To: MTB Distribution

From: Paul W. Benjamin

Date: 07/19/82

Subject: MRDS and DMS

Forum Meeting: >udd>Demo>dbm_test>con>MRDS_Development

## INTRODUCTION

In MR10.2 there will be an initial implementation of the Data Management System (DMS). A raft of documents has been written on this subject and I will assume that the reader is familiar with them. In MR10.2 the Multics Relational Data Store facility (MRDS) will be converted to use DMS. This document will provide a high level description of that conversion and detail some of the design considerations. Three documents will follow:

MTB-588 MRDS and DMS: Conversion Overview
MTB-589 MRDS and DMS: Vfile Relation Manager
MTB-590 MRDS and DMS: Conversion Design

## SOME BACKGROUND

MRDS, while often criticized, is also widely used. Ford alone estimates that there are approximately 1000 MRDS databases on their system of which at least 250 are in production and heavily used. These, of course, are all vfile_ databases. No matter how a conversion of these databases might be approached, it would be both costly and time consuming. There is also no guarantee that DMS will be as fast as vfile_, in fact there is a school of thought that it will be slower, there being a price to be paid in its increased protection and concurrency controls. In addition, the conversion to DMS will involve "gutting" much of MRDS. The DMS effort is ambitious, massive and not without risk. If MRDS was altered to communicate with the relation manager, and for some reason, DMS was not ready in the MR10.2 timeframe, there would be no MR10.2 MRDS. For these reasons, the determination was made that the MRDS group would write a vfile_ relation manager, concurrently with the conversion. What is a vfile_ relation manager? We will take the relation manager design and write a series of entrypoints that perform the same (or similar) tasks, using vfile_. Essentially, this means removing all vfile_ calls from MRDS and repackaging them elsewhere. With this: 1) users will be spared costly conversion; 2) users will not pay a performance penalty if DMS is slower than vfile_; and 3) there will continue to be a working MRDS, even if some unseen obstacle delays the DMS effort. Calls to the relation manager will be made to entry variables that point to the correct relation manager. So, in most cases, MRDS code will not be cognizant of the type of database it is diddling, but rather be calling an entry variable which points to the correct routine.

PEOPLE and PLANS

The primary goal of this effort is to be able to deliver to the DMS people a version of MRDS that uses the relation manager by December 1 of this year. In hopes of reaching that goal, the MRDS group has been staffed to a level that is unparalleled in recent memory. We have a total of six full-time people with myself as project leader, Roger Lackey, Noah Davids and 3 new people, Mike Kubicar, Donna Woodka and Ron Harvey. Donna is in the AEP and is tentatively scheduled to leave about a month before the above-mentioned goal. Obviously, we will attempt to get an extension. Note that, in addition to the size of the group, responsibility for LINUS has been moved to another project.

So, documents are currently being prepared, with meetings along the way to avoid controversy. A series of target dates follow. The are called targets rather than goals in that we have no solid feeling for the time that it will take to do these things. The dates:

| | |
|---|---|
| Documents complete: | FW231 |
| Begin vfile_relmgr: | FW230 |
| Begin Conversion: | FW233 |
| Complete vfile_relmgr: | FW239 |
| Complete Conversion: | FW244 |
| Complete Integration Testing: | FW248 |
| Complete Documentation: | FW305 |

## DESIGN CONSIDERATIONS

The remainder of this document consists of various considerations that will be of interest in the conversion.

### Which Relation Manager?

At database opening time, MRDS will determine whether it is dealing with a page file or a vfile. An array named relmgr_array will consist of entry variables named relmgr_$XXX. It will be initialized using 1 of 2 constant arrays, named relation_manager_array and vfile_relmgr_array. The relation_manager_array will consist of the entries in relation_manager_ and vfile_relmgr_array will consist of the entries in the vfile relation manager with names of the form vfile_relmgr_$XXX.

### Scope

While DMS has an excellent approach to concurrency control, MRDS is saddled with the existing scope mechanism. Existing applications must continue to function. MRDS has scope setting of r, d, m, s and null (r = read, d = delete, m = modify and s = store). DMS has only r, w and null. Further, MRDS allows the user to reduce scope, something that cannot be done under DMS. We could go so far as to make an incompatible change to MRDS, the DMS approach being satisfactory, but what happens to applications that assume the existence of the current scope mechanism? The solution is that the existing MRDS scope strategy will be retained, i.e. the scope shall be enforced by MRDS code. A reduced form of these scopes, however, will be communicated to the relation manager. The mapping is as follows (the notation dms indicates any combination of d, m or s):

| Permit Ops | User's Reduced Scope |
|---|---|
| r | r |
| rdms | rw |
| | |
| Prevent Ops | Other Users' Reduced Scope |
| null | rw |
| dms | r |
| rdms | null |

Further, the documentation will encourage the usage of the prevent op "null" on pagefiles that have concurrency control. This usage is the most efficient, since redundant MRDS code will not be executed and the concurrency controls of the relation manager will be fully utilized.

## Temporary Relations

Neither concurrency nor protection should be used for temporary relations. Concurrency is meaningless as temp_rels are single-user relations. Protection is not worthwhile either, since users cannot modify temp_rels and are per-process. There has been some discussion of a future capability of modifying temp_rels and the temp_rel code should contain a comment that indicates that this issue should be revisited if that change is made. For now, the call to relation_manager_$create_relation, when creating temp_rels, should be made with both concurrence_switch and protected_switch in pf_creation_info OFF.

## Transactions

Most calls to the relation manager, when accessing protected files, will require a transaction to be in progress. What follows is a list of relation_manager_ entrypoints, broken into what requires a transaction and what does not:

| TRANSACTION REQUIRED: | NO TRANSACTION REQUIRED: |
|---|---|
| create_index | close |
| delete_tuple_by_id | create_cursor |
| delete_tuple_by_search | create_relation |
| destroy_index | destroy_cursor |
| get_count | destroy_relation_by_opening |
| get_description | destroy_relation_by_path |
| get_population | open |
| get_max_and_min_attributes | |
| get_tuple_by_id | |
| get_tuple_by_search | |
| get_tuple_id | |
| modify_tuple_by_id | |
| modify_tuple_by_search | |
| put_tuple | |

For those entrypoints requiring transactions, MRDS will start one if there is none already in progress. A transaction may, however, have been started by either LINUS or the user. So, suppose the user hits the break key in the midst of some sort of transaction. A listener level is pushed but the transaction still exists. The user can invoke MRDS again, and this time no transaction would be started because one was in progress. This operation would complete and the user could type "start" and resume the original operation. If that operation aborted for some reason the transaction would be rolled back, and with it, the work done at the higher listener level. This is clearly an unfortunate prospect.

To avoid this, MRDS will establish condition handlers when it starts a transaction. On quit, it will call txnmgr_$suspend_txn. This will prohibit any further transaction-mode work in her process,

and no new transaction can be started. On start_, it will call txnmgr_$resume_txn. On cleanup, it will call txnmgr_$abort_txn.

Rollbacks and Aborts

DMS will do an admirable job of rolling back page files when a transaction aborts or is rolled back. The problem is that not everything that MRDS will modify in the course of a transaction will be on page files. What MRDS will change can range from automatic storage to the data model. To compound the problems that this poses, the MRDS code that modified both pagefile and data model, for example, may have successfully completed and no longer be in the stack when the transaction aborts. I will try and deal with this range of possibilities on a point by point basis.

Automatic Storage

This is the easiest to deal with. All automatic storage that is altered within the confines of a transaction should be restored in a rollback or abort. This is not as difficult as it may sound. Any initialization that occurs should be done inside the transaction loop. Any variable data that may change within the transaction should be saved prior to entering the transaction. If a rollback and restart occurs, the data can then be returned to its original state. Automatic storage should not be a concern for aborted transaction, because the stack and its storage will disappear. Likewise for transactions that are rolled back by the user.

Per-Process or Per-Opening Data

This begins to get more interesting. Things that fall in this category include internal static variables and the resultant. Clearly, copies of internal static variables or portions of the resultant should be made and then reinstated at rollback or abort time. What, however, can be done with this data when the user has started and then aborts or rolls back the transaction? MRDS may no longer be in the stack, may have "successfully completed" its work. The answer to this is non-trivial. What we feel is needed is a means of providing "rollback handlers" of some kind. I need to be able to tell the transaction manager that, if a rollback occurs, procedure X needs to be executed. Further, the transaction manager needs to be able to keep a list of these procedures because a transaction can encompass many operations, none cognizant of the others. It now seems unlikely that there will be such a feature in MR10.2, although there is some sentiment for getting to it eventually. What MRDS is faced with is identifying those data items that are effected by this and insure that none are going to represent a critical problem.

This is perhaps the riskiest area of this whole effort, from the point of view of MRDS.

One example is that statistics about searching are kept in the resultant and then used by the search program in choosing the most optimal method of search. If MRDS cannot roll back the potentially bogus information left residual by an aborted transaction, then search optimization will suffer. Note that this is only an issue where the user has started her own transaction and it aborted. This is what I would call a nonfatal design flaw. It can be tolerated but not indefinitely.

Note further that the resolution to most of these issues is to convert everything in sight to page files. Unfortunately that would essentially constitute a rewrite of MRDS and is not doable given personpower and time constraints. (Sure would like to, though...)

Per-Database Data

The only thing that falls into this category is the model. Fortunately, the model is fairly static and the only transaction work that will effect is in the restructuring subsystem. The problems here are similar to those discussed in the last paragraph, but much more clear cut. Until such time as some sort of rollback handling capability is implemented, the user will not be able to start her own transactions when using rmdb. The restructuring programs will merely start transactions and, if informed by the transaction manager that one is already in progress, refuse to proceed.

Template for Conversion

The following is a template that shows how a typical MRDS module will interact with DMS:

```
mrds_typical:
          proc (...);

/* Save off significant data */
/* for rollback or aborts */

          txn_id = 0;
restart:  code = 0;
          if must_start_txn ()
               then do;
               on cleanup call cleanup_handler;
               on start_ call start_handler;
               on quit call quit_handler;
               call begin_transaction;
          end;
          call initialize;

/* initialize will simply set defaults etc. */

/* Now we can call relmgr and access the DB */

          call done;
          return;

must_start_txn:
          proc returns (bit (1));

          if db_type = vfile_
               then return ("0"b);
          if db_protected = "0"b
               then return ("0"b);

          call txnmgr_$get_current_txn_id (tid, tix, mode, ec);

          if ec = error_table_$no_current_txn
               then return ("1"b);
          return ("0"b);
     end;

cleanup_handler:
          proc;
          call abort_transaction;
          call restore;
     end;
```

```
start_handler:
          proc;
          call txnmgr_$resume_txn (txn_id, ec);
     end;

quit_handler:
          proc;
          call txnmgr_$suspend_txn (txn_id, ec);
          call continue_to_signal_ (ec);
     end;

restore:  proc;

/* Restore the "significant data" saved above */

     end;

done:     proc;
          if txn_id ^= 0
               then if code = 0
                    then call commit_transaction;
               else do;
                    call restore;
                    if <should restart>
                         then do;
                         call rollback_transaction;
                         goto restart;
                    end;
                    else call abort_transaction;
               end;
     end;

begin_transaction:
          proc;
          call txnmgr_$begin_txn (mode, bj_old, txn_id, ec);
     end;

commit_transaction:
          proc;
          call txnmgr_$commit_txn (txn_id, ec);
     end;

abort_transaction:
          proc;
          call txnmgr_$rollback_txn (txn_id, ec);
     end;

end mrds_typical;
```