


Retek[®] Merchandising System 10.0



Operations Guide – Volume 3: Batch Program Overview





The software described in this documentation is furnished under a license agreement and may be used only in accordance with the terms of the agreement.

No part of this documentation may be reproduced or transmitted in any form or by any means without the express written permission of Retek Inc., Retek on the Mall, 950 Nicollet Mall, Minneapolis, MN 55403.

Information in this documentation is subject to change without notice.

Retek provides product documentation in a read-only-format to ensure content integrity. Retek Customer Support cannot support documentation that has been changed without Retek authorization.

Corporate Headquarters:

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403
888.61.RETEK (toll free US)
+1 612 587 5000

Retek[®] Merchandising System[™] is a trademark of Retek Inc.

Retek and the Retek logo are registered trademarks of Retek Inc.

©2002 Retek Inc. All rights reserved.

All other product names mentioned are trademarks or registered trademarks of their respective owners and should be treated as such.

Printed in the United States of America.

European Headquarters:

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46

Fax: +44 (0)20 7563 46 10



Customer Support

Customer Support hours:

8AM to 5PM Central Standard Time (GMT-6), Monday through Friday, excluding Retek company holidays (in 2002: Jan. 1, May 27, July 4, July 5, Sept. 2, Nov. 28, Nov. 29, and Dec. 25).

Customer Support emergency hours:

24 hours a day, 7 days a week.

Contact Method	Contact Information
Phone	US & Canada: 1-800-61-RETEK (1-800-617-3835) World: +1 612-587-5000
Fax	(+1) 612-587-5100
E-mail	support@rettek.com
Internet	www.retek.com/support Retek's secure client Web site to update and view issues
Mail	Retek Customer Support Retek on the Mall 950 Nicollet Mall Minneapolis, MN 55403

When contacting Customer Support, please provide:

- Product version and program/module name.
- Functional and technical description of the problem (include business impact).
- Detailed step by step instructions to recreate.
- Exact error message received.
- Screen shots of each step you take.

Contents

Chapter 1 – Introduction.....	1
Chapter 2 – Restart and recovery	3
Table descriptions & definitions	3
Data model discussion.....	8
Physical set-up.....	8
Table and file-based restart/recovery	9
API functional descriptions	11
Query-based commit thresholds.....	20
Chapter 3 – Multi-threading.....	21
Threading description.....	21
Threading function for query-based.....	21
Restart view for query-based.....	22
Thread scheme maintenance	24
Batch maintenance	25
Scheduling and initialization of restart batch.....	26
Pre- and post-processing	26
Chapter 4 – Array processing	27
Chapter 5 – Input and output formats	29
General interface discussion.....	29
Detail only files	29
Electronic data interchange (EDI).....	31
Interfaces supported by RMS.....	32

Chapter 1 – Introduction

This document describes the following features of Retek Merchandising System (RMS) 10.0 batch (Pro*C) programs:

- Restart and recovery
- Multi-threading
- Commit thresholds
- Array processing
- Input and output formats to external applications and entities

Chapter 2 – Restart and recovery

RMS has implemented a restart/recovery process in most of its batch architecture. The general purpose of restart/recovery is to:

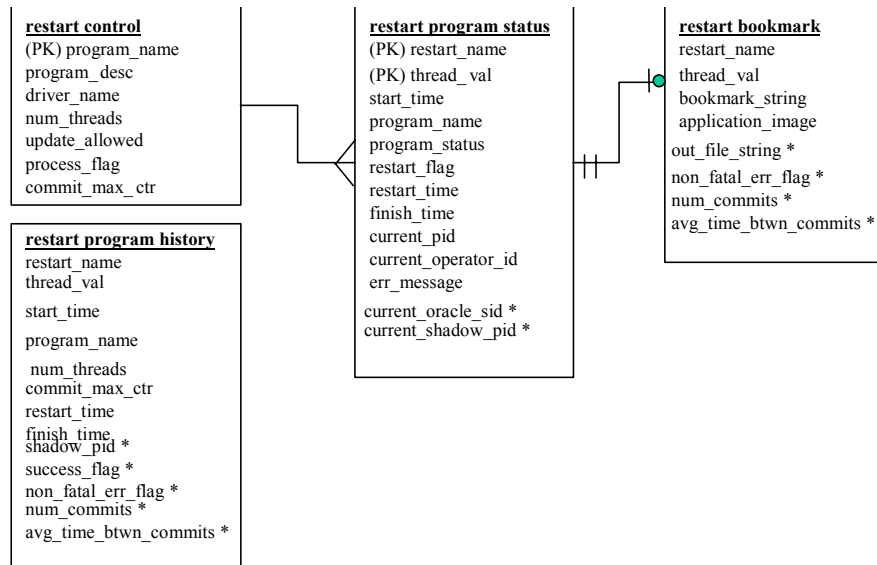
- Recover a halted process from the point of failure
- Prevent system halts due to large numbers of transactions
- Allow multiple instances of a given process to be active at the same time

Further, the RMS restart/recovery tracks batch execution statistics and does not require DBA authority to execute.

The restart capabilities revolve around a program's logical unit of work (LUW). A batch program processes transactions, and commit points are enabled based on the LUW. LUWs consist of a relatively unique transaction key (such as sku/store) and a maximum commit counter. Commit events take place after the number of processed transaction keys meets or exceeds the maximum commit counter. For example, every 10,000 sku/store combinations, a commit occurs. At the time of the commit, key data information that is necessary for restart is stored in the restart tables. In the event of a handled or un-handled exception, transactions will be rolled back to the last commit point, and upon restart the key information will be retrieved from the tables so that processing can continue from the last commit point.

Table descriptions & definitions

The RMS restart/recovery process is driven by a set of four tables. Refer to Diagram 1 for the entity relationship diagram, followed by table descriptions.



Note: The fields with asterisks (*) are only used by new batch programs of release 9.0 or later.

Diagram 1 – Restart/recovery table relationships

restart_control

The restart_control table is the master table in the restart/recovery table set. One record will exist on this table for each batch program that is run with restart/recovery logic in place. The restart/recovery process uses this table to determine:

- whether the restart/recovery is table-based or file-based,
- the total number of threads used for each batch program,
- the maximum records that will be processed before a commit event takes place,
- and the driver for the threading (multi-processing) logic.

restart_control			
(PK) program_name	varchar2	25	batch program name
program_desc	varchar2	50	a brief description of the program function
driver_name	varchar2	25	driver on query, for example, department (non-updatable)
num_threads	num	10	number of threads used for current process
update_allowed	varchar2	2	indicates whether user can update thread numbers or if done programmatically
process_flag	varchar2	1	indicates whether process is table-based (T) or file-based (F)
commit_max_ctr	num	6	numeric maximum value for counter before commit occurs

restart_program_status

The restart_program_status table is the table that holds record keeping information about current program processes. The number of rows for a program on the status table will be equal to its num_threads value on the restart_control table. The status table is modified during restart/recovery initialization and close logic. For table-based processing, the restart/recovery initialization logic will assign the next available thread to a program based on the program status and restart flag. For file-based processing, the thread value is passed in from the input file name. Once a thread has been assigned the program_status is updated to prevent the assignment of that thread to another process. Information will be logged on the current status of a given thread, as well as record keeping information such as operator and process timing information.

Setup note: Allow row level locking and 'dirty reads' (do not wait for rows to be unlocked for table read).

restart_program_status			
(PK)restart_name	vvarchar2	50	Program name
(PK)thread_val	num	10	thread counter
start_time	date		dd-mon-yy hh:mi:ss
program_name	vvarchar2	25	program name
program_status	vvarchar2	15	started, aborted, aborted in init, aborted in process, aborted in final, completed, ready for start
restart_flag	vvarchar2	1	automatically set to 'N' after abnormal end, must be manually set to 'Y' for program to restart
restart_time	date		dd-mon-yy hh:mi:ss
finish_time	date		dd-mon-yy hh:mi:ss
current_pid	num	15	starting program id
current_operator_id	vvarchar2	20	operator that started the program
err_message	vvarchar2	255	record that caused program abort & associated error message
current_oracle_sid	num	15	Oracle SID for the session associated with the current process
current_shadow_pid	num	15	O/S process ID for the shadow process associated with the current process. It is used to locate the session trace file when a process is not finished successfully.

restart_program_history

The restart_program_history table will contain one record for every successfully completed program thread with restart/recovery logic. Upon the successful completion of a program thread, its record on the restart_program_status table will be inserted into the history table. Table purgings will be at user discretion.

restart_program_history			
(PK) restart_name	varchar2	50	
(PK) thread_val	Num	10	
(PK) start_time	Date		
program_name	varchar2	25	
num_threads	Num	9	
commit_max_ctr	num	6	
restart_time	date		
finish_time	date		
shadow_pid	num	15	O/S process ID for the shadow process associated with the process. It is used to locate the session trace file.
success_flag	varchar2	1	Indicates whether the process finished successfully (reserved for future use)
non_fatal_err_flag	varchar2	1	Indicates whether non-fatal errors have occurred for the process
num_commits	num	12	Total number of commits for the process. The possible last commit when restart/recovery is closed is not counted.
avg_time_btwn_commits	num	12	Accumulated average time between commits for the process. The possible last commit when restart/recovery is closed is not counted.

restart_bookmark

When a restart/recovery program thread is currently active, its state is started or aborted, a record for it will exist on the restart_bookmark table. Restart/recovery initialization logic inserts the record into the table for a program thread. The restart/recovery commit process updates the record with the following restart information:

a concatenated string of key values for table processing

a file pointer value for file processing

application context information such as counters and accumulators

The restart/recovery closing process will delete the program thread record if the program finishes successfully. In the event of a restart, the program thread information on this table will allow the process to begin from the last commit point.

restart_bookmark			
restart_name	varchar2	50	
thread_val	num	10	
bookmark_string	varchar2	255	character string of key of last committed record
application_image	varchar2	1000	application parameters from the last save point
out_file_string	varchar2	255	Concatenated file pointers (Unix sometimes refers to these as stream positions) of all the output files from the last commit point of the current process. It is used to return to the right restart point for all the output files during restart process.
non_fatal_err_flag	varchar2	1	Indicates whether non-fatal errors have occurred for the current process.
num_commits	num	12	number of commits for the current process. The possible last commit when restart/recovery is closed is not counted.
avg_time_btwn_commits	num	12	average time between commits for the current process. The possible last commit when restart/recovery is closed is not counted.

v_restart_x

Restart views will be used for query-based programs that require multi-threading. Separate views will be created for each threading driver, e.g. department or store. A join will be made to a view based on threading driver to force the separation of discrete data into particular threads. Please see the threading discussion for more details.

v_restart_x		
driver_name	varchar2	- example dept, store, region, etc.
num_threads	number	total number of threads in set (defined on restart control)
driver_value	number	- will be the numeric value of the driver_name
thread_val	number	thread value defined for driver_value and num_threads combination

Data model discussion

Why restart_program_status and restart_bookmark are separate tables

The initialization process needs to fetch all of the rows associated with restart_name/schema, but will only update one row. The commit process will continually lock a row with a specific restart_name and thread_val. The data involved with these two processes is separated into two tables to reduce the number of hangs that could occur due to locked rows. Even if you allow 'dirty reads' on locked rows, a process will still hang if it attempts to do an update on a locked row. The commit process is only interested in a unique row, so if we move the commit process data to a separate table with row level (not page level) locking, there will not be contention issues during the commit. With the separate tables, the initialization process will now see fewer problems with contention because rows will only be locked twice, at the beginning and end of the process.

Physical set-up

The restart/recovery process needs to be as robust as possible in the event of database related failure. The costs outweigh the benefits of placing the restart/recovery tables in a separate database. The tables should, however, be set up in a separate, mirrored table space with a separate rollback segment.

Table and file-based restart/recovery

The restart/recovery process works by storing all the data necessary to resume processing from the last commit point. Therefore, the necessary information will be updated on the `restart_bookmark` table before the processed data is committed. Query-based and file-based modules will store different information on the restart tables, and will therefore call different functions within the restart/recovery API to perform their tasks.

When a program's process is query-based, that is, a module is driven by a driving query that processes the retrieved rows, then the information that is stored on the `restart_bookmark` table is related to the data retrieved in the driving query. If the program fails while processing, the information that is stored on the restart-tables can be used in the conditional where-clause of the driving query to only retrieve data that has yet to be processed since the last commit event.

File-based processing, however, simply needs to store the file location at the time of the last commit point. This file's byte location is stored on the `restart_bookmark` table and will be retrieved at the time of a restart. This location information will be used to seek forward in the re-opened file to the point at which the data was last committed.

Because there is different information being saved to and retrieved from the `restart_bookmark` table for each of the different types of processing, different functions will need to be called to perform the restart/recovery logic. The query-based processing will call the `restart_init` and `restart_commit` functions while the file-based processing will call the `restart_file_init` and `restart_file_commit` functions.

In addition to the differences in API function calls, the batch processing flow of the restart/recovery will differ between the files. Table-based restart/recovery will need to use a priming fetch logical flow, while the file-based processing will usually read lines in a batch. Table-based processing requires its structure to ensure that the LUW key has changed before a commit event can be allowed to occur, while the file-based processing does not need to evaluate the LUW, which can typically be thought of as the type of transaction being processed by the input file.

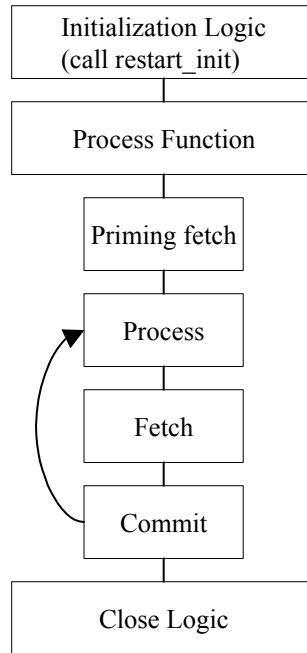


Diagram 2 - Table-Based Restart/Recovery Program Flow

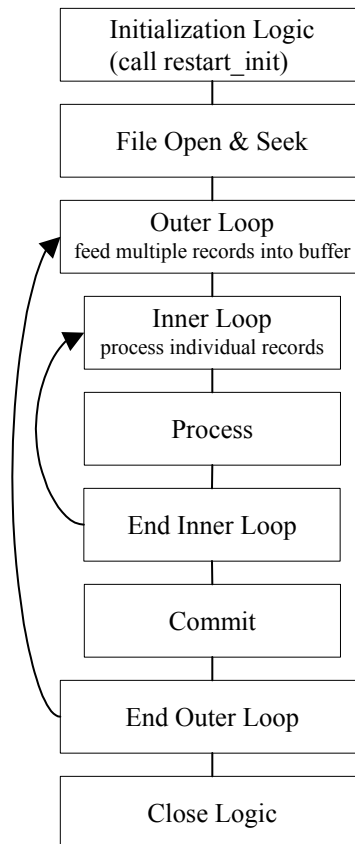


Diagram 3 - File-Based Restart/Recovery Program Flow

Initialization Logic:

- Variable declarations
- File initialization
- Call restart_init() function - will determine start or restart logic
- First fetch on driving query

Start Logic: initialize counters/accumulators to start values

Restart Logic:

- Parse application_image field on bookmark table into counters/accumulators
- Initialize counters/accumulators to values of parsed fields

Process/commit Loop:

- Process updates and manipulations
- Fetch new record
- Create varchar from counters/accumulators to pass into application_image field on restart_bookmark table
- Call restart_commit()

Close logic:

- Reset pointers
- Close files/cursors
- Call restart_close()

API functional descriptions

restart_init:

An initialization function for table-based batch processing.

The process gathers information from the restart control tables

- Total number of threads for a program and thread value assigned to current process.
- Number of records to loop through in driving cursor before commit (LUW).
- Start string - bookmark of last commit to be used for restart or a null string if current process is an initial start and initializes the restart record-keeping (restart_program_status).
- Program status is changed to 'started' for the first available thread.
- Operational information is updated: operator, process, start_time, etc. and bookmarking (restart_bookmark) tables.
- On an initial start, a record is inserted

- On restart, the start string and application context information from the last commit is retrieved.

restart_file_init:

An initialization function for file-based batch processing. It is called from program modules.

- 1 The process gathers information from the restart control tables:
 - number of records to read from file for array processing and for commit cycle
 - file start point- bookmark of last commit to be used for restart or 0 for initial start
- 2 The process initializes the restart record-keeping (restart_program_status):
 - program status is changed to 'started' for the current thread
 - operational information is updated: operator, process, start_time, etc.
- 3 The process initializes the restart bookmarking (restart_bookmark) tables:
 - on an initial start, a record is inserted
 - on restart, the file starting point information and application context information from the last commit is retrieved

restart_commit:

A function that commits the processed transaction for a given number of driving query fetches. It is called from program modules.

The process updates the restart_bookmark start string and application image information if a commit event has taken place:

- the current number of driving query fetches is greater than or equal to the maximum set in the restart_program_status table (and fetched in the restart_init function)
- the bookmark string of the last processed record is greater than or equal to the maximum set in the restart_program_status table (and fetched in the restart_init function)
- the bookmark string increments the counter
- the bookmark string sets the current string to be the most recently fetched key string

restart_file_commit:

A function that commits processed transactions after reading a number of lines from a flat file. It is called from program modules.

The process updates the restart_bookmark table:

- start_string is set to the file pointer location in the current read of the flat file
- application image is updated with context information

restart_close:

A function that updates the restart tables after program completion.

The process determines whether the program was successful. If the program finished successfully:

- the restart_program_status table is updated with finish information and the status is reset
- the corresponding record in the restart_bookmark table is deleted
- the restart_program_history table has a copy of the restart_program_status table record inserted into it
- the restart_program_status is re-initialized

If the program ends with errors

- the transactions are rolled back
- the program_status column on the restart_program_status table is set to 'aborted in *' where * is one of the three main functions in batch: init, process or final
- the changes are committed

parse_array_args:

This function parses a string into components and stuffs results into multidimensional array. It is only called within API functions and will never be called in program modules.

The process is passed a string to parse and a pointer to an array of characters.

The first character of the passed string is the delimiter.

restart_file_write:

This function will append output in temporary files to final output files when a commit point is reached. It is called from program modules.

restart_cat:

This function contains the logic that appends one file to another. It is only called within the restart/recovery API functions and will never be called directly in program modules.

Restart headers and libraries

The restart.h and the std_err.h header files are included in retek.h to utilize the restart/recovery functionality.

restart.h

This library header file contains constant, macro substitutions, and external global variable definitions as well as restart/recovery function prototypes.

The global variables that are defined include:

- the thread number assigned to the current process
- the value of the current process's thread maximum counter
 - for table-based processing, it is equal to the number of iterations of the driving query before a commit can take place
 - for file-based processing, it is equal to the number of lines that will be read from a flat file and processed using a structured array before a commit can take place
- the current count of driving query iterations used for table-based processing or the current array index used in file-based processing
- the name assigned to the program/logical unit of work by the programmer. It is the same as the restart_name column on the restart_program_status, restart_program_history, and restart_bookmark tables

std_rest.h

This library header file contains standard restart variable declarations that are used visible in program modules.

The variable definitions that are included are:

- the concatenated string value of the fetched driving query key that is currently being processed
- the concatenated string value of the fetched driving query key that is next to be processed
- the error message passed to the restart_close function and updated to restart_program_status
- concatenated string of application context information, for example, counters & accumulators
- the name of the threading driver, for example, department, store, warehouse, etc.
- the total number of threads used by this program
- the pointer to pass to initialization function to retail number of threads value

New restart headers & libraries

The current RMS restart/recovery library has been modified to enhance maintainability, enable easier coding and improve performance. While the current mechanism and functionality of batch restart/recovery are preserved, the following improvements and enhancements have been done:

- Organize global variables associated with restart recovery
- Allow the batch developer full control of restart recovery variables parameter passing during initialization
- Remove temporary write files to speed up the commit process
- Move more information and processing from the batch code into the library code
- Add more information into the restart recovery tables for tuning purposes

retek_2.h

This library header file is included by all C code within Retek and serves to centralize system includes, macro defines, globals, function prototypes, and, especially, structs for use in the new restart/recovery library.

The globals used by the old restart/recovery library are all discarded. Instead, each batch program declares variables needed and calls `retek_init()` to get them populated from restart/recovery tables. Therefore, only the following variables are declared:

- `gi_no_commit`: flag for `NO_COMMIT` command line option (used for tuning purposes)
- `gi_error_flag`: fatal error flag
- `gi_non_fatal_err_flag`: non-fatal error flag

In addition, a `rtk_file` struct is defined to handle all file interfaces associated with restart/recovery. Operation functions on the file struct are also defined.

```

#define NOT_PAD                1000  /* Flag not to pad
thread_val */

#define PAD                    1001  /* Flag to pad
thread_val at the end */

#define TEMPLATE              1002  /* Flag to pad
thread_val using filename template */

#define MAX_FILENAME_LEN      50

typedef struct
{
    FILE* fp;                  /* File
pointer */
    char  filename[MAX_FILENAME_LEN + 1]; /* Filename */
    int   pad_flag;           /* Flag whether to pad thread_val
to filename */
} rtk_file;

int   set_filename(rtk_file* file_struct, char*
file_name, int pad_flag);
FILE* get_FILE(rtk_file* file_struct);
int   rtk_print(rtk_file* file_struct, char* format,
...);
int   rtk_seek(rtk_file* file_struct, long offset, int
whence);

```

The parameters `retk_init()` needs to populate are required to be passed in using a format known to `retk_init()`. A struct is defined here for this purpose. An array of parameters of this struct type is needed at each batch program. Other requirements are:

- Need to be initialized at each batch program.
- The lengths of name, type and sub_type should not exceed the definitions here.
- Type can only be: "int", "uint", "long", "string", or "rtk_file".
- For type "int", "uint" or "long", use "" as sub_type.
- For type "string", sub_type can only be "S" (start string) unless the string is the thread value or number of threads, in which case use "" as sub_type or "I" (image string).

- For type "rtk_file", sub_type can only be "I" (input) or "O" (output).

```
#define NULL_PARA_NAME      51
#define NULL_PARA_TYPE     21
#define NULL_PARA_SUB_TYPE  2
typedef struct
{
    char name[NULL_PARA_NAME];
    char type[NULL_PARA_TYPE];
    char sub_type[NULL_PARA_SUB_TYPE];
} init_parameter;
```

New restart/recovery functions

Starting from release 9.0, all new batch programs are coded using the new restart/recovery functions. Batch programs using the old restart/recovery API functions are still in use. Therefore, Retek is currently maintaining two sets of restart/recovery libraries.

`int retek_init(int num_args, init_parameter *parameter, ...)`

`retex_init` initializes restart/recovery (for both table- and file-based):

- 1 Pass in `num_args` as the number of elements in the `init_parameter` array, then the `init_parameter` array, then variables a batch program needs to initialize in the order and types defined in the `init_parameter` array. Note that all int, uint and long variables need to be passes by reference.
- 2 Get all global and module level values from databases.
- 3 Initialize records for `RESTART_PROGRAM_STATUS` and `RESTART_BOOKMARK`.
- 4 Parse out user-specified initialization variables (variable arg list).
- 5 Return `NO_THREAD_AVAILABLE` if no qualified record in `RESTART_CONTROL` or `RESTART_PROGRAM_STATUS`.
- 6 Commit work.

int retek_commit(int num_args, ...)

retek_commit checks and commits if needed (for both table- and file-based):

- 1 Pass in num_args, then variables for start_string first, and those for image string (if needed) second. The num_args is the total number of these two groups. All are string variables and are passed in the same order as in retek_init();
- 2 Concatenate start_string either from passed in variables (table-based) or from ftell of input file pointers (file-based);
- 3 Check if commit point reached (counter check and, if table-based, start string comparison);
- 4 If reached, concatenated image_string from passed in variables (if needed) and call internal_commit() to get out_file_string and update RESTART_BOOKMARK;
- 5 If table-based, increment pl_current_count and update ps_cur_string.

int commit_point_reached(int num_args, ...)

commit_point_reached checks if the commit point has been reached (for both table- and file-based). The difference between this function and the check in retek_commit() is that here the pl_current_count and ps_cur_string are not updated. This checking function is designed to be used with retek_force_commit(), and the logic to ensure integrity of LUW exists in user batch program. It can also be used together with retek_commit() for extra processing at the time of commit.

- 1 Pass in num_args, then all string variables for start_string in the same order as in retek_init(). The num_args is the number of variables for start_string. If no start_string (as in file-based), pass in NULL.
- 2 For table-based, if pl_curren_count reaches pl_max_counter and if newly concatenated bookmark string is different from ps_cur_string, return 1; otherwise return 0.
- 3 For file-based, if pl_curren_count reaches pl_max_counter return 1; otherwise return 0.

`int retek_force_commit(int num_args, ...)`

`retek_force_commit` always commits (for both table- and file-based):

- 1 Pass in `num_args`, then variables for `start_string` first, and those for image string (if needed) second. The `num_args` is the total number of these two groups. All are string variables and are passed in the same order as in `retek_init()`.
- 2 Concatenate `start_string` either from passed in variables (table-based) or from full of input file pointers (file-based).
- 3 Concatenated `image_string` from passed in variables (if needed) and call `internal_commit()` to get `out_file_string` and update `RESTART_BOOKMARK`.
- 4 If table-based, increment `pl_current_count` and update `ps_cur_string`.

`int retek_close(void)`

`retek_close` closes restart/recovery (for both table- and file-based):

- 1 If `gi_error_flag` or `NO_COMMIT` command line option is `TRUE`, rollback all database changes.
- 2 Update `RESTART_PROGRAM_STATUS` according to `gi_error_flag`.
- 3 If no `gi_error_flag`, insert record into `RESTART_PROGRAM_HISTORY` with information fetched from `RESTART_CONTROL`, `RESTART_PROGRAM_BOOKMARK` and `RESTART_PROGRAM_STATUS` tables.
- 4 If no `gi_error_flag`, delete `RESTART_BOOKMARK` record.
- 5 Commit work.
- 6 Close all opened file streams.

`int retek_refresh_thread(void)`

Refreshes a program's thread so that it can be run again.

- 1 Updates the `RESTART_PROGRAM_STATUS` record for the current program's `PROGRAM_STATUS` to be 'ready for start'.
- 2 Deletes any `RESTART_BOOKMARK` records for the current program.
- 3 Commits work.

`void increment_current_count(void)`

`increment_current_count` increases `pl_current_count` by 1.

Note: This is called from `get_record()` of `inrface.pc` for file-based I/O.

`int parse_name_for_thread_val(char* name)`

`parse_name_for_thread_val` parses thread value from the extension of the specified file name.

`int is_new_start(void)`

`is_new_start` checks if current run is a new start; if yes, return 1; otherwise 0.

Query-based commit thresholds

The restart capabilities revolve around a program's logical unit of work (LUW). A batch program processes transactions and enables commit points based on the LUW. An LUW is comprised of a transaction key (such as item-store) and a maximum commit counter. Commit events occur after a given number of transaction keys are processed. At the time of the commit, key data information that is necessary for restart is stored in the restart table. In the event of a handled or un-handled exception, transactions will be rolled back to the last commit point. Upon restart the restart key information will be retrieved from the tables so that processing can resume with the unprocessed data.

Chapter 3 – Multi-threading

Processing multiple instances of a given program can be accomplished through “threading”. This requires driving cursors to be separated into discrete segments of data to be processed by different threads. This will be accomplished through stored procedures that will separate threading mechanisms (for example, departments or stores) into particular threads given value (for example, department 1001) and the total number of threads for a given process.

File-based processing will not truly “thread” its processing. The same data file will never be acted upon by multiple processes, however, Multi-threading will be accomplished by dividing the data into separate files each of which will be acted upon by a separate process. The thread value is related to the input file. This is necessary to ensure that the appropriate information can be tied back to the relevant file in the event of a restart.

RMS 10.0 increased the store length to ten digits. Therefore, thread values, which can be based upon the store number, should allow ten digits as well. Due to the thread values being declared as ‘C’ variables of type int (long), the system is restricting thread values to nine digits.

This does not mean that you cannot use ten digit store numbers. It means that if you do use ten digit store numbers you cannot use them as thread values.

Threading description

The use of multiple threads or processes in Retek batch processing will increase efficiency and decrease processing time. The design of the threading process has allowed maximum flexibility to the end user in defining the number of processes over which a program should be divided.

Originally, the threading function was going to be used directly in the driving queries. This was found, however, to be unacceptably slow. Instead of using the function call directly in the driving queries, the designs call for joining driving query tables to a view (for example, v_restart_store) that includes the function.

Threading function for query-based

A stored procedure has been created to determine thread values. Restart_thread_return returns a thread value derived from a numeric driver value, such as department number, and the total number of threads in a given process. Clients should be able to determine the best algorithm for their design, and if a different means of segmenting data is required, then either the restart_thread_return function can be altered, or a different function can be used in any of the views in which the function is contained.

Currently the `restart_thread_return` function is a very simple modulus routine:

```
CREATE OR REPLACE FUNCTION
RESTART_THREAD_RETURN(in_unit_value  NUMBER,
                      in_total_threads NUMBER)

RETURN NUMBER IS
ret_val NUMBER;
BEGIN
ret_val := MOD(ABS(in_unit_value),in_total_threads) +
1;
RETURN ret_val;
END;
```

Restart view for query-based

Each restart view will have four elements:

- the name of the threading mechanism, `driver_name`
- the total number of threads in a grouping, `num_threads`
- the value of the driving mechanism, `driver_value`
- the thread value for that given combination of `driver_name`, `num_threads`, and driver value, `thread_val`

The view will be based on the `restart_control` table and an information table such as `DEPS` or `STORES`. A row will exist in the view for every driver value and every total number of threads value. Therefore, if a client were to always use the same number of threads for a given driver (dept, store, etc.), then the view would be relatively small. As an example, if all of a client's programs threaded by department have a total of 5 threads, then the view will contain only one value for each department. For example, if there are 10 total departments, 10 rows will exist in `v_restart_dept`. However, if the client wants to have one of the programs to have ten threads, then there will be 2 rows for every department: one for five total threads and one for ten total threads (for example, if 10 total departments, 20 rows will exist in `v_restart_dept`). Obviously, clients should be advised to keep the number of total thread values for a thread driver to a minimum to reduce the scope of the table join of the driving cursor with the view.

Below is an example of how the same driver value can result in differing thread values. This example uses the `restart_thread_return` function as it currently is written to derive thread values.

Driver_name	num_threads	driver_val	thread_val
DEPT	1	101	1
DEPT	2	101	2
DEPT	3	101	3
DEPT	4	101	2
DEPT	5	101	2
DEPT	6	101	6
DEPT	7	101	4

Below is an example of what a distribution of stores might look like given 10 stores and 5 total threads:

Driver_name	num_threads	driver_val	thread_val
STORE	5	1	2
STORE	5	2	3
STORE	5	3	4
STORE	5	4	5
STORE	5	5	1
STORE	5	6	2
STORE	5	7	3
STORE	5	8	4
STORE	5	9	5
STORE	5	10	1

View syntax:

The following is an example of the syntax needed to create the view for the multi-threading join, created with script (see threading discussion for details on `restart_thread_return` function):

```
create or replace view v_restart_store as
  select rc.driver_name driver_name,
         rc.num_threads num_threads,
         s.store driver_value,
         restart_thread_return(s.store, rc.num_threads)
        thread_val
  from restart_control rc, store s
  where rc.driver_name = 'STORE'
```

There is a different threading scheme used within Sales Audit. Because Sales Audit needs to run 24 hours a day and seven days a week, there is no batch window. This means that there may be batch programs running at the same time that there are online users. Sales Audit solved this concurrency problem by creating a locking mechanism for data that is organized by store days. These locks provide a natural threading scheme. Programs that cycle through all of the store day data attempt to lock the store day first. If the lock fails, the program simply goes on to the next store day. This has the effect of automatically balancing the workload between all of the programs executing.

Thread scheme maintenance

All program names will be stored on the `restart_control` table along with a functional description, the query driver (dept, store, class, etc.) and the user-defined number of threads associated with them. Users should be able to scroll through all programs; to view the name, description, and query driver; and if the `update_allowed` flag is set to true to modify the number of threads (update is set to true).

File-based

File based processing does not truly “multi-thread” and therefore the number of threads defined on `restart_control` will always be one. However, a `restart_program_status_record` will need to be created for each input file that is to be processed for the program module. Further, the thread value that is assigned should be part of the input file name. The `restart_parse_name` function that is included in the program module will parse the thread value from the program name and use that to determine the availability and restart requirements on the `restart_program_status` table.

Refer to the beginning of this multi-threading section for a discussion of limits on using large (greater than nine digits) thread values.

Query-based

When the number of threads is modified in the restart_control table, the form should first validate that no records for that program are currently being processed in the restart_program_status_table (that is, all records = 'Completed'). The program should insert or delete rows depending on whether the new thread number is greater than or less than the old thread number. In the event that the new number is less than the previous number all records for that program_name with a thread number greater than the new thread number will be deleted. If the new number is greater than the old number new rows will be inserted. A new record will be inserted for each restart_name/thread_val combination.

For example if the batch program suddenly has its number of processes changed from 2 to 3, then an additional row (3) will be added to the restart_program_status table. Likewise, if the number of threads was reduced to 1 in this example, rows 2 and 3 would be deleted.

Original restart_program_status table:

row #	restart_name	thread_val	program_name	etc...
1	WinSal -main	1	WinSal	...
2	WinSal -main	2	WinSal	...

restart_program_status table after insert:

row #	restart_name	thread_val	program_name	etc...
1	WinSal -main	1	WinSal	...
2	WinSal -main	2	WinSal	...
3	WinSal -main	3	WinSal	...

restart_program_status table after delete:

row #	restart_name	thread_val	program_name	etc...
1	WinSal -main	1	WinSal	...

Users should also be able to modify the commit_max_ctr column in restart_program_status table. This will control the number of iterations in driving query or the number of lines read from a flat file that determine the logical unit of work (LUW).

Batch maintenance

Users should be able to view the status of all records in restart_program_status table. This is where the user will come to view error messages from aborted programs, and statistics and histories of batch runs. The only fields that will be modifiable will be program_status and restart_flag. The user should be able to reset the restart_flag to 'Y' from 'N' on records with a status of aborted, started records to aborted in the event of an abend (abnormal termination), and all records in the event of a restore from tape/re-run of all batch.

Scheduling and initialization of restart batch

Before any batch with restart/recovery logic is run, an initialization program should be run to update the status in the restart_program_status table. This program should update the program_status to 'ready for start' wherever a record's program_status is 'completed'. This will leave unchanged all programs that ended unsuccessfully in the last batch run.

Pre- and post-processing

Due to the nature of the threading algorithm, individual programs might need a pre or a post program run to initialize variables or files before any of the threads have run or to update final data once all the threads are run. The decision was made to create pre-programs and post-programs in these cases rather than let the restart/recovery logic decide whether the currently processed thread is the first thread to start or the last thread to end for a given program.

Chapter 4 – Array processing

Retek batch architecture uses array processing to improve performance wherever possible. Instead of processing SQL statements using scalar data, data is grouped into arrays and used as bind variables in SQL statements. This improves performance by reducing the server/client and network traffic.

Array processing is used for select, insert, delete, and update statements. Retek typically does not statically define the array sizes, but uses the restart maximum commit variable as a sizing multiple. Users should keep this in mind when defining the system's maximum commit counters.

An important factor to keep in mind when using array processing is that Oracle does not allow a single array operation to be performed for more than 32000 records in one step. The Retek restart/recovery libraries have been updated to define macros for this value: `MAX_ORACLE_ARRAY_SIZE`.

All batch programs that use array processing need to limit the size of their array operations to `MAX_ORACLE_ARRAY_SIZE`.

If the commit max counter is used for array processing size, check it after the call to `restart_init()` and, if necessary, reset it to the maximum value if greater. If `retek_init()` is used to initialize, check the returned commit max counter and reset it to the maximum size if it is greater. In case of `retek_init()`, reset the library's internal commit max counter by calling `extern int limit_commit_max_ctr(unsigned int new_max_ctr)`.

If some other variable is used for sizing the array processing, the actual array-processing step will have to be encapsulated in a calling loop that performs the array operation in sub segments of the total array size where each sub-segment is at most `MAX_ORACLE_ARRAY_SIZE` large. Currently all Retek batch programs are implemented this way.

Chapter 5 – Input and output formats

Retek batch processing will utilize input from both tables and flat files. Further, the outcome of processing can both modify data structures and write output data. Interfacing Retek with external systems is the main use of file based I/O.

General interface discussion

To simplify the interface requirements, Retek requires that all in-bound and out-bound file-based transactions adhere to standard file layouts. There are two types of file layouts, detail-only and master-detail, which are described below.

An interfacing API exists within Retek to simplify the coding and the maintenance of input files. The API provides functionality to read input from files, ensure file layout integrity, and write and maintain files for rejected transactions.

Standard file layouts

The RMS interface library supports two standard file layouts; one for master/detail processing, and one for processing detail records only. True sub-details are not supported within the RMS base package interface library functions.

A 5-character identification code or record type identifies all records within an I/O file, regardless of file type. Valid record type values include the following:

- FHEAD—File Header
- FDETL—File Detail
- FTAIL—File Tail
- THEAD—Transaction Header
- TDETL—Transaction Detail
- TTAIL—Transaction Tail

Each line of the file must begin with the record type code followed by a 10-character record ID.

Detail only files

File layouts have a standard file header record, a detail record for each transaction to be processed, and a file trailer record. Valid record types are FHEAD, FDETL, and FTAIL.

Example:

```
FHEAD0000000000STKU1996010100000019960929
FDETL0000000001SKU100000040000011011
FDETL0000000001SKU100000050003002001
FDETL0000000001SKU100000050003002001
FTAIL00000000020000000003
```

Master and detail files

File layouts will have a standard file header record, a set of records for each transaction to be processed, and a file trailer record. The transaction set will consist of a transaction set header record, a transaction set detail for detail within the transaction, and a transaction trailer record. Valid record types are FHEAD, THEAD, TDETL, TTAIL, and FTAIL.

Example:

```
FHEAD0000000001RTV 19960908172000
THEAD000000000200000000001234199609091202000000000003R
TDETL000000000300000000001234000001SKU10000012
TTAIL0000000004000001
THEAD000000000500000000001234199609091202001215720131R
TDETL000000000600000000001234000001UPC400100002667
TDETL000000000700000000001234000001UPC400100002643 0
TTAIL0000000008000002
FTAIL00000000090000000007
```

Record Name	Field Name	Field Type	Default Value	Description
File Header	File Type Record Descriptor	Char(5)	FHEAD	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	File Type Definition	Char(4)	n/a	Identifies transaction type
	File Create Date	Date	Create date	Date file was written by external system
Transaction Header	File Type Record Descriptor	Char(5)	THEAD	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	Transaction Set Control Number	Char(14)	Specified by external system	Used to force unique transaction check
	Transaction Date	Char(14)	Specified by external system	Date the transaction was created in external system
Transaction Detail	File Type Record Descriptor	Char(5)	TDETL	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file

Record Name	Field Name	Field Type	Default Value	Description
	Transaction Set Control Number	Char(14)	Specified by external system	Used to force unique transaction check
	Detail Sequence Number	Char(6)	Specified by external system	Sequential number assigned to detail records within a transaction
Transaction Trailer	File Type Record Descriptor	Char(5)	TTAIL	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	Transaction Detail Line Count	Number(6)	Sum of detail lines	Sum of the detail lines within a transaction
File Trailer	File Type Record Descriptor	Char(5)	FTAIL	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	Total Transaction Line Count	Number(10)	Sum of all transaction lines	All lines in file less the file header and trailer records

Electronic data interchange (EDI)

Starting with release 7.0, EDI files used or created by RMS are in a generic format: RMS no longer supports particular EDI standards. By processing EDI output and input in a generic format, RMS is no longer limited to a single standard, which allows Retek customers to better utilize any and all standards they choose to use. Translating EDI input and output files into any format from any format by third-party software is an industry “best practice”.

Formerly, EDI transactions in RMS conformed to ASC X12/VICS (version 3040) and ANA/TRADACOMS standards. EDI transactions are now expected to be in a format that adheres to the RMS file interfacing standards. Both in-bound and out-bound files are written in a fixed field layout with standard file header and trailer records. Transaction information is included in master/detail or detail-only records. The layouts are consistent with interface files used elsewhere in the RMS.

RMS EDI batch processes write out-bound transaction files into the generic layout format, which are then translated by the third-party software into the standard required by each trading partner. The post-translated versions are transmitted to the trading partner. In-bound transactions should be formatted by the trading partner in a predetermined standard, transmitted, and then translated by the Retek client’s translation software into the generic file layout. The generic file is used as the input file for RMS EDI batch processing.

It is impractical for Retek to continue to maintain code that supports any particular EDI standard. There are multiple viable standards that are utilized by vendors and clients. Further, those standards have multiple versions. Most retailers are already using software to map and translate EDI transactions into the required standard or version. There are excellent third-party software packages, such as Sterling Software's Gentran™ translator, that effectively translate in-bound and out-bound transactions into the necessary formats. The use of third-party translation software is not only the common business practice, but also the best business practice of today's retailer.

Interfaces supported by RMS

The RMS batch modules include interfaces with many external systems. Point of sales systems both send sales files through Retek Sales Audit and receive item pricing and maintenance files from RMS. EDI translation packages translate in-bound and out-bound transactions from standard EDI formats to Retek's generic interface layouts. Planning and forecasting systems receive sales history and item information and provide order and replenishment information. Finally, external open to buy systems receive histories and provide new parameters to RMS.