To: MTB Distribution

From: N.S.Davids and Mike Kubicar

Date: August 6, 1982

Subject: MRDS and DMS: Conversion Overview

Comments may be made:

Via electronic Mail:
Davids.Multics
Kubicar.Multics

Via forum (method of choice):
>udd>Demo>dbmt>con>mrdsdev

CONTENTS

INTRODUCTION

This MTB discusses in detail, areas of the MRDS/DMS conversion which were not discussed in MTBs 587, 588, and 589 or areas which were introduced in those MTBs but not completely dealt with. This MTB in combination with MTBs 587 and 588 completely describe the conversion of MRDS from vfile_ (and a few other system routines) to the relation_manager_.

Each section of this MTB describes changes to a particular module or set of modules. If the number of modules is small the section will be titled with their names, if the number is large the section will be titled with the topic that is forcing the change, i.e. "cursors" or "changes to the tuple structure".

WHICH TYPE OF DATABASE - VFILE or PAGE_FILE_

The db_model structure has an element called db_type. This element is referenced only in the mrds_rst_create_db module where it is set to 1. The value 1 will indicate a vfile_ data base while the value 2 will indicate a page_file_ data base.

CMDB EXTENSIONS

MRDS will not be changed to use some of the more esoteric features of the relation_manager_, i.e. multi-attribute secondary indices. Given this the only change needed to the cmdb user interface are the new control arguments "-page_file" and "-vfile".

The code dealing with database creation will have to be changed as described in mtb 588. In addition mrds_rst_create_db will have to set the correct value of db_type in the db_model and the relation collection and index collection ids will have to be stored in the rel_info and attr_info structures. Currently the rel_id in the rel_info structure is declared as bit (12) aligned. Expanding this to the needed 36 bits will not change the storage pattern of the rest of the elements in the structure. Similarly the index_id in the attr_info structure which is currently declared bit (8) aligned needs to be expanded to 36 bits. Note that because of the implementation of the relation manager it will be necessary to call the relation_manager_$create_index with the attr_info.index_id variable in a call by reference mode so that the id is immediately recorded in the model, this is needed in case the index creation process is interrupted and the index needs to be deleted (via the delete_index request in rmdb).

SWITCHING BETWEEN THE VFILE_ AND PAGE_FILE_ RELATION MANAGERS

Within the mrds per database opening data structures (called the resultant) will be a structure of entry variables. This structure will be initialized to either the page_file_ entries or the vfile_ entries when the database is opened and before the actual file structures containing the data are referenced. References to the relation manager will be made via these entry variables. The actual structure to be extended will be the dbcb structure. It will also be required to extend the rsc structure to include a pointer to the dbcb structure so that mrds_rst_format_file will be able to reference it.

Some of the rmdb modules do not execute in an "open database" environment, i.e. there is no dbcb structure to reference. The rmdb subsystem will have to determine the database type and set up its own structure for these modules to use.

CONVERTING FROM A VFILE_ TO A PAGE_FILE_ DATABASE

A conversion tool called convert_mdb_to_pf, short name cvmdbp will be created (see appendix F for user documentation). This command will take an unpopulated mrds page_file_ database and load it from a populated mrds vfile_ database. It will require that the data models for both databases be identical and that the vfile_ database be a version 4 database.

It is not reasonable to convert update_mrds_db_version for two reasons. First the function would no longer fit the name - a confusing situation. Second umdbv requires that the calls to mrds version 1 code be hardcoded in order to read version 3 data models. The code to convert from a vfile_ data base to a page_file_ database would have to be independent of the existing code.

DMDM (command and rmdb request) AND CMDB, AN INCOMPATIBLE CHANGE:

The long display form of the dmdm command and the listing produced by cmdb both include the bit length and bit offsets of the attributes within a tuple. In the case of varying strings these numbers have never been correct; they are completely meaningless for page_file_ databases. They will be deleted from the output (see appendix G for examples of the output).

In addition, since the user needs an indication of the type of database he is displaying, an indication of type will be added to the display.

CURSORS:

The maximum number of cursors that can be referenced is based on the maximum number of keys (equivalent to maximum number of attributes) and maximum number of tuple variables.

(max_attrs + 1) * max_tvs + 1
257 * 20 + 1 = 5141

The maximum number of cursors that can be used in any given selection expression is far larger which implies that all 5141 cursors could be required.

max-and-groups * max-and-terms + 1
100 * 100 + 1 = 10001

Two methods of converting MRDS to use DMS:

The first and easiest method is to enlarge the iocb (cursor) pointer array in the MRDS resultant from 20 to 5141. Given that 10% of the array is actually used (514 cursors) during the life of the database opening that would leave 4627 pointers in each relation that are not used. For a maximum size data base of 256 relations this is 2,369,024 (4627 * 256 * 2) words that are allocated but never referenced.

The second approach requires changing all the mrds modules that reference an iocb pointer in that array (appendix A). References would be changed from a simple array reference to a call to a procedure which returns a cursor pointer. This procedure would manage MRDS's use of the cursors so space would be allocated only for those cursors that were actually used (see appendix C for a functional spec). Note that there will be a performance degradation from what we currently have, also an application that needs all the cursors will not experience a savings in allocated space (it will probably use more space). This method does disconnect the space used for cursor management from the maximums of tuple_variables, and-groups, and and-terms making it easier to increase these values and saves significant space for an application that uses only a few cursors.

Recommendation:

Because of the potential for significant space savings in the vast majority of cases I feel that approach two is the best way to deal with cursors. The procedure mu_cursor_manager will be written and calls to it will replace all references to the array rm_rel_info.iocb_ptr and calls to the procedure mu_open_iocb_manager. This procedure will also open a relation and store its opening id in the rm_rel_info structure if the relation needs to be opened.

Calculation of access cost

For each tuple variable in each and-group permute chooses 1 of the following methods of access:

- total primary key: each attribute in the primary key has an "=" condition against it.
- long key head: The first N attributes in the primary key have an "=" condition against them.
- short key head: The first attribute of the primary key has an inequality condition against it.
- indexed attribute: Access will be via some secondary index.
- unordered sequential: Each tuple will be access in the order they are stored in the MSF. Used if a sequential search is needed and no updates may be performed.
- ordered sequential: Each tuple will be stored in primary key order. Used if a sequential search is needed and tuples may be updated.

Each method has its own cost formula based on the operations needed to perform the access method and an estimate of the number of tuples that will be returned:

- total primary key: cost = TOTAL_PRIMARY_KEY_COST
- long key head: cost = ACCESS_COST * #_of_tuples + ACCESS_OVERHEAD
- short key head: cost = ACCESS_COST * #_of_tuples + ACCESS_OVERHEAD
- indexed attribute: cost = ACCESS_COST * #_of_tuples + ACCESS_OVERHEAD
- unordered sequential: cost = US_ACCESS_COST * relation_size + US_ACCESS_OVERHEAD
- ordered sequential: cost = OS_ACCESS_COST * relation_size + OS_ACCESS_OVERHEAD

Currently the ACCESS_COST and ACCESS_OVERHEAD for long key head, short key head and indexed attribute are all the same, it is not expected that this will change. The current split of sequential into ordered and unordered is required because tuples cannot be updated when using the unordered sequential access method, this will not be the case when using the relation manager and we can combine them into a "sequential" access method. The costs and overheads are currently the virtual cpu time (in hundredths of a second) needed to perform the operation. Experimentation will be necessary in order to assign new values. In order to keep permute independent of the knowledge of which relation type (vfile_ or page_file_) it is dealing with these cost constants cannot be hardcoded into the code, instead the structure containing the relation_manager_ entry points will also contain fixed bin variables which will be set to the value of the constants when the structure is initialized.

Calculating number of tuples selected:

Currently all keys are stored in the same key tree so only information about the average selectivity of a combination of all the indices is available. For vfile_ relations this will remain the case but page_file_ relations will contain information about the average selectivity of each index. This information will allow better estimates of the number of tuples that will be retrieved. The modifications needed to permute to do this will not be extensive, it will require that an array of the indexed attributes which are useable be kept and that a loop over all useable indices be implemented to determine the index with the minimum accessing cost. In addition the rm_attr_info structure of the resultant will have to be expanded to include the duplicate key count and the duplicate key count for the entire relation may be removed from rm_rel_info.

Recommendation:

Maintain the current values of the cost constants until experimentation with the relation_manager_ (vfile_ and page_file_) can be done.

Implement a version of permute which utilizes the duplicate key counts for each index.

MRDS_DSL_MODIFY, MU_MODIFY

The checks to be sure that the user has update scope set, that the view in use can be used to modify tuples, and if the database has been secured that the user has modify access on each of the tuples he is trying to modify will be moved from mu_modify to mrds_dsl_modify. This will also be a small performance improvement since it is necessary to make these checks just once, not for every tuple being modified. In addition mrds_dsl_modify will be changed to call mu_cursor_manager_$get inorder to get the relation collection cursor. Finally the code calling mrds_dsl_search and mu_modify will be changed so that relation_manager_$modify_record_by_id is called instead of mu_modify and so that modify_record_by_id is passed an array of 100 tuple_ids. This will also be a performance improvement since less calls will be executed. The module mu_modify may be deleted.

It has been decided not to utilize relation_manager_$modify_record_by_search because of the increased time to convert both mrds_dsl_modify and mrds_dsl_search. Once the conversion has been completed this modification can be made.

MRDS_DSL_DELETE, MU_DELETE

The checks to be sure that the user has update scope set, that the view in use can be used to delete tuples, and if the database is secured that the user has access to delete tuples will be moved from mu_delete to mrds_dsl_delete. This will also be a small performance improvement since it is necessary to make these checks just once, not for every tuple being deleted. In addition mrds_dsl_delete will be changed to call mu_cursor_manager_$get inorder to get the relation collection cursor. Finally the code calling mrds_dsl_search and mu_delete will be changed so that relation_manager_$delete_record_by_id is called instead of mu_delete and so that delete_record_by_id is passed an array of 100 tuple_ids. This will also be a performance improvement since less calls will be executed. The module mu_delete may be deleted.

It has been decided not to utilize relation_manager_$delete_record_by_search because of the increased time to convert both mrds_dsl_delete and mrds_dsl_search. Once the conversion has been completed this modification can be made.

DISPLAY_MRDS_DB_POPULATION

The output for this command when the -long control argument is used will be incompatibly changed. The output of the vfile_ version, total number of bytes in the vfile records, number of vfile keys and their total bytes, number of duplicate keys and their bytes, tree height, number of pages, amount of free space and number of updates will be deleted. They will be replaced with a list of the indexed attributes and the number of tuples that each index can on the average be expected to select. The formula for calculating the number of tuples selected will be:

```
                        (T/(T-D),   if D ^= T
     tuples selected = (
                        (T,         if D = T
```

where:

T is the number of tuples in the relation

D is the number of duplicated key values for each index, ala vfile_status_ dup_keys.

For vfile_ relations the number of tuples selected will be the same for all the indices since the value D is not known for each individual index. The list will not be displayed for version 3 databases since D is not known.

In addition the message "Opening version <number> data model: <path>" will be changed to "Displaying version <number> data model: <path>". The reason for the change is that there is no need to tell the user that the data model is being opened and since there is no message that the data model has been closed the user can be confused and think that some other command to close the data model is required. See appendix E for example outputs.

The procedure will use relation_manager_$get_count and get_duplicate_key_count. The performance of the get_count entry will not be a problem and it will return the exact number of tuples in the relation at the time of the call.

MU_GET_REL_SIZE

This module will use the get_count entry in the relation_manager_.

SCOPE

The module mrds_dsl_set_fscope will need to be modified to call the relation_manager_$set_scope entry after the scopes have been added to the dbc. A pointer to the relation's rm_rel_info structure is known so that the relation's page_file_opening id is readily available.

The module mrds_dsl_delete_fscope will have to be modified to call either the relation_manager_$set_scope or delete_scope. The set_scope module will be called if only part of the relation's scope is being deleted, delete_scope will be called if all the scope for the relation is being deleted.

CHANGES TO THE TUPLE STRUCTURE

Part of the modifications needed for mrds to effectively use the new Data Management System are changes to the data structures used by mrds. The major change will be to the tuple structure. Currently, mrds calls iox_ directly to get and put records to the relation data files. Each of these records is a complete tuple in a format which is managed by mrds. Before writing a tuple, mrds must construct it from the data given to it by a user program. Likewise, when is needed, mrds must extract it from the tuple.

This is not the situation with the new Data Management System. Mrds no longer manages the data in a tuple. This function will be handled by the relation manager. When mrds needs to read or write a tuple, the data items contained in the tuple are described by a vector structure. The idea and purpose of vectors is described in the draft mtb "The Vector Concept". The specific vector structure "simple typed vector", used by mrds is described in draft mtb-545, "DM: Relation Manager Functional Spec".

Although there in only one type of vector described in mtb-draft, "The Vector Concept", in reality there are two. The first is the general type vector. In addition to describing where the data is located, the general type vector describes which of the dimensions of the vector the data item belongs to. This allows the possibility of omitting fields in the vector during calls to relation_manager_. In earlier design, it was decided that this feature was overly complex. Because of this, the simple vector type was created for use by mrds. A simple vector is basically an array of pointers to the data. Using this type of vector is simpler and cheaper. Also, few if any calls would have to be made to the vector_util_ subroutines to manipulate the vectors. There will be the restriction, though, that incompletely specified vectors can not be used. This is not a problem since mrds currently handles only complete tuples at the low levels. Null attribute values are not permitted. The only place that specifying incomplete tuples might be desirable is during a modify operation. This is not necessary, though, since mrds will always read a tuple before modifying it. Thus, it can copy the fields that don't change into the new tuple.

The current tuple structure used by mrds is:

```
dcl 1 tuple aligned based (t_ptr),
    2 rel_id bit (12) unal,
    2 attr_exists (tuple_num_atts) bit (1) unal,
    2 var_offsets (tuple_nvar_atts) fixed bin (35) unal,
    2 force_even_word (tuple_pad_length)
                           fixed bin (71) aligned,
    2 data char (tuple_max_dlen) unal;
```

where:

rel_id
: is the relation id in the file. Currently it is always one.

attr_exists
: is true if the corresponding attribute in the tuple has other than a null value. Currently, all mrds attributes must have non-null values.

var_offsets
: is the bit offset, into the data area, of the start of a varying attribute.

force_even_word
: is for padding. Currently it is not used.

data
: is the data area where attribute values go.

The new vector structure is:

```
dcl 1 simple_typed_vector  based (simple_typed_vector_ptr),
    2 type                 fixed bin (17) unal,
    2 number_of_dimensions fixed bin (17) unal,
    2 dimension            (tv_number_of_dimensions refer
                            (typed_vector
                             .number_of_dimensions)),
      3 value_ptr          pointer unal;
```

where:

type
: indicates the type of the vector structure. 1 indicates a general_typed_vector structure and 2 indicates a simple_typed_vector structure.

number_of_dimensions
: is the number of dimensions present in the vector.

dimension.value_ptr
is a pointer to the value of the dimension.

The changes needed to mrds to replace tuple structures with vectors are of two general kinds. First of all, code which reads or writes the internal structure of a tuple must be changed to operate on simple vectors. The second is that code which manages tuple storage space must be changed to work correctly with vectors.

There are several differences between tuples and vectors that are relevant to mrds. The first, and most obvious, is that a vector is accessed differently than a tuple. The tuple structure is a template for a record which will actually be stored in the database via vfile_ operations. The structure contains both the data that is to be stored and control information that specifies how to access that data. Information concerning the maximum length, data type, and start of the attribute (for fixed length attributes) is contained in the model definition of the database. During database open, this information is copied to the resultant for ease of access. The attributes values of the tuple are stored in tuple.data. Their order is not the definition order; all of the fixed length tuples are stored first, followed by all the varying length ones. Storage order in each of these two sections is definition order. Only the portion of the varying length data that is actually defined is stored in the tuple to conserve space. The array tuple.exists is a bit array which tells whether a particular field in the tuple is valid or not. The bits correspond one to one with the attributes in definition order. Currently, mrds does not support the notion of a null attribute. All attributes in the tuple must be defined. Thus, all the bits will be set. The reason they exist in the database is historical. The field, tuple.var_offsets, describes where in the tuple a varying attribute begins. The value describes the bit offset of the start of the data from the beginning of tuple.data.

The tuple structure will be eliminated from mrds. It will be replaced with the vector structure which will be used in all data store/retrieve calls to relation_manager_. The vector structure, unlike the tuple structure, does not include a section to hold the actual data. It is basically an array of pointers, where each pointer locates the value of the attribute in the corresponding position of the relation. Mrds no longer has to manage the contents of the records in the storage files. Relation_manager_ now takes over this job. To mrds, a complete tuple will now appear as an array of pointers to attribute data.

There are three types of changes that need to be made to mrds in order to use the vector instead of the tuple structure.

The first is to change the manner in which mrds retrieves data from tuples. Since it now manages the contents of a tuple, mrds has to calculate where in the record the attribute data is and then copy it out. With the vector structure, mrds will directly have a pointer to the data. Currently, when mrds needs to obtain an attribute's value, it starts with a pointer to the rm_attr_info structure in the resultant which describes the attribute of interest. There is one of these structure for each attribute in each relation of a database. The structure contains, among other things, whether or not the attribute is varying, the definition order of the attribute, the position in the tuple, and the length. Mrds uses this, and a pointer to the tuple itself, to extract attribute data. The bit offset in rm_attr_info specifies where the attribute begins and can be either positive or negative. If positive, the attribute it describes is a fixed size one. The number is the bit offset, from the start of tuple.data, of the beginning of the attribute's value. If negative, the number describes a varying attribute. The absolute value of the offset is an index into tuple.var_offsets. This value is a bit offset of the start of the varying data object. Using the appropriate offset, mrds builds a pointer to the start of the attribute data value. The maximum size of the data object is obtained from a descriptor in the rm_domain_info structure.

The modification needed to use the vector structure is relatively straightforward. Mrds will use the field rm_attr_info.defn_order to find the attribute's definition order in the relation. It will use this value as an index into simple_typed_vector.dimension.value_ptr. This will give mrds the data pointer it needs. The maximum data size can still be retrieved from rm_domain_info's descriptor.

The next change that is necessary is in the way mrds builds a tuple for storage into the database. The routine which does this is mu_build_tuple. The code in it performs three functions. The first is, of course, inserting values in a tuple from a move list. Mu_build_tuple also does encoding of data with user supplied encode procedures, and checking of the data after it's been encoded. Any data conversions that are needed are also done by this procedure.

The code associated with inserting values in tuples will have to be rewritten to use the vector structure. This should not be a time consuming task as building a vector is a simpler operation than building a tuple. Mu_build_tuple will construct the vector structure by simply copying the pointer from the move list into the value_ptr of the simple_typed_vector. If a conversion or encode procedure call is necessary, the final value will be created in temporary storage and a pointer to it placed in the vector structure. One other routine which builds tuples is mu_get_tuple. Since it is doing so for a temporary relation or an rmdb create_relation function, no conversion or encoding will be required.

The other change needed to convert to the vector structure is the manner in which space is allocated for tuples. Currently, mrds will allocate tuple space on each store if it is storing a tuple to a different relation than on the last store. If the relation is the same one, the previous space can be reused.

The internal structure of a tuple varies from relation to relation. Thus, space needs to be allocated and released for each different relation stored. For vectors, there is, no reason to allocate and deallocate vector structures with each store. A single structure can be allocated when the database is opened and used through the life of the database. The same pointer in the dbcb that points to the tuple space used in the last store (dbcb.sti_ptr) can be used to point to this static vector structure. Allocating the biggest possible vector will not take a prohibitively large amount of storage space. Also, space must be allocated for data items that will be placed in the tuple if their values must be converted or encoded. This space can be allocated in the area which mu_build_tuple is passed via pointer. The area is emptied on each call to dsl_$store so there is no need to ever free the data items.

The changes to structures allocated for a retrieve will be slightly more extensive. Currently, mrds allocates all the tuple space before any retrieves actually happen. Since several tuples may actually be needed to do the comparisons specified in a selection expression, space for as many tuples as are needed to satisfy the where clause are allocated and pointed to by the structure, tuple_info. Tuple_info is pointed to by dbcb.ti_ptr. When the search list is built in mrds_dsl_gen_srch_prog, pointers to where the tuple actually will be placed are copied from the tuple_info structure into the search_list.

Relation_manager_ will allocate any space it needs when retrieving tuples from the database; mrds must not reserve space for the tuples. Therefore, when the search list is built, a pointer to the actual tuple location can not be obtained. The search list will have to be modified so that, instead of a pointer to a tuple, an index into tuple_info.tuple is kept. Then, when the search program needs to access an attribute of a particular tuple, it will use this index to get the correct pointer from tuple_info. Of course, when the tuple is retrieved from the database, it must be stored into the correct position in tuple_info.tuple immediately. Relation_manager_ should be given the area pointed to by dbcb.retrieve_area_ptr to allocate the space it needs. It is passed down to the mu_retrieve routine via a pointer. This area is emptied on each call to dsl_$retrieve. Allocations in it do not have to be freed.

OTHER CHANGES TO DATA STRUCTURES

There are two other minor changes that will have to be made related to data structures. The first one is to the tuple_id_unbl data structure. This structure will no longer be used by mrds. It is now used in the conversion of vfile_ descriptors to mrds tuple ids. This conversion is, even today, not necessary and is present only for historical reasons. Using relation manager, mrds must not alter the tuple ids it is given so this code must be removed.

Also, since mrds will no longer manage indices in the data files, the key_list structure that is used to identify these indices must be deleted.

USER INTERFACE MODULES THAT NEED TO BE MODIFIED TO HANDLE TRANSACTIONS

See mtb 587 (MRDS and DMS) for the discussion on what changes to make.

Commands

display_mrds_db_population
unpopulate_mrds_db
update_mrds_db_version
convert_mdb_to_pf (proposed in this mtb)

Subroutines

dsl_$define_temp_rel
dsl_$delete
dsl_$get_population
dsl_$modify
dsl_$retrieve
dsl_$store

RMDB Subsystem

create_index
delete_index

APPENDIX A - Modules that will not be deleted and which reference the array rm_rel_info.iocb_ptr

```
mrds_dsl_define_temp_rel.pl1
mrds_dsl_finish_file.pl1
mrds_dsl_gen_srch_prog.pl1
mrds_dsl_optimize.pl1
mrds_dsl_search.pl1
mu_delete.pl1
mu_get_rel_size.pl1
mu_get_tid.pl1
mu_sec_get_tuple.pl1
mu_sec_make_res.pl1
mu_store.pl1
```

APPENDIX B - Changes to include files

The following set of include files have fields which no longer have meaning when using the relation manager. They will have to be changed as will any modules using these fields. The following include files must be modified:

mdbm_comp_val_list:

This structure contains fields which are bit offsets into the tuple. These are the fields comp_val_list.db_offset and comp_val_list.db_offset2. These fields must be changed to be the position of the attribute in the tuple.

Modules which reference these fields in mdbm_comp_val_list:

mrds_dsl_gen_srch_prog
mu_retrieve

mdbm_key_list:

The structure, key_list, is used by mrds to manage indices in the relation data files. Since mrds will no longer manage indices when using relation manager, the structure should be deleted.

Modules which reference key_list:

mu_store

mdbm_rm_attr:

The field, rm_attr_info.bit_offset is a bit offset into the tuple if positive, or an index into the tuple.var_offset if negative. Rm_attr_info.bit_offset can be deleted since bit offsets into tuples are no longer meaningful.

Modules which reference this field in mdbm_rm_attr:

```
mrds_dsl_define_temp_rel
mrds_dsl_eval_expr
mrds_dsl_eval_func
mrds_dsl_gen_srch_prog
mrds_dsl_get_rslt_info
mrds_dsl_retrieve
mu_build_tuple
mu_get_data
mu_get_tuple
mu_sec_get_tuple
mu_sec_make_res
mu_store
```

mdbm_rm_rel_info:

rm_rel_info.max_data_len is the maximum length, in characters, of the data portion of the tuple. This number is no longer meaningful since relation_manager_ handles the tuple structure. It can be deleted.

Modules which reference this field in mdbm_rm_rel_info:

```
mrds_dsl_define_temp_rel
mrds_dsl_eval_expr
mrds_dsl_eval_func
mrds_dsl_gen_srch_prog
mrds_dsl_retrieve
mrds_dsl_select_clause
mrds_dsl_store
mu_build_tuple
mu_get_data
mu_get_tuple
mu_retrieve
mu_sec_get_tuple
mu_sec_make_res
mu_store
rmdb_create_and_pop_rel
```

mdbm_tuple_id:

This set of structures has been rendered obsolete. Mrds no longer interprets the internal structure of tuple ids. It considers them to be a one word identifier.

Modules which reference this include file:

mu_get_tid
mu_sec_get_tuple
rmdb_create_index

mrds_rel_desc:

The field rel_desc.attributes.bit_offset is the bit offset of the attribute within the tuple. This is no longer meaningful.

Modules which reference this field in mrds_rel_desc:

mrds_dm_get_attributes

APPENDIX C - mu_cursor_manager functional specification

entry: mu_cursor_manager$get

Returns the indicated cursor_ptr, creating it if necessary. If the relation is not yet opened it will be opened and its opening id stored in the rm_rel_info structure. If storage_ptr is null storage will be allocated.

Usage:

```
declare mu_cursor_manager$get entry (ptr, fixed bin, fixed
     bin, ptr, ptr, fixed bin (35));

call mu_cursor_manager$get (rmri_ptr, tuple_variable_index,
     collection_index, storage_ptr, cursor_ptr, code);
```

where:

rmri_ptr
: pointer to the relation's rm_rel_info structure.

tuple_variable_index
: is the index of the tuple variable within the selection expression

collection_index
: is the index of the collection, the tuples themselves have an index of -1, the primary key has an index of 0, and each of the secondary keys is number 1 through N.

storage_ptr
: is a pointer to the storage where the cursor ptr and rel_name-tuple_variable-collection_id relationship for a given database index is kept. If the pointer is null storage space will be created. The call that creates the first cursor should have a null storage_ptr.

cursor_ptr
: is a pointer to the cursor associated with the rel_name-tuple_variable-collection_id.

code

is a standard error code.

entry: mu_cursor_manager$delete_all

Deletes all the cursors in the storage area and closes all the relations with cursors in the storage area.

Usage:

```
declare mu_cursor_manager$delete_all entry (ptr, fixed bin
    (35));

call mu_cursor_manager$delete (storage_ptr, code);
```

where:

storage_ptr
: is a pointer to the storage where the cursor ptr and rel_name-tuple_variable-collection_id relationship for a given database index is kept.

code
: is the standard error code.

APPENDIX D - mu_cursor_manager cursor access and storage mechanism design notes

The number of cursors that can be associated with an open database can range from 1 to over 2 million. The access mechanism must be based on the number of cursors in order to preclude storage or access inefficiencies. It has been decided to use two mechanisms, the first will be based on an array overlaid on a segment, the second on a keyed vfile. Both the segment and the vfile will be created in the current mrds temp directory.

The search key for both mechanisms will be a 144 bit string made up of the rmri_ptr, tuple_variable_index, and collection_id. The "record" associated with the key will be the pointer to the cursor.

The first mechanism will be used when the number of cursors is less than "N". The value of "N" must be determined experimentally but is expected to be less than 10,000. The array will start with 0 elements and be built up 1 element at a time using an insertion sort mechanism. An ALM program for efficiently moving blocks of characters (bits) will be written so that the expense of shifting the array to do an insert will be minimal.

In the second mechanism the keyed sequential vfile must be built and loaded from the array. The input output parameter storage_ptr will be changed to point to an iocb instead of the base of a segment.

Note that current plans call for cursors to be deleted only when the database is closed.

The cursors themselves will be stored in an extensible area in a temp segment in the process directory. The process directory is used so that segments to extend the area are all located in the same directory.

APPENDIX E - example output from display_mrds_db_population

Each example has 2 parts. The first part is the output as it currently looks, the second part (indented 3 spaces) is how the output will look after the change.

```
! display_mrds_db_population db1

Opening version 4 data model: >udd>m>databases>db1

RELATION                                TUPLES

personnel                               100
parts                                   500

   ! display_mrds_db_population db1

   Displaying version 4 data model:  >udd>m>databases>db1.db

   RELATION                                TUPLES

   personnel                               100
   parts                                   500
```

```
! dmdbp db1 -long

Opening version 4 data model: >udd>m>databases>db1.db

Vfile version:  40/41

Relation:  personnel
  Tuples:  100
   Bytes:  557

          Vfile keys:  300      bytes:  691
            dup keys:  98       bytes:  166
         tree height:  2        pages:  10
          free space:  1      updates:  309

Relation:  parts
  Tuples:  500
   Bytes:  117

          Vfile keys:  1000     bytes:  157
            dup keys:  0        bytes:  470
         tree height:  2        pages:  50
          free space:  1      updates:  309
```

```
! dmdbp db1 -long

Displaying version 4 data model:  >udd>m>databases>db1.db

RELATION      TUPLES      INDEX       AVE TUPLES SELECTED

personnel      100
                          ssn                  1
                          sex                 50
parts          500
                          part_no              1
```

```
! dmdbp old_db1

Opening version 3 data model: >udd>m>databases>old_db1

RELATION                                   TUPLES

personnel                                  100
parts                                      500
```

```
   ! dmdbp old_db1

   Displaying version 3 data model: >udd>m>databases>old_db1

   RELATION                                   TUPLES

   personnel                                  100
   parts                                      500
```

```
! dmdbp old_db1 -long

Opening version 3 data model: >udd>m>databases>old_db1

Vfile version:  40/41

Relation:  personnel
  Tuples:  100
   Bytes:  557

          Vfile keys:  300      bytes:  691
         tree height:  2        pages:  10
          free space:  1      updates:  309

Relation:  parts
  Tuples:  500
   Bytes:  117

          Vfile keys:  1000     bytes:  157
         tree height:  2        pages:  50
          free space:  1      updates:  309
```

```
! dmdbp old_db1 -long

Displaying version 3 data model: >udd>m>databases>old_db1

RELATION                                TUPLES

personnel                               100
parts                                   500
```

APPENDIX F - User documentation for convert_mdb_to_pf

convert_mdb_to_pf (cvmdbp)	convert_mdb_to_pf (cvmdbp)

SYNTAX AS A COMMAND:

```
   convert_mdb_to_pf vfile_db_path page_file_db_path
```

FUNCTION: Loads a newly created mrds page_file_ database from a populated mrds vfile_ database.

ARGUMENTS:

vfile_db_path
: is the path (relative or absolute) to the populated vfile_ database. The ".db" suffix is not required.

page_file_db_path
: is the path (relative or absolute) to an unpopulated page_file_ database. The ".db" suffix is not required.

NOTES:

The data model of the two databases must be the same.

APPENDIX G - dmdm -long and cmdb list examples

Each example has 2 parts. The first part is the output as it currently looks, the second part (indented 3 spaces) is how the output will look after the change.

```
! dmdm db1 -long

DATA MODEL FOR DATA BASE  >udd>m>database>db1.db

Version:                  4
Created by:              FOOBAR.Multics.a
Created on:              07/28/82  1544.7 mst Wed

Total Domains:            4
Total Attributes:         7
Total Relations:          2


RELATION NAME:    parts
Number attributes:           3
Key length (bits):         288
Data Length (bits):        612

   ATTRIBUTES:

      Name:        part_name
      Type:        Key
      Offset:      1 (bits)
      Length:      288 (bits)
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned


      Name:        order_name
      Type:        Data
      Offset:      289 (bits)
      Length:      288 (bits)
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned
```

```
      Name:         part_no
      Type:         Data  Index
      Offset:       577 (bits)
      Length:       36 (bits)
      Domain_info:
            name:   type
            dcl:    real fixed binary (17,0) aligned

RELATION NAME:      personnel
Number attributes:            4
Key length (bits):          288
Data Length (bits):         666

   ATTRIBUTES:

      Name:         last_name
      Type:         Key
      Offset:       1 (bits)
      Length:       288 (bits)
      Domain_info:
            name:   name
            dcl:    character (32) nonvarying unaligned


      Name:         first_name
      Type:         Data
      Offset:       289 (bits)
      Length:       288 (bits)
      Domain_info:
            name:   name
            dcl:    character (32) nonvarying unaligned


      Name:         ssn
      Type:         Data  Index
      Offset:       577 (bits)
      Length:       81 (bits)
      Domain_info:
            name:   ssn
            dcl:    character (9) nonvarying unaligned


      Name:         sex
      Type:         Data  Index
      Offset:       658 (bits)
      Length:       9 (bits)
      Domain_info:
            name:   sex
            dcl:    character (1) nonvarying unaligned
```

```
! dmdm db1 -long

DATA MODEL FOR VFILE DATA BASE  >udd>m>database>db1.db

Version:                    4
Created by:              FOOBAR.Multics.a
Created on:              07/28/82  1544.7 mst Wed

Total Domains:              4
Total Attributes:           7
Total Relations:            2


RELATION NAME:     parts
Number attributes:           3

   ATTRIBUTES:

      Name:         part_name
      Type:         Key
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned


      Name:         order_name
      Type:         Data
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned


      Name:         part_no
      Type:         Data  Index
      Domain_info:
            name: type
            dcl:  real fixed binary (17,0) aligned

RELATION NAME:     personnel
Number attributes:           4

   ATTRIBUTES:

      Name:         last_name
      Type:         Key
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned


      Name:         first_name
      Type:         Data
```

```
Domain_info:
      name: name
      dcl:  character (32) nonvarying unaligned


Name:       ssn
Type:       Data  Index
Domain_info:
      name: ssn
      dcl:  character (9) nonvarying unaligned


Name:       sex
Type:       Data  Index
Domain_info:
      name: sex
      dcl:  character (1) nonvarying unaligned
```

```
! cmdb db1 -list
CMDB Version 4 models.
! pr db1.list -nhe

        CREATE_MRDS_DB LISTING FOR >udd>m>databases>db1.cmdb
        Created by:          FOOBAR.Multics.a
        Created on:          07/28/82  1551.3 mst Wed
        Data base path:      >udd>m>databases>db1.db
               Options:      list


        domain:
                  name char (32) nonvarying unaligned,
                  sex  char (1) nonvarying unaligned,
                  ssn char (9) nonvarying unaligned,
                  type fixed bin (17,0) aligned;

        attribute:
                  last_name name,
                  first_name          name,
                  part_name           name,
                  order_name          name,
                  part_no             type;

        relation:
                  personnel (last_name* first_name ssn sex),
                  parts     (part_name* order_name part_no);

        index:
                  personnel (ssn sex),
                  parts     (part_no);

NO ERRORS

DATA MODEL FOR DATA BASE  >udd>m>databases>db1.db

Version:                  4
Created by:            FOOBAR.Multics.a
Created on:            07/28/82  1551.3 mst Wed

Total Domains:            4
Total Attributes:         7
Total Relations:          2


RELATION NAME:    parts
Number attributes:          3
Key length (bits):        288
Data Length (bits):       612

   ATTRIBUTES:
```

```
      Name:         part_name
      Type:         Key
      Offset:       1 (bits)
      Length:       288 (bits)
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned


      Name:         order_name
      Type:         Data
      Offset:       289 (bits)
      Length:       288 (bits)
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned


      Name:         part_no
      Type:         Data  Index
      Offset:       577 (bits)
      Length:       36 (bits)
      Domain_info:
            name: type
            dcl:  real fixed binary (17,0) aligned

RELATION NAME:      personnel
Number attributes:            4
Key length (bits):          288
Data Length (bits):         666

   ATTRIBUTES:

      Name:         last_name
      Type:         Key
      Offset:       1 (bits)
      Length:       288 (bits)
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned


      Name:         first_name
      Type:         Data
      Offset:       289 (bits)
      Length:       288 (bits)
      Domain_info:
            name: name
            dcl:  character (32) nonvarying unaligned
```

```
Name:         ssn
Type:         Data  Index
Offset:       577 (bits)
Length:       81 (bits)
Domain_info:
      name: ssn
      dcl:  character (9) nonvarying unaligned


Name:         sex
Type:         Data  Index
Offset:       658 (bits)
Length:       9 (bits)
Domain_info:
      name: sex
      dcl:  character (1) nonvarying unaligned
```

```
! cmdb db1 -list -page_file_
CMDB Version 4 models.
! pr db1.list -nhe

          CREATE_MRDS_DB LISTING FOR >udd>m>databases>db1.cmdb
          Created by:           FOOBAR.Multics.a
          Created on:           07/28/82  1551.3 mst Wed
          Data base path:       >udd>m>databases>db1.db
                 Options:       list page_file_


          domain:
                    name char (32) nonvarying unaligned,
                    sex  char (1) nonvarying unaligned,
                    ssn char (9) nonvarying unaligned,
                    type fixed bin (17,0) aligned;

          attribute:
                    last_name name,
                    first_name          name,
                    part_name           name,
                    order_name          name,
                    part_no             type;

          relation:
                    personnel (last_name* first_name ssn sex),
                    parts     (part_name* order_name part_no);

          index:
                    personnel (ssn sex),
                    parts     (part_no);

NO ERRORS

DATA MODEL FOR PAGE_FILE_ DATA BASE  >udd>m>databases>db1.db

Version:                  4
Created by:            FOOBAR.Multics.a
Created on:            07/28/82  1551.3 mst Wed

Total Domains:            4
Total Attributes:         7
Total Relations:          2


RELATION NAME:    parts
Number attributes:           3

   ATTRIBUTES:

      Name:       part_name
      Type:       Key
```

```
        Domain_info:
              name: name
              dcl:  character (32) nonvarying unaligned


        Name:         order_name
        Type:         Data
        Domain_info:
              name: name
              dcl:  character (32) nonvarying unaligned


        Name:         part_no
        Type:         Data  Index
        Domain_info:
              name: type
              dcl:  real fixed binary (17,0) aligned

  RELATION NAME:      personnel
  Number attributes:           4

     ATTRIBUTES:

        Name:         last_name
        Type:         Key
        Domain_info:
              name: name
              dcl:  character (32) nonvarying unaligned


        Name:         first_name
        Type:         Data
        Domain_info:
              name: name
              dcl:  character (32) nonvarying unaligned


        Name:         ssn
        Type:         Data  Index
        Domain_info:
              name: ssn
              dcl:  character (9) nonvarying unaligned


        Name:         sex
        Type:         Data  Index
        Domain_info:
              name: sex
              dcl:  character (1) nonvarying unaligned
```