To: MTB Distribution

From: N.S.Davids and Mike Kubicar

Date: July 19, 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

INTRODUCTION

This MTB will discuss the conversion of mrds from using vfile_ and other system routines for a relation's data management to using the relation_manager_ being developed at CISL. The use of the relation_manager_ will increase the functionality of MRDS, providing transaction processing, better concurrency control, and in the future larger data files with better paging.

The MTB is broken up into the sections: "changes to modules manipulating the relation's data and the MSF containing the data", and "unresolved issues". The first section is further divided along the function lines of the modules. The pertinent features of each module are described in greater or lesser detail depending on its complexity as are the changes that are needed. In cases where the correct change is not obvious several possible changes are described, these are summarized in section 2.

Changes to Modules Manipulating the Relation's Data and the MSF Containing the Data:

Modules that display or return statistics about a relation:

display_mrds_db_population
This command calls vfile_status_. This call will have to be replaced with calls to relation_manager_$get_duplicate_key_count and either relation_manager_$get_population or get_count. The entry get_population returns a close (but not exact) count of the number of tuples in the relation. The entry get_count will return an exact count but will be slower than get_population. The statistics concerning the number of bytes in the tuples, number of bytes used in keys, number of bytes used in the duplicated keys, vfile_ tree height, number of pages, amount of free space, and number of updates which are currently displayed by the command will not be available when the command is converted to use the relation_manager_. Of all these statistics only the number of pages and number of updates seems useful to the user, the number of pages can be obtained via other system calls. This command will have to be incompatibly changed, first to remove the statistics that will not be available via the relation_manager_ and second to indicate that the population displayed will *no* longer be exact or to indicate an increase in the execution time of the command. It is also possible to call the relation_manager_ only for page_file relations and continue to call vfile_status_ for databases composed of vfile_ relations.

mu_get_rel_size
This internal (not externally documented) subroutine calls iox_$control with a control order of file_status. it returns to its caller the number of tuples in the relation, the total number of keys, and the number of duplicate keys which are used to estimate the costs of search paths. The call to iox_ can be replaced with a call to the relation_manager_$get_duplicate_key_count and either relation_manager_$get_population or get_count. Which one will depend on the further investigations of the accuracy of get_population and the sensitivity of the cost estimate and the performance of get_count.

Modules that setup I/O with the relation:

mu_open_iocb_manager
This internal subroutine is called by many other modules when they determine that they need a new iocb pointer. These iocbs are stored in the relation's resultant. Multiple iocbs are needed when the same relation is referred to by multiple tuple variables. The maximum number of iocbs needed is the maximum number of tuple variables plus 1. The concept of an iocb is being replaced with the concept of cursor. A cursor is a position marker into the relation. Each cursor is associated with either the relation's tuples or 1 of the relation's keys (primary or secondary). In this respect they are more limited than an iocb which can be used to reference any of the keys or the tuples. The worse case maximum number of cursors needed is (maximum number of attributes + 1) * (maximum tuple variables + 1) which requires more space than is reasonable. There will therefore have to be a change in the current algorithm which allocates iocbs as needed and then keeps them around for future use to one which perhaps keeps a certain number of cursors around in a "cache" of cursors but allocates extra cursors when needed and frees them afterward. It would be simpler to not have the "cache" of cursors but performance may suffer. In addition this module will have to open the page_file containing the relation with a call to the relation_manager_$open if the relation is not yet opened.

mrds_dsl_finish_file
This internal subroutine closes, detaches, and destroys the iocbs associated with a relation. The calls must be changed to the relation_manager_$destroy_cursor and close.

rmdb_create_index
rmdb_delete_index
rmdb_create_relation
These internal subroutines attach, open, close, detach, and destroy iocbs to the relations. They do this independently of mu_open_iocb_manager and mrds_dsl_finish_file because the modules do not work in an "open database" environment. Calls to iox_$attach and open must be replaced with calls to the relation_manager_$open and create_cursor and calls to iox_$close, detach, and destroy must be replaced with calls to the relation_manager_$destroy_cursor and close.

The modules that create or modify a relation:

mrds_rst_format_file
This internal subroutine calls iox_ to attach and open the relation's MSF, in the process the MSF is created. These calls must be replaced with a call to the

relation_manager_$create_relation and N calls to create_index. The other function of this module is to format the MSF, this function is no longer needed and it might be reasonable to move to calls to the relation_manager_ up into the caller of mrds_rst_format_file (create_mrds_db).

rmdb_create_relation
This internal subroutine uses the same logic as mrds_dsl_format_file and must be modified in the same way. Its other functions however cannot be subsumed into its caller (rmdb_rq_create_relation).

rmdb_create_index
This internal subroutine writes (using iox_) a new index key for each tuple in the relation. It does this by calling mu_scan_records to read each tuple. The call to the relation_manager_$create_index will automatically scan the tuples and write the new index so that the calls mu_scan_records and iox_$write_record can be replaced with one call to create_index. While it will be necessary to open the relation via relation_manager_$open it will not be necessary to create any cursors.

rmdb_delete_index
This internal subroutine deletes, from the vfile_ key tree, all keys with a certain key head, which corresponds to the keys for a particular attribute. The calls to iox_$control (order delete_key) must be replaced with a call to the relation_manager_$delete_index. As in rmdb_create_index it will be necessary to call relation_manager_$open to open the relation but it will not be necessary to create any cursors.

The module that deletes the relation:

rmdb_delete_relation
This internal subroutine calls delete_ to delete the relation's MSF, it must be changed to call the relation_manager_$delete_relation.

The modules that store or modify a tuple:

mu_build_indl
This internal subroutine extracts from the tuple those values that will be secondary indices and builds a list of secondary index strings that include the relation and attribute identifiers. It also encodes the value so that collating sequence of the string and the collating sequence of the values is the same. This module may be deleted, its function has been taken over by the relation_manager_.

mu_encd_key
: This internal subroutine extracts from the tuple the values that will make up the primary key of the relation and encodes them and builds a key string. It may also be deleted since its function has been moved to the relation_manager_.

mus_add_ind
: This internal subroutine adds the indices created by mu_build_indl to the tuple. It may be deleted since its function has been moved to the relation_manager_.

mus_del_ind
: This internal subroutine deletes the indices created by mu_build_indl from the tuple. It may be deleted since its function has been moved to the relation_manager_.

mus_add_ubtup
: This internal subroutines adds a tuple to the relation it may be deleted and calls to it (in mu_store) may be replaced with a call to the relation_manager_$put_tuple.

mus_mod_ubtup
: This internal subroutine calculates the length of the new tuple and calls iox_ to locate the tuple given its tuple_id and to rewrite it. It may be deleted and calls to this module (in mu_modify) may be replaced with a call to the relation_manager_$modify_tuple_by_id.

mu_store
: For this internal subroutine replace the call to mus_add_ubtup with a call to the relation_manager_$put_tuple. Delete the calls to mu_build_indl, mu_encd_key, mus_add_ind.

mu_modify
: For this internal subroutine replace the call to mus_mod_ubtup with a call to the relation_manager_$modify_tuple_by_id. Delete the calls to mu_build_indl, mu_del_ind, and mus_add_ind.

mrds_dsl_modify
: An alternative to modifying mu_modify is to delete it and modify this external subroutine to perform the bookkeeping tasks in mu_modify and to call the relation_manager_$modify_tuple_by_id directly. This would allow a reduction in calls to the modify_tuple_by_id since an array of tuple ids could be passed to modify_tuple_by_id instead of just one tuple id. Also if the selection expression controlling the modify ranges over just 1 tuple variable a single call to modify_tuple_by_search could be made, this would require more extensive changes to mrds_dsl_modify than just passing an array of tuple ids.

The modules that delete a tuple:

mus_del_ind
see above section

mu_delete
This internal subroutine calls iox_$seek_key and delete_record to delete a tuple and then calls mus_del_ind to delete the secondary indices. The call to mus_del_ind can be deleted and the calls to seek_key and delete_record replaced with a call to the relation_manager_$delete_tuple_by_id.

mrds_dsl_delete
An alternative to modifying mu_delete is to delete it and modify this external subroutine to perform the bookkeeping tasks in mu_delete and to call the relation_manager_$delete_tuple_by_id directly. This would allow a reduction in calls to the delete_tuple_by_id since an array of tuple ids could be passed to delete_tuple_by_id instead of just one tuple id. Also if the selection expression controlling the delete ranges over just 1 tuple variable a single call to delete_tuple_by_search could be made, this would require more extensive changes to mrds_dsl_delete than just passing an array of tuple ids.

The modules that retrieve a tuple:

mu_scan_records
This internal subroutine scans the relation sequentially and returns a pointer to the tuple. It may be deleted since this is the function of the relation_manager_$get_tuple_by_search used with the relation collection cursor and a specification that will include all tuples.

mus_loc_tup
This internal subroutine has two entry points. The "given_id" entry point calls iox_$control with a control order of record_status to get a pointer to the tuple given its tuple_id. The entry point "given_key" calls iox_$control with a control order of get_key to get the tuple_id and then calls the procedure which implements the given_id entry. The reason for having a separate module for doing this is to localize the manipulation of tuple_ids to vfile_ descriptors. This module can be deleted and calls to it replaced with calls to the relation_manager_$get_tuple_by_id and get_tuple_by_search.

mu_get_tid
This internal subroutine has two entry points. The "key" entry point calls mus_loc_tup$given_key. The "index" entry

point calls iox_$control with control orders of select and exclude to locate the tuples whose index (or key head) match the relation operator. These two entries can both be changed to call the relation_manager_$get_tuple_by_search.

mu_sec_get_tuple
This internal subroutine has two entry points. The "id" entry point calls mus_loc_tup$given_id to obtain a tuple from a tuple_id, this call can be replaced with a call to the relation_manager_$get_tuple_by_id. The "next" entry point returns the tuple_id and tuple for the next tuple. There are two definitions of next, first is by primary key order and second is by storage order. If primary key order is not used then a call is made to mu_scan_records. If the primary key order is used than calls to iox_$control with control orders of get_key and record_status and a call to iox_$position are made. Both the call to mu_scan_records and iox_ must be replaced with a call to the relation_manager_$get_tuple_by_search. For a key order search the the primary key collection cursor will be used along with a specification that will indicate a relative position of one. For an unordered search the relation collection cursor will be used with the same specification.

mrds_dsl_search
This internal subroutine is the one that executes the search program specified by the user's selection expression. It returns to its callers a pointer to a tuple and its tuple_id for a tuple that satisfies the expression. If the search program indicates that an entire relation is to be searched this module will call iox_$position to position to the beginning of the relation; it will further call mu_scan_records$init if the search is to be unordered. These two calls must be replaced with a call to the relation_manager_$get_tuple_by_search with a specification that indicates that the cursor should be positioned to the first record. If the search is to unordered the relation collection cursor will be used, else the cursor for one of the keys (primary or secondary) will be used. There is one other call to iox_ (iox_$control, order select) that is there do to a bug in mus_loc_tup$given_id which can be removed when mus_loc_tup is converted.

The modules that define a temporary relation:

mrds_dsl_define_temp_rel

This external subroutine creates, loads and destroys temporary relations. It creates a relation by calling mu_open_iocb_manager to attach and open and iocb and then calls iox_$control with a control order of record_status to force the newly created segment to be an MSF. This will have to be changed to call the relation_manager_$create_relation to create the relation and then to call the replacement for mu_open_iocb_manager to open the page_file and create the cursors. Relations are deleted by first calling iox_ to close, detach, and destroy the iocbs associated with the relation and then calling hcs_ to delete the MSF. This will have to be changed to call the relation_manager_$destroy_cursor and delete_relation.

Search Program Generation:

mrds_dsl_permute

This internal module calculates the cost of searching the tuple variables for each and-group in the selection expression. The minimum change required will be to change the cost values for each method of searching of a relation. This will have to be done by experimenting to determine each cost. A secondary change would be to change the algorithm for estimating the number of tuples which will selected from a relation to use the duplicate count for the selecting index instead of the duplicate count for all indices. Also it might be reasonable to use the min and max value of each attribute but the cost of determining this may make it prohibitive.

Tools

MRDS has a great many software tools which are not distributed as part of the MRDS product. These tools will also have to be converted. In most cases this will entail opening and closing the relations. Development of new tools to deal with page_files may also be required.

Unresolved Issues:

The following unresolved issues are divided into 3 groups. Group 1 concerns incompatible changes, both data returned to the user and performance. The incompatibilities may be eliminated at a cost of more effort during the conversion and more complex code to maintain. Can users live with these incompatible changes? Group 2 concerns a potential increase in performance verus more effort during conversion. We do not yet know how big or under what percentage of the circumstances a performance increase would be observed. Are they worth doing now? Group 3 concerns areas where we do not yet have enough information to plan effectively.

Group 1 - incompatible changes:

display_mrds_db_population

The use of the relation_manager_ will change the information that can be returned to the user. Information on the average selectivity of each secondary index can be returned (currently only the average selectivity over all secondary indices is returned) while the lengths of the secondary indices and a few other things cannot be. An alternative is to continue to call vfile_status_ for vfile_ relations and call the relation_manager_ only for page_file relations, this of course will increase the complexity of the code.

A decision must be made as to whether to call the relation_manager_ entry get_population or get_count. The command currently (and speedily) returns the exact number of tuples in the relation and it will continue to do so for vfile_ relations, get_population which is speedy returns an approximate count while get_count returns the exact count but may be to slow.

mu_get_rel_size

The same considerations as the second point under display_mrds_db_population with the added concern that the count will be used to estimate search program cost.

Group 2 - potential performance increase verus increased conversion effort:

mrds_dsl_modify, mu_modify, mrds_dsl_search
A "straight" conversion of mu_modify would be the simplest conversion task but moving part of mu_modify up into mrds_dsl_modify and further changing mrds_dsl_modify to form an array of N tuple ids and call the relation_manager_ to modify all N tuples at once should give increased performance when multiple tuples are modified via the same selection expression.

The module mrds_dsl_modify could be modified so that if only 1 tuple variable is used a call to relation_manager_$modify_tuple_by_search could be made instead of obtaining the tuple ids by calling mrds_dsl_search and then calling modify_tuple_by_id to modify each tuple individually. While it would not be necessary to modify mrds_dsl_search to not generate the search program it would increase the performance gain still more.

mrds_dsl_delete, mu_delete, mrds_dsl_search
The same points as mentioned for mrds_dsl_modify and mu_modify.

mrds_dsl_permute
Changing permute to take into consideration the average selectivity of each secondary index instead of just the average selectivity of all of the secondary indices would increase the effectiveness of calculating the cheapest search program.

Group 3 - more investigation needed:

tools
The tools library needs to be reviewed, those tools still useful need to be updated, those tools no longer used need to be deleted or archived. New tools may need to be created.

mu_open_iocb_manager
The cost or creating a cursor needs to be determined. The size of a cursor also needs to be determined, it might be reasonable to allow space for the creation of the maximum number of cursors. Alternatively new algorithms must be implemented for the dynamic creation and deletion of cursors. perhaps the maximum number of tuple variables should be reduced?

mrds_dsl_permute
New values of the cost constants must be determined by experiment.