:114,:115,:116,:117,:118,:119,:120,
 :121,:122,:123,:124,:125,:126,:127,:128,:129,:130,
 :131,:132,:133,:134,:135,:136,:137,:138,:139,:140,
 :141,:142,:143,:144,:145,:146,:147,:148,:149,:150,
 :151,:152,:153,:154,:155,:156,:157,:158,:159,:160,
 :161,:162,:163,:164,:165,:166,:167,:168,:169,:170,
 :171,:172,:173,:174,:175,:176,:177,:178,:179,:180,
 :181,:182,:183,:184,:185,:186,:187,:188,:189,:190,
 :191,:192,:193,:194,:195,:196,:197,:198,:199,:200)))
```

There will always be 200 bind variables in the IN clause created by the Expand(). If the actual size of person_enc->plist is not equal to a multiple of 200, its size will be increased to the next multiple of 200. The last person_id in the list will be copied to the new positions of the list. This may seem counterintuitive because extra bind variables of the same value are passed into the query. The advantage of passing the extra bind variables will be more effective caching by the RDBMS. If the person_enc->plist were not padded, then a query would be created with a different number of bind variables. For example, if person_enc->plist contained 290 elements, its size would be increased to 400. This would result in the query being executed 2 times (400/200 = 2). Each time the query is executed, 200 of the values in the person_enc->plist will be assigned to the 200 bind variables. The RDBMS will see the same query

2 times, but will only have to optimize it 1 time. If the size of the person_enc->plist is not padded to a multiple of 200, or the Dummyt table is not used to ensure that 200 bind variables are always passed in the IN clause, then each time the program is executed a new query will most likely be passed to the RDBMS. For example if the program is executed and person_enc->plist contains 290 elements, a query with 290 literal values would be passed to the RDBMS. If the program is executed a second time and the person_enc->plist contains 291 elements a query with 291 literal values would be passed to the RDBMS. Since this query has a different number of literal values in the IN clause, the optimizer treats it as a brand new query and goes through the complete optimization process to generate an execution plan for the query. The execution plan for this query would be placed in the cache, taking the place of another query. Ensuring the query that is passed to the RDBMS always contains the same number of bind variables allows us to take full advantage of the RDBMS caching and execute the query using the cached execution plan instead of creating and caching a new execution plan for each query that has a different number of bind variables in the IN clause or uses literal values in the IN clause. Once the query is executed, ensure that the size of the record structure list is set back to the actual size either in the head or foot report sections or at the end of the query.

The general recommendation is to pad the record structure list and use the Expand() function along with a join to the Dummyt table to ensure that the query that is passed to the RDBMS will use a consistent number of bind variables in the IN clause. However, if the query is not executed often enough to keep the execution plan in the cache, passing a different number of bind variables or using literals in the IN clause does not cause a performance problem. If the execution plan for the query is not in the cache the RDBMS will need to go through the full optimization process any way. If the query is going to be executed less than once a week then you might consider just using the Expand() without padding the record structure list and using the join to the Dummyt table.

If the query is using a NOT Expand(), this method of using the Dummyt table to ensure that 200 or less bind variables are passed in the IN clause should not be used. For example suppose in the following example that person_enc->plist contained 400 person_ids.

```
set actual_size  = size(person_enc->plist,5)
set expand_size  = 200
set expand_total = actual_size +
 (expand_size - mod(actual_size,expand_size))
set expand_start = 1
set expand_stop  = 200
set num = 0

select into "nl:"
 e.encntr_id,
 encntr_disp = uar_get_code_display(e.encntr_type_class_cd)
from
 encounter  e,
 (dummyt d with seq = value(expand_total/expand_size))
 ;Use dummyt to execute query with expand_size number of
 ;elements in the IN clause.
 ;Expand_total/expand_size returns the number of times
 ;the query will be executed.
 ;Expand_start and expand_stop are reset using assign()
plan d where
assign(expand_start,evaluate(d.seq,1,1,expand_start+expand_size))
```

```
 and assign(expand_stop,expand_start+(expand_size-1))
join e  where NOT expand(num,expand_start,expand_stop,
  e.person_id, person_enc->plist[num].person_id )
```

The above would cause a query containing the following to be executed two times by the RDBMS:

```
SELECT  /*+  CCL<SAME_EXPAND_SIZE_EXAM:S9999:O1> */
E.PERSON_ID,E.ENCNTR_ID, E.ENCNTR_TYPE_CLASS_CD
 FROM  ENCOUNTER E  WHERE NOT ((E.PERSON_ID IN
(:1,  :2,  :3,  :4,  :5,  :6,  :7,  :8,  :9,  :10,
 :11, :12, :13, :14, :15, :16, :17, :18, :19, :20,
…
 :191,:192,:193,:194,:195,:196,:197,:198,:199,:200)))
```

The first execution of the above query would qualify all encntr_ids that were not equal to one of the 200 values that were assigned to bind variables :1 through :200. The second execution of the query would qualify all encntr_ids that were not equal to one of the 200 values that were assigned to the bind variables :1 through :200. This would include encntr_ids that were assigned to one of the bind variables in the first execution of the query. So in effect all encntr_ids would be returned because every encntr_id would not be in the IN clause of one of the queries that are executed.

The next example assumes the following record structure has already been created and populated with person and encounter IDs.

```
record person_enc (
  1 elist [*]
    2 person_id = f8
    2 encntr_id = f8
    2 encntr_type = vc
)
```

The next example shows usage of the Expand() function with multiple search expressions. It also pads the record structure list and uses the Dummyt table to ensure that the same number of bind variables is always passed to the RDBMS. This example uses the Initarray() function instead of the Assign() function to set expand_start for each iteration. Either function can be used to achieve the same results. The expand stop index is calculated within the expand function.

```
declare Actual_size = i4 with
 protect,
 noconstant(size(person_enc->elist,5))
declare Expand_size = i2 with protect, constant(100)
declare Expand_start = i4 with protect, noconstant(1)
declare Expand_stop = i4 with protect, noconstant(100)
declare Expand_total = i4 with protect,
 noconstant(actual_size+(expand_size -
   mod(actual_size,expand_size)))
declare num = i4 with protect, noconstant(0)
```

```
;increase the size of the list to expand_total
set stat = alterlist(person_enc->elist, expand_total)

;set the added positions of list equal to the last item in the list
for(idx = actual_size+1 to expand_total)
 set person_enc->elist[idx].person_id =
  person_enc->elist[actual_size].person_id
 set person_enc->elist[idx].encntr_id =
  person_enc->elist[actual_size].encntr_id
endfor

select into "nl:"
 e.person_id,
 e.encntr_id,
 encntr_disp = uar_get_code_display( e.encntr_type_class_cd )

from
 encounter  e,
 (dummyt d1 with seq = value(expand_total/expand_size))
plan d1 where initarray(expand_start,
              evaluate(d1.seq,1,1, expand_start +expand_size))
join e  where expand(num,expand_start,
  expand_start + (expand_size-1),
  e.person_id, person_enc->elist[num].person_id,
  e.encntr_id, person_enc->elist[num].encntr_id  )

order by e.person_id
```

The statement that will be passed to RDBMS for the above query is:

```
  SELECT  /*+ CCL<SAME_EXPAND_SIZE_2_ITEMS:S9999:O1:Q02.1>*/
E.PERSON_ID,E.ENCNTR_ID, E.ENCNTR_TYPE_CLASS_CD
FROM  ENCOUNTER E  WHERE (E.PERSON_ID,E.ENCNTR_ID) IN
((:1  ,:2  ),(:3  ,:4  ),(:5  ,:6  ),(:7  ,:8  ),(:9  ,:10  ),
 (:11 ,:12 ),(:13 ,:14 ),(:15 ,:16 ),(:17 ,:18 ),(:19 ,:20  ),
 (:21 ,:22 ),(:23 ,:24 ),(:25 ,:26 ),(:27 ,:28 ),(:29 ,:30  ),
 (:31 ,:32 ),(:33 ,:34 ),(:35 ,:36 ),(:37 ,:38 ),(:39 ,:40  ),
 (:41 ,:42 ),(:43 ,:44 ),(:45 ,:46 ),(:47 ,:48 ),(:49 ,:50  ),
 (:51 ,:52 ),(:53 ,:54 ),(:55 ,:56 ),(:57 ,:58 ),(:59 ,:60  ),
 (:61 ,:62 ),(:63 ,:64 ),(:65 ,:66 ),(:67 ,:68 ),(:69 ,:70  ),
 (:71 ,:72 ),(:73 ,:74 ),(:75 ,:76 ),(:77 ,:78 ),(:79 ,:80  ),
 (:81 ,:82 ),(:83 ,:84 ),(:85 ,:86 ),(:87 ,:88 ),(:89 ,:90  ),
 (:91 ,:92 ),(:93 ,:94 ),(:95 ,:96 ),(:97 ,:98 ),(:99 ,:100 ),
 (:101,:102),(:103,:104),(:105,:106),(:107,:108),(:109,:110 ),
 (:111,:112),(:113,:114),(:115,:116),(:117,:118),(:119,:120 ),
 (:121,:122),(:123,:124),(:125,:126),(:127,:128),(:129,:130 ),
 (:131,:132),(:133,:134),(:135,:136),(:137,:138),(:139,:140 ),
```

```
(:141,:142),(:143,:144),(:145,:146),(:147,:148),(:149,:150 ),
(:151,:152),(:153,:154),(:155,:156),(:157,:158),(:159,:160 ),
(:161,:162),(:163,:164),(:165,:166),(:167,:168),(:169,:170 ),
(:171,:172),(:173,:174),(:175,:176),(:177,:178),(:179,:180 ),
(:181,:182),(:183,:184),(:185,:186),(:187,:188),(:189,:190 ),
(:191,:192),(:193,:194),(:195,:196),(:197,:198),(:199,:200 ))
```

If the values in a record structure list are sorted in ascending order, use the LocateValsort() function to find a specific value in the list. If the values in a record structure list are not sorted in ascending order use the LocateVal() function to find a specific value in the record structure list.

The LocateVal() and LocateValsort() functions are used to find the location subscript of a value in a record structure list. If the values in the record structure list are sorted in ascending order, the LocateValsort() function should be used to locate a specific value. If the values in the record structure list are not sorted in ascending order, the LocateVal() function should be used to locate a specific value.

The second and third parameters passed to both the LocateVal() and LocateValsort() functions are a starting subscript and stopping subscript. LocateVal() performs a sequential search of the record structure list reading every value from the starting index until it either finds the value or reaches the stopping index. LocateValsort() performs a binary search which will look at the value that is half way between the starting subscript and the stopping subscript. If the value at that location is less than the searched for value, LocateValsort() will look at the value that is half way between the current location and the starting subscript. It will continue to look at the value that is half way between two possible locations until the searched for value is found. For example if you had the following record structure definition:

```
record rec (
  1 list [*]
    2 fld = i4
)
```

and the following values were stored in the rec->list[subscript].fld

Subscript	Value
1	A
2	B
3	C
4	D
5	E
6	F
7	G

8	H
9	I

Both of the following commands would set the variable pos equal to 7:

```
set pos = LocateVal(num,1,9,"G",rec->list[num].fld)
set pos = LocateValsort(num,1,9,"G",rec->list[num].fld)
```

Using the above, LocateVal() would read all of the values from subscript 1 through 7 looking for "G".

Using the above, LocateValsort() would first read the value at subscript 5 since 5 is half way between 1 and 9. Since E is less than G, LocateValsort() would next read the value at subscript 7 because 7 is halfway between 5 and 9. Since the value G is found at subscript 7, pos is set equal to 7.

Both of the following commands would set the variable pos equal to 4:

```
set pos = LocateVal(num,1,9,"D",rec->list[num].fld)
set pos = LocateValsort(num,1,9,"D",rec->list[num].fld)
```

Using the above, LocateVal() would read all of the values from subscript 1 through 4 looking for "D".

Using the above, LocateValsort() would first read the value at subscript 5 since 5 is half way between 1 and 9. Since E is greater than D, LocateValsort() would next read the value at subscript 3 because 3 is halfway between 1 and 5. Since C is less than D, LocateValsort() would read the value at subscript 4 because 4 is halfway between 3 and 5. Since the value D is found at subscript 4, pos is set equal to 4.

On average, the binary search on a record structure list containing sorted values will read fewer values than a sequential search. So on average the LocateValsort() will outperform LocateVal() when the values in the list are in sorted order.

If the values in a record structure list are not unique, the LocateVal() function will find the first instance of the value that is being searched for. Subsequent executions of the LocateVal() can be used to find the additional locations of the value. Each subsequent execution of the LocateVal() function would need to use a starting location that is one greater than the last location where the value was found. The following example will find all the locations of the string "value" in the record structure list.

```
;find the first occurrence of "value" in the list
set first = locateval(num,1,size(rec->list,5), "value",
     rec->list[num].fld)
call echo(first)
;find the subsequent occurrences of "value" in the list
set next = first
while(next != 0 and next <= size(rec->list,5) )
 set next = locateval(num,next+1,size(rec->list,5), "value",
     rec->list[num].fld)
 call echo(next)
endwhile
```

If the values in a record structure list are not unique, the LocateValsort() function will find one occurrence of the value that is being searched for in the record structure list. Since the values in the record

structure list are sorted, all occurrences of the searched for value will be in a contiguous group. Since the LocateValsort() function performs a binary search, it will find one of the occurrences of the value. Additional occurrences of the value could be before or after the occurrence that was found. The following example uses the LocateValsort() function to find an occurrence of the string “value”. It then uses While loops to look for additional occurrences of the string “value” before and after the location that was found by the LocateValsort() function.

```
;find an occurrence of "value" in the list
set first = locatevalsort(num,1, size(rec->list,5),
    "value",rec->list[num].fld)
set last = first
call echo(first)
set prev = first -1
set next = first +1
;find the subsequent occurrences of "value" in the list
if(next <= size(rec->list,5))
 while(next <= size(rec->list,5)
   and rec->list[first].fld = rec->list[next].fld )
  call echo(next)
  set next = next +1
 endwhile
 set last = next -1
endif
;find the prior occurrences of "value" in the list
while(rec->list[first].fld = rec->list[prev].fld and prev != 0)
 call echo(prev)
 set prev = prev -1
endwhile
set first = prev +1
```

The locateval function will only find the first instance of the search_item that is being searched for. When it is desirable to use the locateval function on an array or array combination that is not filled with unique values, it is necessary to keep track of where the last index was found. The following code segment shows an example dealing with a non-unique list. Here the output structure has already been partially filled out. This query is effectively an outerjoin to retrieve the surgeon or anesthesiologist for each surgical case.

If the query uses the Dummyt table, it is either used as the first table in the Plan clause or in the last Join clause. The Dummyt table is not used in the middle of the Plan/Join clauses.

When a Join to the dummyt table is placed in the middle of the Plan/Join clauses, the *Discern Explorer* query will be split up and passed to the RDBMS as multiple queries. Generally, it is more efficient to pass one query to the RDBMS. Passing multiple queries to the RDBMS can result in decreased performance compared to using a method that only passes a single query to the RDBMS.

In the following example, only one query is passed to the RDBMS because the dummyt table is used first in the plan clause. This query will get the aliases for all person ids that are stored in a record structure named Person.

```
select into "nl:"
pa.alias
from (dummyt d with seq = value(size(person->person,5))),
person_alias pa
plan d
join pa where person->person[d.seq].id = pa.person_id
```

The following example will select information from the encounter table where the encounter was registered during an even hour in the last 30 days. Because the EVEN() function is not supported in the WHERE clause of the RDBMS, the dummyt table is used so the EVEN() function can be applied at the Discern Explorer level to only qualify encounters that were registered during even hours. Because the dummyt table is used as the last table in the Plan/Join, only one query will be passed to the RDBMS.

```
select into $1
e.encntr_id,
e.reg_dt_tm "DD-MMM-YYYY HH:MM;;D"
from encounter e,
dummyt d1
plan e where e.reg_dt_tm between
cnvtdatetime( curdate-30, curtime3 ) and
cnvtdatetime( curdate, curtime3 )
join d1 where even(cnvtint(hour( e.reg_dt_tm ))) = 1
```

The following example should not be used because it passes three queries to the RDBMS. The same functionality can be accomplished without the Joins to the dummyt tables by using the Outerjoin() function.

```
select p.name_full_formatted,
e.encntr_type_cd,
o.catalog_cd
from person p,
```

```
encounter e,
orders o,
dummyt d1,
dummyt d2
plan p
join d1
join e where p.person_id = e.person_id
join d2
join o where e.encntr_id = o.encntr_id
with outerjoin = d1, outerjoin = d2, nocounter
```

In rare cases, when processing a large complex query, the RDBMS optimizer may not choose the best execution plan for the query. In these cases, adding a Join to the dummyt table in the middle of the Plan/Join clauses may improve performance. Occasionally, splitting a large query into two smaller queries may allow the optimizer to do a better job of processing the smaller queries than it does when processing the larger ones. However, it is generally more efficient to only pass one query to the RDBMS.

The query uses the Outerjoin function in the Where clause instead of the Outerjoin or DontCare control options in the With clause when performing an outerjoin on an RDBMS table.

Prior to using an Outerjoin, verify that the Outerjoin is actually needed. The HNA Millennium system adds a zero row to tables where the primary key is a field that ends in _ID or _CD. In a zero row, the primary key field will be equal to zero all other fields should be null. These zero rows can be used to eliminate Outerjoins.

For example, suppose you are creating a pathology case report and you want to display the name of the resident that was responsible for a task on the case report. The Report_Task table stores the Person_ID of the resident that is responsible for the task in the Responsible_Resident_ID field. If there is no resident responsible for the task, the field will default to zero (0.0). Therefore, the following code segment could be used for this report:

```
select
from
case_report cr
,pathology_case pc
,report_task rt
,prsnl p
plan cr
```

```
join pc where cr.case_id = pc.case_id
join rt where cr.report_id = rt.report_id
join p where rt.responsible_resident_id = p.person_id
```

```
select from case_report cr ,pathology_case pc ,report_task rt ,prsnl
pplan cr join pc where cr.case_id = pc.case_idjoin rt where cr.report_id =
rt.report_idjoin p where rt.responsible_resident_id = p.person_id
```

The above code segment will return data without using an Outerjoin, even if there is no resident responsible for the task. Because there is a zero row on the Prsnl table and a zero is stored in the Responsible_Resident_ID field, if there is no resident responsible for the task, the Report_Task row will be joined to the zero row on the Prsnl table. The above code segment will return data when there is a resident responsible for the task and when there is not a responsible resident for the task. No outerjoin is needed.

In *Discern Explorer*, you can use either the Outerjoin( ) function or the Outerjoin or Dontcare control options to perform Outerjoins.

Using the incorrect Outerjoin method can cause your query to be inefficient, resulting in the use of excess CPU time, buffer gets, and direct drive reads.

In a standard join, qualifying records must exist on all tables in order for the query to return data from any of the tables. The following example will only return information when the person has at least one active order and one active alias.

```
Select
p.name_full_formatted,
o.order_mnemonic,
pa.alias
from person p,
orders o,
person_alias pa
plan p
join pa where p.person_id = pa.person_id
and pa.active_ind = 1
join o where p.person_id = o.person_id
and o.active_ind = 1
with counter
```

There are times when you may need to get information when there are no matching records on one or more of the tables. For example, if you need to get the person information even if there are no orders or aliases for the person, you can either use the Outerjoin() function or the Outerjoin and Dontcare control options. However, the Outerjoin() function is generally the more efficient method.

The following example shows how to use the Outerjoin() function to get person information, even if there are no aliases or orders for the person. The Outerjoin() function is placed around any field or non-null value that you are comparing to a field on the table that may not have a matching row:

```
select p.name_full_formatted,
o.order_mnemonic,
pa.alias
from person p,
orders o,
person_alias pa
```

```
plan p
join pa where outerjoin(p.person_id) = pa.person_id
and pa.active_ind = outerjoin(1)
join o where outerjoin(p.person_id) = o.person_id
and o.active_ind = outerjoin(1)
with counter
```

The preferred method for creating an outerjoin is to use the Outerjoin() function. However, there are limitations on usage of the Outerjoin() function:

- The outerjoin() function may only be used for joining a table to a single parent table.
- The outerjoin() function may be applied only to a column or a constant.
- If multiple join conditions are present, the Outerjoin() function must be specified for every condition.
- The outerjoin() can not be used on non-RDBMS tables created using the Select Into Table command.
- The Outerjoin() function may not be combined with another condition using the OR operator.
- The Outerjoin() function may not be used with an IN condition. Note: expand() builds an in clause
- Queries may not mix the outerjoin operator with the new ANSI semantics

An alternative method for creating the Outerjoin is to use the Outerjoin, or DontCare control option. Generally the only time you would use the Dontcare or Outerjoin control options is when one of the limitations prevent you from using the Outerjoin() function.

The Outerjoin() function and the Outerjoin control option process differently. In the above example if the person does not have any aliases on the person_alias table, the query will look for orders on the order table. However, if the Outerjoin control option is used and qualifying rows do not exist on an outerjoined table, *Discern Explorer* will not look at tables that are below the outerjoined table in the Plan/Join clauses to see if they have qualifying rows. In the following example using the Outerjoin control option, if the person does not have any encounters on the encounter table, *Discern Explorer* will not look for orders on the orders table.

```
select p.name_full_formatted,
encntr_disp = uar_get_code_display(e.encntr_type_cd),
o.order_mnemonic
from person p,
encounter e,
orders o,
dummyt d1,
dummyt d2
plan p
join d1
join e where p.person_id = e.person_id and e.active_ind = 1
and e.encntr_type_cd in(inpat_var, ambulate_var)
join d2
join o where e.encntr_id = o.encntr_id and o.active_ind = 1
and o.catalog_cd in(bun_var, cbc_var)
with counter, outerjoin = d1, outerjoin = d2
```

The above example will return all rows on the person table. If the person has encounters you will get the encounter information. If the encounter has orders you will get the order information. If the person

does not have any encounters there is no reason to look on the orders table. To have orders a person must have an encounter.

The Outerjoin control option will stop the query from processing through the remainder of the query path if a qualifying row is not found on an outerjoined table. In the above example that is exactly what is wanted. If a person does not have an encounter then they cannot have an order so there is no reason to continue through the query path and read the orders table. However, in some cases you will want the processing to continue through the query path to see if there are matching rows on tables that are farther down in the plan join clauses. In those cases you will need to use the Dontcare control option. The Dontcare control option will continue to check for records on the following tables, even if there are no matching records on the table on which you placed the Dontcare. The following example shows how to use the *Discern Explorer* DontCare control option to get person and order information, even if there are no aliases for the person.

```
select p.name_full_formatted,
o.order_mnemonic,
pa.alias
from person p,
orders o,
person_alias pa,
dummyt d1,
dummyt d2
plan p
join d1
join pa where p.person_id = pa.person_id and pa.active_ind = 1
and p.person_alias_type_cd in(mrn_var, ssn_var)
join d2
join o where p.person_id = o.person_id and o.active_ind = 1
and o.catalog_cd in(bun_var, cbc_var)
with counter, dontcare = pa
```

The above example returns person and order information for each person that has at least one order. If the person has an alias, that information will also be returned. The efficiency of the above query would be improved by eliminating one of the dummyt references and moving the join to the remaining dummyt table and the person_alias table to the end of the plan join clauses. You would then use the Outerjoin control option instead of the Dontcare control option. However, if you want to get all person records even if the person did not have an order or an alias you would need to use a combination of both the Outerjoin and Dontcare control options. The following example shows how to use the Outerjoin and Dontcare control options to get all person records, if the person has a qualifying row on the orders table or on the person_alias table then that data would be returned as well. But if there is not a qualifying row on the orders or person_alias table then the data that exists on the other tables will still be returned.

```
select p.name_full_formatted,
o.order_mnemonic,
pa.alias
from person p,
orders o,
person_alias pa,
dummyt d1,
dummyt d2
plan p
```

```
join d1
join pa where p.person_id = pa.person_id and pa.active_ind = 1
and p.person_alias_type_cd in(mrn_var, ssn_var)
join d2
join o where p.person_id = o.person_id and o.active_ind = 1
and o.catalog_cd in(bun_var, cbc_var)
with counter, dontcare = pa, outerjoin = d2
```

The following example shows how to use the *Discern Explorer* Outerjoin to get person and alias information, even if there are no orders for the person.

```
select p.name_full_formatted,
o.order_mnemonic,
alias = decode(pa.seq,pa.alias)
from person p,
orders o,
person_alias pa,
dummyt d1
plan p
join pa where p.person_id = pa.person_id and pa.active_ind = 1
and p.person_alias_type_cd in(mrn_var, ssn_var)
join d1
join o where p.person_id = o.person_id and o.active_ind = 1
and o.catalog_cd in(bun_var, cbc_var)
with counter, outerjoin = d1
```

Use Where Not Exists and a nested Select or Outerjoin() and field = null to create an exception query on an RDBMS table. Only use the Outerjoin, DontExist control options to create and exception query on a non-RDBMS table.

There are times when you need to get information from one table when there are no matching records on another table. For example, you might need to get person information only when a person has no orders. To get this type of exception information, use a nested select, or Outerjoin() and field = null. The Outerjoin and DontExist control options should only be used if the exception table is not a RDBMS table. Comparing the cost, CPU time and elapsed time from the cost metrics in the following examples shows that the Outerjoin, Dontexist method is the least efficient method of creating an exception query on a RDBMS table. You must compare the cost metrics and the RDBMS system resources used by the Not

Exists and nested select method to the cost metrics and RDBMS system resources used by the Outerjoin() and field = null method to determine which of these two methods is best for each specific case. But one of these methods will always be more efficient than using the Outerjoin and DontExist control options when creating on exception query on a RDBMS table.

The following example shows how to use the nested select and the Not Exists qualifier to get person information for people that have no orders:

```
select into "nl:"
p.person_id,
name = substring(1,20,p.name_last_key)
from person p
plan p where p.name_last_key = "SMITH"
and not exists
(select o.person_id from orders o
where p.person_id = o.person_id)
with counter
```

This query produced the following CCLCOST metrics:

74929:051898 CCL_TEST_NEST_EXCP **Cost 0.12 Cpu 0.00 Ela 0.01** Bio 13 Dio 0

O0M0R0 P1R149 S1,1

The following example shows how to use Outerjoin() and field = null to get person information for people that have no orders:

```
select into "nl:"
p.person_id,
name = substring(1,20,p.name_last_key)
from person p,
orders o
plan p where p.name_last_key = "SMITH"
join o where outerjoin(p.person_id) = o.person_id
and o.person_id = null
with counter
```

This query produced the following CCLCOST metrics:

74929:051892 CCL_TEST_OJ_NULL Cost 0.17 Cpu 0.00 Ela 0.01 Bio 16 Dio 0

O0M0R0 P1R296 S1,1

The following example shows how to use the Outerjoin, Dontexist control options to get the same person information for people that have no orders:

```
select into "nl:"
p.person_id,
name = substring(1,20,p.name_last_key)
from person p,
orders o,
dummyt d2
plan p where p.name_last_key = "SMITH"
join d2
join o where p.person_id = o.person_id
```

```
with counter, outerjoin = d2, dontexist
```

This query produced the following CCLCOST metrics:

74929:051886 CCL_TEST_OJ_DONT Cost 1.20 Cpu 0.07 Ela 0.10 Bio 370 Dio 2

O0M0R0 P1R1353 S1,161

As you can see, the cost, cpu usage, elasped time, buffered i/o, and direct i/o are greater when using the Outerjoin, Dontexist than they are when using the not exists and nested select.

Avoid dynamically building a query using call parser() or the parser() function. Use Select If or conditional statements to execute a compiled query instead of using parser functionality.

Dynamically building a query using the Call Parser() command makes it very difficult to properly monitor and tune a query. It makes it difficult to ensure that the best possible query using the correct indexes is passed to the RDBMS. Using Call Parser() or the Parser() function requires that the query be created on the fly by *Discern Explorer* prior to being translated and passed to the RDBMS. Extensive use of Call Parser() and the Parser() function can cause problems with memory management.

The *Discern Explorer* Select If functionality can be used to flex any clause in the Select command. Using Select If is generally more efficient than using the Call Parser() command or the Parser() function because Select If executes the compiled clauses of the program based on the conditions. When parser functionality is used the clauses must be compiled during run-time. The following example uses the Select If functionality of *Discern Explorer* to change the Where clause, Order By clause, and the Detail section of a Select statement based on values that the user entered at a prompt named $sortp.

```
select if(cnvtupper($sortp) = "PERSON ID")
where p.person_id > 0.0
order p.person_id
detail
col 0 p.person_id
col +1 name
col +1 p.updt_dt_tm
row +1
elseif(cnvtupper($sortp) = "NAME")
where p.name_last_key = name_var
order p.name_last_key
detail
col 0 name
```

```
col +1 p.person_id
col +1 p.updt_dt_tm
row +1
endif
into $outdev
p.person_id,
name = substring(1,20,p.name_last_key),
p.updt_dt_tm ";;q"
from
person p
;if $sortp does not equal PERSON ID or NAME
;the following clauses will be used by default
where p.updt_dt_tm > cnvtdatetime(curdate-7,0)
order p.updt_dt_tm
detail
col 0 p.updt_dt_tm
col +1 p.person_id
col +1 name
row +1
with maxrec = 100 , nocounter, separator=" ", format
```

Select If is generally more efficient than using the Call Parser() command or the Parser() function.

In the following code segment, the parser is used to dynamically build a query depending on what is requested.

```
call parser ("select into 'nl:'")
call parser(from person p)
if ( GET_ENCNTRS = 1)
call parser(,encounter e)
endif
if( GET_ORDERS = 1)
call parser(",orders o")
endif
if( GET_ENCNTRS = 1)
call parser(plan p where p.person_id = user->person_id)
call parser(join e where e.person_id = p.person_id)
if( GET_ORDERS = 1)
call parser(join o where o.person_id = e.person_id
and o.encntr_id = e.encntr_id)
endif
else
call parser( where p.person_id = user->person_id)
endif
call parser( with nocounter)
```

The following code will operate with the same functionality as the above code. Using this method will make it easier to tune and measure the performance of each query. Structuring a program so that it will execute the select, as opposed to having the Call Parser() execute the select, will make it much easier to ensure

proper index usage and memory management. It also causes the compiled version of the select to be executed instead of dynamically building the select at run-time.

```
if ( GET_ENCNTRS = 1)
if( GET_ORDERS = 1)
select into nl:
from person p,
encounter e,
orders o
plan p where p.person_id = user->person_id
join e where e.person_id = p.person_id
join o where o.person_id = e.person_id
and o.encntr_id = e.encntr_id
with nocounter
else
select into nl:
from person p,
encounter e
plan p where p.person_id = user->person_id
join e where e.person_id = p.person_id
with nocounter
endif
else
select into nl:
from person p
where p.person_id = user->person_id
with nocounter
endif
```

If it is absolutely necessary to use Call Parser(), it is better to build one string using the concat() function and issue Call Parser() one time than it is to issue Call Parser() multiple times.

Issuing Call Parser() statements extensively within a program can cause problems with memory management. The Call Parser() memory can not be reclaimed until the program finishes due to the dynamic nature of building commands in a program and symbols that have now become part of the program. If a program will be making extensive use of Call Parser(), you should call a child program to perform the parser commands for a subset. The parent program can execute the child program to perform the parser. If a child program is not used, Call Parser() may grab too much memory and cause the program to crash.

Ensure proper index usage for all possible execution paths, if parser functionality is used.

When using parser functionality, it is necessary to ensure proper index usage for each possible path of execution in the script. For example, in the following code, the proper index may not be used if encounter_id != 0.

```
;checking for encounters
if( ENCNTR_ID != 0 )
set where_clause = concat( where_clause,
"nt.encounter_id = user->encounter_id ")
endif
;checking for person
if( PERSON_ID != 0 )
set where_clause = concat( where_clause,
"nt.person_id = user->person_id ")
endif
;checking for author
if( TYPE_MEAN ="PRE" and SHARED_NOTE_IND =1 and USER_ID != 0)
set where_clause = concat( where_clause,
"(( nt.author_id = user->user_id) or (nt.author_id = 0))")
elseif(user->user_id != 0 and user->type_mean = "PRE")
set where_clause = concat( where_clause,
"nt.author_id = user->user_id ")
endif
select into nl:
*
from scd_story nt
where parser( where_clause)
detail

with nocounter
```

The following code segment will ensure proper index usage by adding logic to dynamically choose the right index for all cases. It checks for cases where encounter_id != 0 and person_id != 0 and uses +0 to help the optimizer choose the correct index.

```
;checking for encounters
if( ENCNTR_ID != 0)
set where_clause = concat( where_clause,
nt.encounter_id = user->encounter_id")
endif
;checking for person
if( PERSON_ID != 0)
```

```
if(ENCNTR_ID != 0 )
set where_clause = concat( where_clause,
" nt.person_id + 0 = user->person_id ")
else
set where_clause = concat( where_clause,
" nt.person_id = user->person_id ")
endif
endif
;checking for author
if( TYPE_MEAN ="PRE" and SHARED_NOTE_IND =1 and USER_ID != 0)
if((user->encounter_id != 0 ) or (user->person_id != 0 ))
set where_clause = concat( where_clause,
"(( nt.author_id +0 = user->user_id) or
(nt.author_id + 0 = 0)) ")
else
set where_clause = concat( where_clause,
"((nt.author_id = user->user_id) or (nt.author_id = 0)) ")
endif
elseif(user->user_id != 0 and user->type_mean = "PRE")
if((user->encounter_id != 0 ) or (user->person_id != 0 ))
set where_clause = concat( where_clause,
" nt.author_id + 0= user->user_id ")
else
set where_clause = concat( where_clause,
" nt.author_id = user->user_id ")
endif
endif
select into nl:
*
from scd_story nt
where parser( where_clause)
detail

with nocounter
```

In order to view the execution plan of a query that is created at run time using parser functionality, it is necessary to issue set trace rdbplan, execute the program, and then use CCLORAPLAN. Refer to DiscernExplorerHelp.exe>System Reference>Useful CCL Utilities for more information on using CCLORAPLAN.

From the previous example, it can be seen that as the qualifications for building a parser statement become more and more complex, it becomes increasingly difficult to properly monitor and control index usage and overall query performance.

The From clause or Plan/Join clauses are structured so the tables in the query are read in the most efficient order.

The order of the tables in the FROM clause or the PLAN/JOIN clauses of a *Discern Explorer* query may affect how the RDBMS optimizer processes the query. The FROM clause or PLAN/JOIN clauses of *Discern Explorer* queries should be structured so they are processed using the most efficient execution plan.

The following rules can be used to determine if the FROM clause or PLAN/JOIN clauses are affecting optimization:

- If the query does not have PLAN/JOIN clauses, the order of the tables listed in the FROM clause will affect optimization.
- If the query has PLAN/JOIN clauses, the order of the PLAN/JOIN clauses will affect optimization.
- If the query has PLAN/JOIN clauses, the RDBRANGE control option can be used to ignore the order of the PLAN/JOIN clauses and have the order of the tables listed in the FROM clause affect optimization.

The following rules can be used to structure the FROM clause or PLAN/JOIN clauses to obtain the most efficient execution plan:

- List the tables in the FROM clause beginning with the least restrictive table and ending with the most restrictive table.
- PLAN/JOIN clauses should be structured so that you are planning on the most restrictive table and joining to the least restrictive table.
- The least restrictive table is the table that will return the most records using the broadest index or result in a larger number of buffer or disk reads.
- The most restrictive table is the table that will return the fewest records using the best index or result in the smaller number of buffer or disk reads.

The correct structuring of the PLAN/JOIN clauses or FROM clause will help the RDBMS optimizer determine the best execution plan for the query, resulting in the most efficient execution of the program.

Orjoins are avoided wherever possible.

Orjoins can be avoided by carefully architecting programs. In order to avoid Orjoins, it may be necessary to split up large queries into more manageable sub-sections. The data retrieved by these smaller queries can then be combined to produce the desired result.

The following example shows usage of the Orjoin functionality.

```
select into "nl:"
side = decode( a.seq, "ADDR", pa.seq, "ALIAS")
```

```
from person p,
address a,
person_alias pa,
dummyt d1,
dummyt d2
plan p where p.updt_dt_tm >= cnvtdatetime(curdate-7,0)
join d1
join ( a where a.parent_entity_id = p.person_id and
a.parent_entity_name = "PERSON")
orjoin d2
join ( pa where pa.person_id = p.person_id)
order by p.person_id
head report
cnt = 0
head p.person_id
add = 0
ali = 0
cnt = cnt + 1
if( mod( cnt, 50) = 1)
stat = alterlist( t_record->qual, cnt + 49)
endif
t_record->qual[cnt].person_id = p.person_id
t_record->qual[cnt].name = p.name_full_formatted
detail
if( side = "ADDR")
add = add+1
if(mod(add, 10) = 1)
stat = alterlist(t_record->qual[cnt]->address, add+9)
endif
t_record->qual[cnt].address[add].address_id = a.address_id
t_record->qual[cnt].address[add].street_addr = a.street_addr
elseif( side = "ALIAS")
ali = ali+1
if(mod(ali, 10) = 1)
stat = alterlist(t_record->qual[cnt]->alias, ali+9)
endif
t_record->qual[cnt].alias[ali].person_alias_id=pa.person_alias_id
t_record->qual[cnt].alias[ali].pa_alias = pa.alias
endif
foot p.person_id
stat = alterlist(t_record->qual[cnt]->address, add)
stat = alterlist(t_record->qual[cnt]->alias, ali)
foot report
stat = alterlist( t_record->qual, cnt)
with nocounter
```

The following code will return the same data without using the Orjoin. Two queries are used instead of an Orjoin. This method will reduce run time as well as buffer gets, disk reads, and executions.

```
;First select to get people and addresses
select into "nl:"
from person p, address a
plan p where p.updt_dt_tm >= cnvtdatetime(curdate-7,0)
join a where a.parent_entity_id = p.person_id
and a.parent_entity_name = "PERSON"
order by p.person_id
head report
cnt = 0
head p.person_id
add = 0
cnt = cnt + 1
if( mod( cnt, 50) = 1)
stat = alterlist( t_record->qual, cnt + 49)
endif
t_record->qual[cnt].person_id = p.person_id
t_record->qual[cnt].name = p.name_full_formatted
detail
add = add+1
if(mod(add, 10) = 1)
stat = alterlist(t_record->qual[cnt]->address, add+9)
endif
t_record->qual[cnt].address[add].address_id = a.address_id
t_record->qual[cnt].address[add].street_addr = a.street_addr
foot p.person_id
stat = alterlist(t_record->qual[cnt]->address, add)
foot report
stat = alterlist(t_record->qual, cnt)
with nocounter
;Second select to get people and aliases
select into "nl:"
from person p, person_alias pa
plan p where p.updt_dt_tm >= cnvtdatetime(curdate-7,0)
join pa where pa.person_id = p.person_id
head report
ali = 0
num = 0
tsize = size(t_record->qual,5)
head p.person_id
ali = 0
index = locateval(num,1,tsize, p.person_id,
t_record->qual[num].person_id)
if (index = 0)
tsize = tsize + 1
if(tsize > size(t_record->qual,5))
stat = alterlist( t_record->qual, tsize + 49)
endif
t_record->qual[tsize].person_id = p.person_id
```

```
t_record->qual[tsize].name = p.name_full_formatted
endif
detail
ali = ali+1
if (index > 0)
if(mod(ali, 10) = 1)
stat = alterlist(t_record->qual[index]->alias, ali+9)
endif
t_record->qual[index].alias[ali].person_alias_id =
pa.person_alias_id
t_record->qual[index].alias[ali].pa_alias = pa.alias
else
if(mod(ali, 10) = 1)
stat = alterlist(t_record->qual[tsize]->alias, ali+9)
endif
t_record->qual[tsize].alias[ali].person_alias_id =
pa.person_alias_id
t_record->qual[tsize].alias[ali].pa_alias = pa.alias
endif
foot p.person_id
if(index >0)
stat = alterlist(t_record->qual[index]->alias, ali)
else
stat = alterlist(t_record->qual[tsize]->alias, ali)
endif
foot report
stat = alterlist( t_record->qual, tsize)
with nocounter
```

The CCLCOST metrics for executing the above program using two selects were:**74958:053597 PERSON_ALIAS_ADD_2SEL Cost 5.19 Cpu 3.40 Ela 4.55 Bio 8 Dio 312 O0M0R1 P2R12758 S2,2**

The CCLCOST metrics for getting the same information using the Orjoin were:**74958:053567 PERSON_ALIAS_ADD_ORJOIN Cost96.50 Cpu 3.87 Ela21.42 Bio 8 Dio22950 O0M1R0 P2R14936 S1,5705**

A review of the cost metrics shows that using the two selects is much more efficient than using the Orjoin.

Values are placed on the right-hand side of the operator in qualification clauses.

Values should be placed on the right-hand side of operators in the WHERE clause. For example, the qualification clause should be: **where p.name_last_key="SMITH" not: where "SMITH"=p.name_last_key**

Placing the value on the right-hand side will allow a bind variable to be used when the query is passed to the RDBMS. Passing bind variables to the RDBMS is more efficient because if the query is in the cache, and the values assigned to the bind variable change, the query will not need to be re-optimized. When a query is executed, *Discern Explorer* passes the query to the RDBMS. When the RDBMS receives the query it checks to see if the optimized plan for executing the query is already in the cache. If the optimized plan for executing the query is in the cache, the RDBMS will execute the query using the cached plan. If the optimized plan is not in the cache, the RDBMS will need to optimize the query, cache the plan, and execute the query using the plan. The RDBMS looks at the entire query when it determines if the optimized plan for executing the query is already in the cache. If any part of the query that is sent to the RDBMS is different from any of the queries that are already cached, the RDBMS will optimize the query and cache the plan. Using bind variables allows the same query to be passed to the RDBMS when only the value that is used in the qualification changes.

The following example shows a query written in *Discern Explorer*, the query that is passed to the RDBMS, and the values assigned to the bind variables when the value is placed on the right-hand side of the operator. Because the two queries that are passed to the RDBMS are exactly the same, the second query, qualifying on WILLIAMS will be executed using the plan that was optimized and cached when the first query qualifying on SMITH was executed.

```
select p.name_last_key
from person p
where p.name_last_key = "SMITH"
with nocounter
RDB
SELECT /*+ CCL<GOOD_BIND_TEST:R000:Q01> */ P.NAME_LAST_KEY FROM
PERSON P WHERE (P.NAME_LAST_KEY = :1 ) GO
BIND(Q1:1) CHAR(SMITH)
select p.name_last_key
from person p
where p.name_last_key = "WILLIAMS"
with nocounter
RDB
SELECT /*+ CCL<GOOD_BIND_TEST:R000:Q01> */ P.NAME_LAST_KEY FROM
PERSON P WHERE (P.NAME_LAST_KEY = :1 ) GO
BIND(Q1:1) CHAR(WILLIAMS)
```

The following example shows a query written in *Discern Explorer* and the query that is passed to the RDBMS. Because the values used in the qualification are placed on the left-hand side of the operator, bind variables cannot be used. This results in two different queries getting passed to the RDBMS. Both of these queries will need to be optimized.

```
select p.name_last_key
from person p
where "SMITH" = p.name_last_key
with nocounter
RDB
SELECT /*+ CCL<BAD_BIND_TEST:R000:Q01> */ P.NAME_LAST_KEY FROM
PERSON P WHERE ('SMITH' = P.NAME_LAST_KEY ) GO
select p.name_last_key
```

```
from person p
where "WILLIAMS" = p.name_last_key
with nocounter
RDB
SELECT /*+ CCL<BAD_BIND_TEST:R000:Q01> */ P.NAME_LAST_KEY FROM
PERSON P WHERE ('WILLIAMS' = P.NAME_LAST_ KEY) G
```

Create lists in record structures using the asterisk [*] and use the alterlist() function to initialize memory instead of using a fixed [number] value and the alter() function.

Defining the record structure list using an asterisk and the alterlist function to initialize memory for the list creates a linked list in memory. A linked list can use small pieces of memory requiring less memory management than a contiguous list. Using a fixed number when defining a record structure list requires that the entire list be stored in one contiguous section of memory. Changing the size of the contiguous list using the alter function requires more memory management than adding a new segment to the linked list using the alterlist function. The following example uses an asterisk to define a linked list called plist:

```
Record Person_rec
(
1 plist[*]
2 person_id = f8
2 name = vc
)
```

When a list is defined using the asterisk, the alterlist function is used to initialize memory to store the list. The standard is to initialize memory in reasonable sized blocks. If you know the approximate size of the list, initialize the list to that size. Before attempting to store items in the list, the program should check to see if the list is large enough to hold the new item. If the list is not large enough, it can be sized appropriately. Memory that is initialized for a list and is not used should be freed by using the alterlist function. The following example initializes memory for the person_rec->plist list in blocks of 50 in the detail section of the select. It then frees the unused memory using the alterlist function in the foot report section of the same select.

```
select into "NL:"
p.person_id,
p.name_full_formatted
from person p
head report
cnt = 0
detail
cnt = cnt +1
```

```
;initialize memory for the list
;size function method
if(cnt > size(person_rec->plist, 5))
stat = alterlist(person_rec->plist, cnt +49)
endif
person_rec->plist[cnt].person_id = p.person_id
person_rec->plist[cnt].name = p.name_full_formatted
foot report
;free unused memory
stat = alterlist(person_rec->plist, cnt)
with nocounter
```

The above example uses one common method of initializing memory for lists in blocks. Two other methods that are commonly used to initialize memory for lists in blocks are shown below.

```
;mod function method
head report
cnt = 0
detail
cnt = cnt +1
;initialize memory for the list
if(mod(cnt,50) = 1)
stat = alterlist(person_rec->plist, cnt +49)
endif
person_rec->plist[cnt].person_id = p.person_id
person_rec->plist[cnt].name = p.name_full_formatted
;two variable method
head report
cnt = 0
avail = 0
detail
cnt = cnt +1
;initialize memory for the list
if(cnt > avail)
avail = cnt + 49
stat = alterlist(person_rec->plist, avail)
endif
person_rec->plist[cnt].person_id = p.person_id
person_rec->plist[cnt].name = p.name_full_formatted
```

Generally, lists should not be resized in blocks of one. The following example increases the size of the person_rec->plist in blocks of one. This should not be done.

```
detail
cnt = cnt +1
stat = alterlist(person_rec->plist, cnt)
```

Use the VC data type when creating character items in a record structure if the average length of the character data is greater than or equal to 40.

Declare string items in a record structure as a VC data type if the average size of the string that will be stored in the item is greater than 40 characters. If the average size of the string is less than 40 characters, or is a fixed number of characters, the item should be declared as a fixed length character data type.

When the VC data type is used, the dynamic memory area will be resized based on the size of the string that is being stored, plus 8 bytes of pointer and location information. A fixed length character data type will not be resized and has the possibility of using more memory than is necessary.

For example, if a record item called rec->test is declared as a VC, and a string of 80 characters is stored in rec->test, the total size of rec->test will be 88 bytes (the 80 character string plus 4 byte pointer plus 4 bytes of location information). If rec->test had been declared as a C100, the total size of rec->test would have been 100 bytes, even though the string is only 80 characters.

Therefore, in most cases it is more efficient to initialize items in a record structure to a fixed size if the average size of the string that will be stored in the item is less than 40 characters. If the average size of the string that will be stored in the record item is greater than 40 characters it is more efficient to declare the record structure item as VC data type.

The binary size of the program object is less than the cache size of the server that will execute the program object.

When a *Discern Explorer* program (script) is called from a front-end application, a server is used to execute the program. Servers have the ability to cache programs. It is more efficient to execute the program from the cache than it is to load and execute the program from the *Discern Explorer* library. However, there is a limit on the binary size of the programs that the server can cache. There is also a limit on the number of programs that the server can cache. If your program is going to be executed only once a day, you should not be concerned if it is too large to be cached. However, if your program is going to be executed several times a minute, it must be cached.

The utilities, CCLPROT and CCLQUERY can be used to determine the binary size of your program. In CCLQUERY, refer to the number under BinaryCnt to see the binary size of the program. In CCLPROT, refer to the number under Size to see the binary size of the program.

During the startup process, the server will write a CCL CACHE line to the log file. The CCL CACHE line displays the maximum binary size of a program that can be cached. The CCLSRV utility can be used to read the log file for a server and determine the maximum size of a program that can be cached. A CCL CACHE line similar to the one shown below should be seen near the top of the log file.

CCL CACHE Rng(Max:200 Hit:0 Miss:0 Full:0) Prog(Max:100 Size:75 Hit:0 Miss:4 Full:0 Over:0)

The value to the right of the keyword, Size, is the maximum binary size of a program that can be cached by the server. In the above example, the server can cache any program that is less than 75 blocks. If the CCL CACHE line appears more than once in the log file, the last line that appears will define the server cache properties.

Note: Refer to DiscernExplorerHelp.exe for more information on executing the CCLPROT, CCLQUERY, and CCLSRV utility programs.

Set Trace RecPersist is not used. If needed With PersistScript or With Persist is used to create objects in subprograms that are made known to the calling program.

By default, record structures and variables that are created in a parent program or subroutine are recognized by any child program or subroutine called by the parent. In addition, the calling program does not know Record structures and variables that are created in a child program or subroutine when control returns to the parent. By default, the memory used to store record structures and variables is freed when the child program or subroutine that creates them returns control to the parent.

Occasionally a situation arises where a record structure or variable that is created in a child program or subroutine needs to be accessed by a parent program or subroutine when control returns to the parent. The preferred method for making an object created in the child program or subroutine available when control returns to the parent is to use the With PersistScript control option. Using the With PersistScript control option when creating variables or record structures will cause the object to be recognized by the parent after the child object has executed. When the parent program or subroutine completes, the memory used for the record structure or variable is freed.

In some very rare and specific instances, a record structure or variable needs to remain in memory after the programs that initially created and used it have completed. In those cases the record structure or variable can be created using the With Persist control option. Prior to using With Persist you should investigate other options that might be more efficient. For example the context record could be used instead of a record created using the With Persist or the server could create the record at startup to make it available to all scripts that use the server. By default the CPM Script server has the SkipPersist

trace option turned on. The SkipPersist trace option will treat the With Persist declaration of a variable or record as if the With PersistScript was used.

Set Trace RecPersist should not be used. Record structures created after Set Trace RecPersist but before Set Trace NoRecPersist, will remain in memory until the server that executed the program that created the record is cycled. Improper use of Set Trace RecPersist results in unnecessary persistent records getting created and held in memory. These unnecessary persistent records result in degraded system performance.

Document Revision History

Revision Number:	Effective Date:	Revised by:	Description:
001	August 30, 2004	Bob Ross, Mike Schweder	Initial Release
002	May 16, 2005	Bob Ross, Julie Johnson & Mike Schweder	Updated existing guidelines and added new items to the checklist.
003	May 28, 2008	Bob Ross	Updated checklist and corresponding CCL.