TPump in a Continuous Environment

Bob Hahn & Carrie Ballinger, Active Data Warehouse Center of Expertise, April, 2001

This is a new document in a series of papers designed to assist Teradata users successfully evolve to active data warehousing. All papers in the Assembling the Teradata Active Data Warehouse Series can be found by Teradata users by selecting a link on the front page of the SupportLink web site, or by NCR associates on the ADW COE home page at http://infobahn.sandiegoca.ncr.com/tadc/adw/ .

An important part of a Teradata Active Data Warehouse is the ability to provide data that is as close to up-to-date as it needs to be for the user applications. Web-based requests and intra-time-zone applications are putting the squeeze on data warehouses to stay open for longer. At the same time, inside many companies a drive is underway to sustain a more up-to-the minute picture of customers, of inventory, of pricing.

TPump, Fastload and MultiLoad all provide various alternatives in planning an active load strategy. Several specific options available for you today include:

1. Short, frequent FastLoad executions, loading data into an empty table for later insert/select processing into the final target table
2. Frequent MultiLoad executions directly into the target table
3. Periodic executions of TPump in a batch mode
4. TPump, fed from a queue, in a continuously executing mode

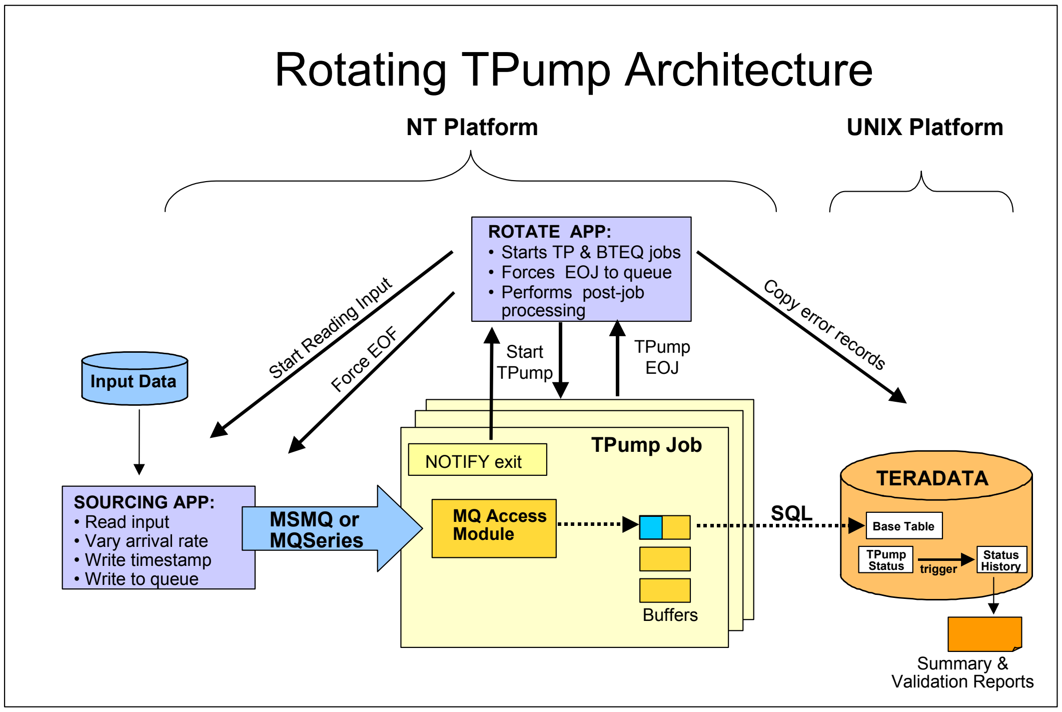
This paper addresses the last of these choices: TPump in a continuous environment. For those who desire to load data through a queue in an ongoing fashion, a continuous TPump architecture is worth considering. Because this approach involves some planning and effort, a prototype has been set up and executed, the results of which are detailed in this paper. The contents of this paper include:

Overview of the Continuous TPump Process

The diagram below illustrates the architecture used in this prototype. It is based on an automatic and tightly coordinated rotation of two different TPump instances, both fed by the same queue. Two Visual Basic applications were written. The first one feeds the queue, simulating a myriad of upstream data feeds. The second one, standing in for the job scheduler, supports the automatic hand-off of control between TPumps, and to handle the end-of-job processing. Sections later in this paper will detail exactly what these applications do and will describe information about the queue.

This prototype attempts to reflect real world conditions. An average of 1% errors is spread randomly throughout the input file and some number of rows with errors appear in every TPump job. The row being loaded into Teradata is wide: about 650 bytes and contains 33 columns, over 20 of them varchar. The data arrival rate can be varied at different times of days. A generally-available release of both TPump and Teradata software are utilized.

When activated, this architecture is successful in providing a less than 2-second gap between the final read of the queue from one TPump job and the first read of the queue by the next. It also demonstrates a technique for having timely access to, as well as keeping, vital statistics about, job success or failure. Further, it has built in a mechanism for tracking latency of input records for further tuning or performance trend analysis.



This prototype focuses on the central feeding and loading functionality. Some of the more site-specific activities that will take place in conjunction with this type of strategy, such as reconstruction and reprocessing of error rows, is not included in this prototype.

The Platform and the Data

This section provides detail on the hardware and software used in this prototype, as well as information about the table participating in the load.

Hardware and Software

The prototype platform consisted of the following:

- 2-node 4800, four 550 MHz CPUs per node, 2 GB memory on each node
- A separate client node, four 700 MHz CPUs
- 10 AMPs per node on the server for a total of 20 AMPs system-wide
- Teradata V2R4.0 software
- Teradata Parallel Data Pump for Windows (i486) 01.03.01.02
- Client node was LAN-connected to the two server nodes via private 100 megabit subnet

The Database Table Being Loaded

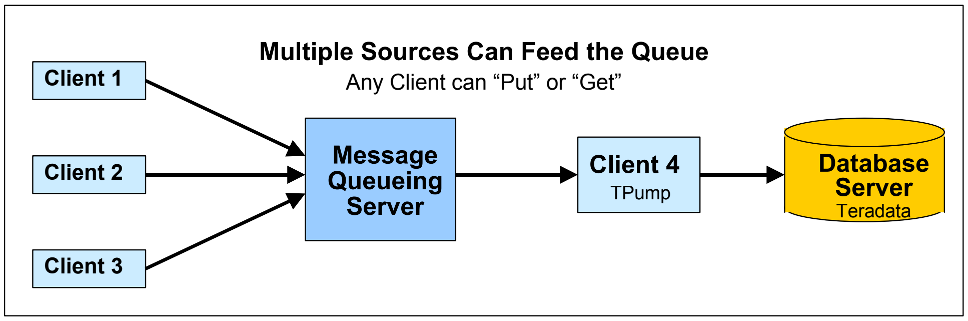
The initial table was 134,640,000 rows, with each TPump instance inserting 1,224,000 rows. The average row width was 350 bytes and the table was defined with 33 columns (see table definition below).

Some of the TPump tests contain error rows. Five different columns were used to simulate five different types of errors for these test cases. Types of errors simulated included duplicate rows, invalid contents in a date column, an integer column populated with character data, character data in a decimal column, and an invalid date range. The layout of the base table that TPump is executing against follows:

```
CREATE SET TABLE test50g.person ,
    NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
    (
     id CHAR(20) NOT NULL,
     last_update DATE FORMAT 'mm/dd/yyyy',
     zip CHAR(10),
     Prefix_Description VARCHAR(4),
     fname VARCHAR(20) ,
     lname VARCHAR(20) ,
     mname VARCHAR(20) ,
     Suffix_Description VARCHAR(4) ,
     address1 VARCHAR(40)  ,
     Credit_Type VARCHAR(12) ,
     address2 VARCHAR(40)  ,
     city VARCHAR(20)  ,
     telephone CHAR(14)  ,
     email VARCHAR(55)  ,
     state VARCHAR(20) ,
     country VARCHAR(20) ,
     Home_Status  VARCHAR(12) ,
     age INTEGER ,
     gender CHAR(1)   ,
     occupation VARCHAR(50) ,
     income DECIMAL(9,2) FORMAT 'z,zzz,zz9.99',
     os VARCHAR(15),
     Marital_Status VARCHAR(12),
     education VARCHAR(24),
     isp VARCHAR(100),
     dma_lu VARCHAR(12) ,
     browser VARCHAR(55),
     creditnum VARCHAR(20),
     creditexp DATE FORMAT 'mm/dd/yyyy',
     Work_Status VARCHAR(12),
     events VARCHAR(50),
     queue_time TIMESTAMP(6) TITLE 'Time-Queued',
     db_time TIMESTAMP(6) DEFAULT CURRENT_TIMESTAMP(6)) )
UNIQUE PRIMARY INDEX ( id );
```

Background on Queueing

Queues were used in this prototype because queues provide several unique characteristics that are well-suited for continuous loading. First, like the water in your faucet, the contents of a queue are able to continuously flow, or be held back, ready to flow at any moment. Second, queues can receive data from many, literally thousands, of different platforms or geographies. Queues alleviate the need to bring all data together in one location first where it can then be merged into a single input file in order to load it. Third, queues are reliable, they are guaranteed to deliver data placed into them.



Message queueing provides reliable, secure, time-independent, cross-platform application messaging services. Examples of Message Queueing products are IBM's MQSeries, Microsoft MSMQ, Progress Software's SonicMQ, and BEA TopEnd RTQ.

Using MQSeries as an example, MQSeries clients indirectly use MQServer to read and write message buffers to and from message queues. Communication protocols are hidden from the application. One or many applications can put messages to a queue that is serviced by one or more applications.

Message queueing offers a solution to low-latency data feeds to the Active Data Warehouse. Transaction data can be put to a message queue in real time. Data Warehouse load utilities can asynchronously read these queues. Asynchronous means that the sending application gets an immediately acknowledgement of the put and the reader may process the message immediately or some time later.

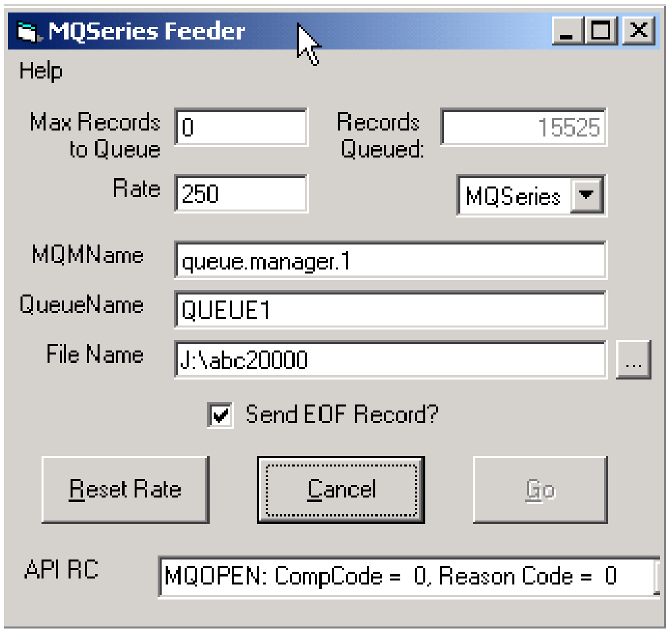
The Processes Built Around TPump

Two Visual Basic programs were built to support the continuous activity. Both the Sourcing Application and the Rotating Application are explained below.

The Sourcing Application

This application was written to simulate the output coming from a business application that generates data intended for the warehouse. A flat file of 1.2 million records was created before the prototype was run. This flat file contains all of the input records that are eventually placed on the queue, read by TPump, and inserted into the database. The role of the Sourcing Application is simply to read this flat file, and write each record into the queue.

There is a setting in the prototype to specify the rate that the queue will be fed (the Rate variable is shown in the figure below). Either a specific rate can be specified or the Sourcing Application will read and enqueue data as fast as it can. This setting can be changed at any point while the TPump jobs are running, and is intended to simulate a normal change of arrival rate in the real world. The Sourcing Application is designed to respond to the change in rate immediately by increasing or slowing down the rate of writing to the queue.



Max Records to Queue: 0

Records Queued: 15525

Rate: 250

MQSeries

MQMName: queue.manager.1

QueueName: QUEUE1

File Name: J:\abc20000

[✓] Send EOF Record?

API RC: MQOPEN: CompCode = 0, Reason Code = 0

The feeder picture above represents the writer. “Records Queued” represents the number of records placed in the queue since this writer job began. TPump, as the reader, is independent from this queue-writer application. “Rate” represents the rate of messages per second that will be written to the queue. Using queues, reading and writing are asynchronous.

The Sourcing Application writes a timestamp on the input row just before it places the row onto the queue. This provides a record of when the row began its journey through the continuous load process. This timestamp, combined with a second one (which reflects when the row is written into the database), is instrumentation that was placed into the prototype in order to monitor latency and better observe performance. Latency actually achieved in this prototype is discussed in the last section of this paper.

Data could have been placed on the queue in any number of ways. This Sourcing Application was a simple, single-client approach that allows the prototype to execute. Its main purpose is to provide queued input to the Access Module within TPump, which is more central to the prototype.

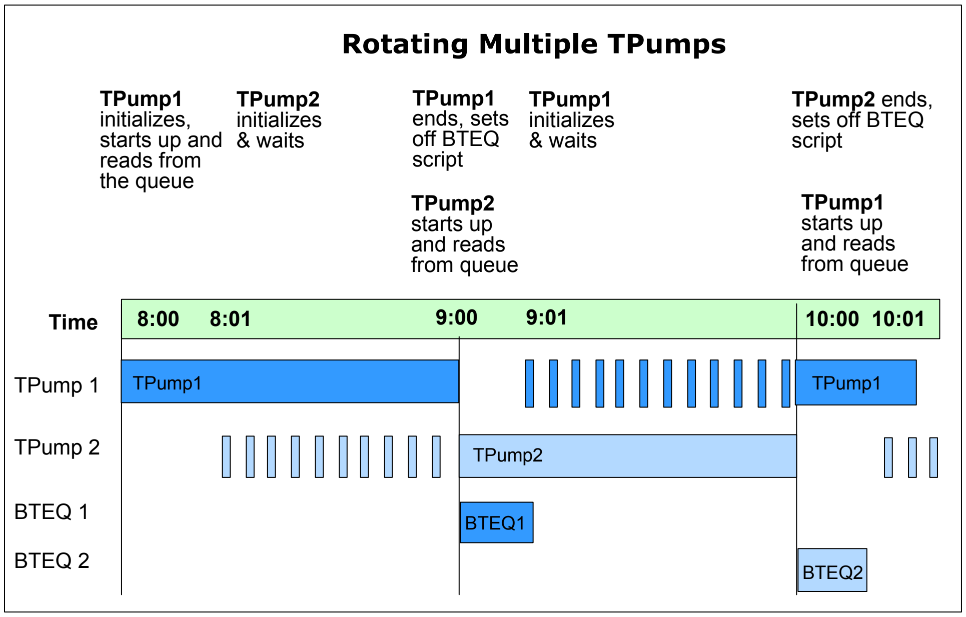
The Rotater Application Coordinates the Two TPump Instances

The point at which a TPump job completes is a good management point for housekeeping. Single, discrete executions, with a clear beginning and ending point, offer several advantages over a single continuous TPump job:

1. You have quicker confirmation of the success or failure of this portion of the load.
2. You have more timely access to tables that contain errors so errors can be reprocessed quickly.
3. You get end-of-job statistics sooner, for job validation and performance analysis.
4. You have the opportunity to change parameters on subsequent iterations of TPump, based on changes in the environment.

This second Visual Basic program acts as a job scheduler, responsible for managing the starting and stopping of TPump instances and the post-job processing that follows each TPump instance. This program alleviates the need for DBA intervention as the jobs switch on and off. In a production environment, this Rotater Application would be replaced by the on-site job scheduler, such as Control/M from BMC.

The diagram below shows how the Rotater Application synchronizes the two TPump and two BTEQ jobs over time.


Rotating Multiple TPumps
TPump1 initializes, starts up and reads from the queue
TPump2 initializes & waits
TPump1 ends, sets off BTEQ script
TPump1 initializes & waits
TPump2 ends, sets off BTEQ script
TPump2 starts up and reads from queue
TPump1 starts up and reads from queue
Time
8:00
8:01
9:00
9:01
10:00
10:01
TPump 1
TPump 2
BTEQ 1
BTEQ 2
TPump1
TPump2
BTEQ1
TPump1
BTEQ2

Following the clock times reflected in that diagram, at 8:00 the Rotater Application starts up the first TPump instance. There will be several seconds of TPump initialization time before TPump1 reads from the queue. After performing job initialization, TPump1 opens the queue in exclusive access mode. When a queue is opened in exclusive mode, that client and that client alone can read from the queue, until such time as that client closes the queue.

As soon as TPump1 begins actively reading from the queue, TPump2 is started and attempts to read from the queue in exclusive mode. In the diagram above, this activity takes place at 8:01, but it could be at any time after TPump1 begins to read the queue. Because TPump1 has already opened the queue in exclusive mode, TPump2 completes job initialization and then waits.

What happens during TPump Initialization

During job initialization, TPump does some or all of the following: Create macros, create log table, create error tables, log on all sessions.

TPump2 will continue to attempt to open the queue every second, and will continue to be blocked until the TPump1 marks the queue as closed.   The one-second interval between retries is an arbitrary number, designed to control the lag time between when one TPump instance ends and the next one begins.   It could be a larger or a smaller number, and can be easily modified within the Access Module. Additional information on the Access Modules used will be covered in the next section.

After some amount of specified time, the Rotater Application forces the current TPump job to terminate by writing an end-of-job (EOJ) record to the queue. In the diagram above, this event takes place one hour after the TPump job began to read from the queue. The Access Module was coded to recognize this EOJ record when it arrives through the queue, depending on the queueing mechanism being used:

- When MSMQ is used, the Rotator Application associates a special property with the message, a property which tells the Access Module that this message is the EOJ marker.

- For MQSeries, the Rotator Application defines a zero-length message body, which has the same meaning to the Access Module.

These are only two of many other choices in how to pass the EOJ indicator via the queue.

As soon as the Access Module within TPump reads the EOJ marker from the queue TPump1 begins to shut down.   The time between the Rotating Application's issuance of the EOJ record and TPump1's closing of the queue will depend on the queue depth and the rate of movement through the queue, as well as the length of time job termination takes.

What happens during TPump Termination

During job termination, TPump does some or all of the following: Flushes all buffers, deletes error table (if empty), writes to the monitor table.

This queue was set up to process in the normal First-In-First-Out (FIFO) manner, so the EOF record goes to the end of the queue.   As soon as the record makes it to the front of the queue the Access Module reads it and notes it is an EOJ marker.   The Access Module then passes control to TPump1 to do job termination, after which the Access Module closes the queue.

TPump2 immediately begins to read from the queue, and having already performed job initialization, begins to load input rows into Teradata.   In this prototype it takes 1 second from the time the EOJ marker is issued by the Rotating Application until TPump1 stops reading from the queue.   Because the Access Module used by both TPump1 and TPump2 was designed to attempt to open the queue every second, within one second of TPump1 closing the queue, TPump2 begins to read.

The Rotater Application and Post-TPump Processing

At the end of the TPump instance, there is an opportunity to do various management activities. This section discusses the handful of these tasks that were chosen for this prototype, but others could have been included.

In addition to managing the starting and stopping of TPump instances, the Rotater Application sends off one BTEQ job for each TPump at the time that TPump instance completes. The purpose of this BTEQ job is to consolidate error rows and manage cleanup of the error tables. Consider one TPump execution and its associated BTEQ job as a pair.

Actually, this TPump-BTEQ pair are tightly associated in a couple of ways. First, the BTEQ script uses as its logon and password the strings that were specified in the TPump job it is associated with. Second, the output that is produced by each pair is named similarly, as both use the TPump job name, which itself contains a start date and a start time. Here is a sample of the output file names produced by a pair that resulted from a TPump2 script:

TPump job name:	20010220_035548_tpump_02
TPump output name:	20010220_035548_tpump_02.output
BTEQ script output name:	20010220_035548_bteq_02.output

1% errors were intentionally spread across the input file that feeds the queue. So at EOJ the error table will contain rows. This BTEQ job reads the rows from each error table produced by the associated TPump job and inserts the rows into a Teradata table, without altering their format or content. The distinct TPump job name could be added to the error row when it is inserted into the consolidated error table, for later identification purposes. Finally, the error rows are deleted from the TPump error files. The BTEQ script for a TPump1 rotation looks like this:

```
.logon dbc/rmh,rmh;
insert into et_hist_table1 select * from ET_JOB1;
drop table ET_JOB1;
```

The layout of the error history table, which is a mirror image of the error table, follows:

```
CREATE MULTISET TABLE CONTINUOUS.et_hist_table1 ,NO FALLBACK ,
    ( ImportSeq BYTEINT,
     DMLSeq BYTEINT,
     SMTSeq BYTEINT,
     ApplySeq BYTEINT,
     sourceseq INTEGER,
     DataSeq BYTEINT,
     ErrorCode INTEGER,
     ErrorMsg VARCHAR(255) CHARACTER SET LATIN NOT CASESPECIFIC,
     ErrorField SMALLINT,
     HostData VARBYTE(31728))
PRIMARY INDEX ( ImportSeq ,DMLSeq ,SMTSeq ,ApplySeq ,sourceseq ,DataSeq );
```

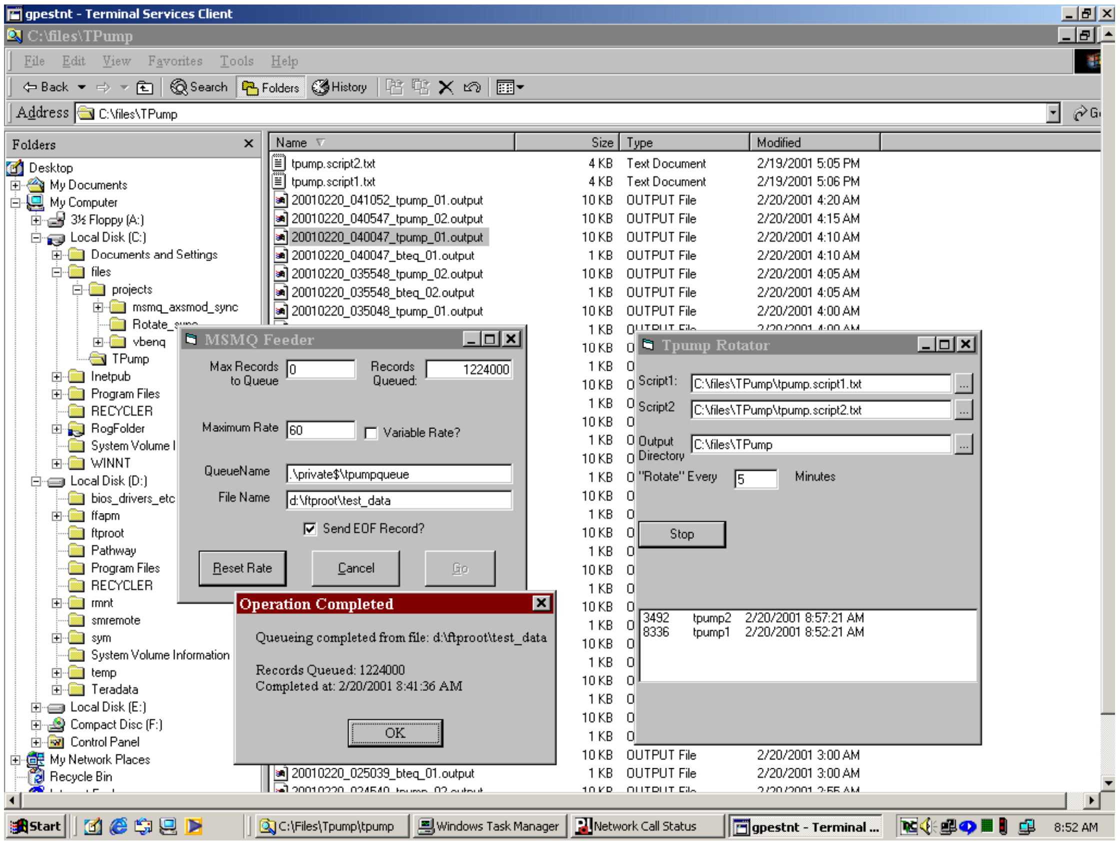
The process of dropping the error tables has been incorporated into the BTEQ script. Because the name of the error tables is embedded within each TPump script, each iteration of TPump1 will have the same error table names (in the above BTEQ script example, the error table name is

ET_JOB1), as will each TPump2.   While the TPump job names will differ, depending on the date and time that each TPump1 was started, the error table names will always be the same, whatever is hard-coded in the TPump script.

Output from both TPump and BTEQ jobs will be redirected to an output directory that has been specified at the beginning of the prototype.

The Rotater Application is a simple and slimmed-down implementation of automatic continuous TPump job management. In the real world the functions provided by the Rotater Application would fall to the job scheduler or the workflow engine at the site.   Most such workflow scheduling packages offer significantly more elaborate workload scenarios and complex rules, so numerous ways to enrich this prototype are readily available.

Below is an illustration of some of the components used by the Rotater Application:



```
Name                              Size   Type           Modified
tpump.script2.txt                 4 KB   Text Document  2/19/2001 5:05 PM
tpump.script1.txt                 4 KB   Text Document  2/19/2001 5:06 PM
20010220_041052_tpump_01.output  10 KB   OUTPUT File    2/20/2001 4:20 AM
20010220_040547_tpump_02.output  10 KB   OUTPUT File    2/20/2001 4:15 AM
20010220_040047_tpump_01.output  10 KB   OUTPUT File    2/20/2001 4:10 AM
20010220_040047_bteq_01.output    1 KB   OUTPUT File    2/20/2001 4:10 AM
20010220_035548_tpump_02.output  10 KB   OUTPUT File    2/20/2001 4:05 AM
20010220_035548_bteq_02.output    1 KB   OUTPUT File    2/20/2001 4:05 AM
20010220_035048_tpump_01.output  10 KB   OUTPUT File    2/20/2001 4:00 AM
                                 10 KB   OUTPUT File    2/20/2001 3:00 AM
20010220_025039_bteq_01.output    1 KB   OUTPUT File    2/20/2001 3:00 AM
```

MSMQ Feeder

Max Records to Queue	0
Records Queued:	1224000
Maximum Rate	60
Variable Rate?	☐
QueueName	.\private$\tpumpqueue
File Name	d:\ftproot\test_data
Send EOF Record?	☑

Reset Rate · Cancel · Go

Operation Completed

Queueing completed from file: d:\ftproot\test_data

Records Queued: 1224000
Completed at: 2/20/2001 8:41:36 AM

OK

Tpump Rotator

Script1:	C:\files\TPump\tpump.script1.txt
Script2	C:\files\TPump\tpump.script2.txt
Output Directory	C:\files\TPump
"Rotate" Every	5 Minutes

Stop

```
3492    tpump2    2/20/2001 8:57:21 AM
8336    tpump1    2/20/2001 8:52:21 AM
```

In the background is the Windows directory where the output has been directed. The box in the foreground on the right side has a place to specify the directory chosen by the administrator, this case the directory is C:\files\TPump.

Also in the foreground is a box that allows the administrator to select the arrival rate of input data going in the queue, as well as place to name the flat file where the input data is housed and the queue itself resides.

Inside the TPump Job

Both TPump instances used the same parameters for all tests. They included:

- Pack Factor 10
- Number of sessions 20
- Checkpoint 30
- ROBUST ON
- SERIALIZE OFF

In a real world environment, where arrival rates and system resource usage levels are likely to differ throughout the day, it might be desirable to alter one or more of these parameters while the TPump instances are rotating.

For example, it might be noticed that the queue depth is getting high. If you plan ahead, you could arrange for this condition to automatically cause an EOJ marker to be sent to the current TPump instance with a high queue priority, and at the same time raise the pack factor and/or the number of sessions on the next TPump instance to run. The logic to perform all of this could be embedded in the workflow manager.

Access Modules

Access Modules are used by the Teradata Client utilities to read and write client data sources. The interface is open to support custom Access Modules in addition to those that NCR provides. NCR offers Disk, Tape, Pipes, and the OLE-DB Consumer. Additional product Access Modules are intended.

Informal Message Queueing Access Modules were created based on the needs of this prototype; one for Microsoft Message Queue and another for IBM MQSeries. Both are attached in the Appendix of this paper. Both were used with TPump on Windows 2000 to stream data read from a message queue into Teradata.

The Access Modules used in this prototype are simple and are intended to be be a starting point for customized implementations. NCR will offer supported product Message Queueing Access Modules. The current target is MQSeries Client on Windows and Solaris SPARC for TPump, Multiload, Fastload, FastExport and TeraBuilder. These will offer the function, quality, internationalization, and error handling, as required in a supported product.

Queues can at times support high levels of activity, while at other times the arrival rate may plummet. Access Modules can be useful in order to manage the potential of stale data with uneven arrival rates.

The Access Module can be controlled in such a way that the TPump instance can be ended if no messages arrive on the queue for a specified amount of time. This will flush all rows to the database that may be sitting in the TPump buffers and enable any post-job processing. It also offers the opportunity of adjusting the pack factor or number of sessions on the next TPump job. In TUF 6.1 there will be a new TPump feature called LATENCY, which will partially alleviate the need for access module logic to force EOJ for buffer flushing.

Journaling

Both MSMQ Access Module and the MQSeries Access Module used in this prototype support the journaling of all messages read from the queue. The journaling functionality is optional in this Access Module. The prototype offers the opportunity to name a journal file as an optional environmental variable. If the journal file is named, the Access Module will append to this file each record read from the queue. Since this journaling activity is taking place on the client, not on the server, no performance impact on database throughput should be expected.

Journaling the input could be very useful for error row re-processing or for backup purposes when a queue is used as the source of data. In this prototype, the Access Modules read the queue destructively. Once a record passes through the queue and is read, it is no longer available. This is in contrast to flat file data, which can be saved and held on to after initial processing. Since there is no queue-processed data that can be saved in order to reconstruct error rows, or to use to restart the job, this journal file becomes more compelling when queues are used.

Checkpointing

Access modules in this prototype support TPump's Checkpoint-Restart processing to avoid loss or duplication of data due to processing failure. Input records are preserved in a disk file up until the time of a successful checkpoint. At checkpoint restart, the Access Module first reads from the checkpoint file, then resumes reading the queue.

This checkpoint file never gets larger than the number of input rows between checkpoints, as rows are removed once the checkpoint interval they are a part of is past. At TPump termination the checkpoint file is deleted in its entirety.

Similar to the journal file, the checkpoint file name is a run time input parameter which is used by the Access Module. If a checkpoint file name is not found during TPump intialization, the check-pointing is not performed.

Robust ON

ROBUST ON is the default for all TPump jobs. By inserting some extra information about the rows just processed into the database log table at the completion of each buffer's request, TPump has a way to avoid re-applying rows that have already been processed in the event of a restart.

What ROBUST ON does

This variable causes a row to be written to the log table each time a buffer has successfully completed its updates. These mini-checkpoint are deleted from the log when a checkpoint is taken, and are used during a restart to identify which rows have already been successfully processed since the most recent checkpoint was taken, then bypassing them on a restart.

ROBUST ON is particularly important if re-applying rows after a restart would cause either a data integrity problem or have an unacceptable performance impact. ROBUST ON is advisable for these specific conditions:

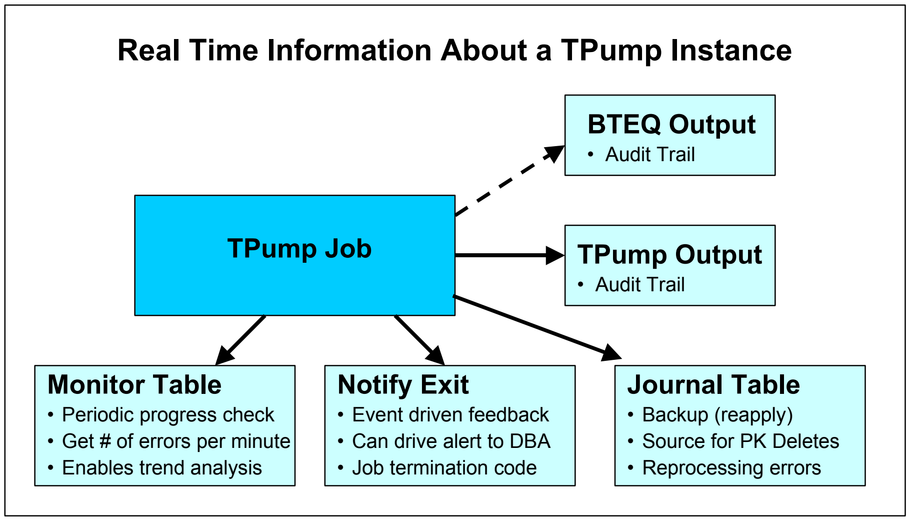
1. Essential for INSERTs into multi-set tables, as such tables will accept re-inserted rows
2. Essential when updates are based on calculations or percentage increases

3. If pack factors are large, and applying and rejecting duplicates after a restart would be unduly time-consuming
4. If data is time-stamped at the time it inserted into the database (see below)

In this continuous TPump prototype, input arrives via a queue and the date and time it is inserted into the database is recorded on each row. ROBUST ON is always a good idea for TPump jobs that read from queues, but is particularly important if timestamps are used in this manner. The original rows that were inserted before the restart will carry a timestamp that reflects their insertion time. When the same row is re-applied, it will carry a timestamp that is different, even though all of the other data is identical. To Teradata, this will appear as a different row with the same primary index value, so duplicate row checking will not prevent the duplicate insertion. ROBUST ON is needed to keep duplicates from being added to the table being loaded in the case of restart.

Consolidating Statistics

Equally important as managing the errors in a continuous environment is the handling of statistics and the real-time availability of information about job status. In a simple, stand-alone TPump job, statistics that show the success of the load being performed are often difficult to get until the completion of the job. This prototype offers an automatic and expansive level of information about the load, both while the load is happening and after.



Above is a picture that summarizes the various sources of statistical information that is available in real time and near real time from the continuous TPump executions.

Notify User Exit

TPump comes with a user exit referred to as the Notify Exit, which allows you to capture important statistics about significant events during your TPump job. Using the Notify Exit will give you greater confidence that what was sent to the database was actually committed.

The Notify Exit was not created just for the sake of this prototype, but was exploited for its usefulness in the context of continuous TPump rotations. If the Notify Exit is named explicitly in the TPump script, then TPump will load the exit code at job initialization and the exit will be called each time an event takes place.

What type of Events Engage the Notify Exit?
Some of the events that cause TPump's Notify Exit to be called include: Import start and end, Teradata errors, CLI errors, begin or end checkpoint, and end of job.

In this prototype, the Notify Exit was implemented in a platform-independent way, using C. While it was executed on Windows, in the future the same exit is planned for use on UNIX and the mainframe. The exit code is attached in Appendix A and may be copied directly to another platform and used as is, if the same functionality is useful.

Exits such as this are easy to extend. This particular exit writes its output to a file on the client, but could be easily modified to queue into a monitoring program, insert rows into a database, or send data to an alert manager program.

A sample of the printed output from the prototype's Notify Exit follows. The final entry includes such useful information as elapsed time for the import phase, total records read, user name, name of the error table, job type, effected database name, and more.

```
13:35:21 TPump - Init: Version=Teradata Parallel Data Pump 01.03.01, Username=rmh.
13:35:25 TPump - ImportBegin: ImportNumber=1.
13:35:26 TPump - FileOpen: File/Inmod Name=QUEUE1, ImportNumber=1.
13:41:21 TPump - CheckPoint Begin: Records=150000.
13:41:21 TPump - CheckPoint End: Records=150000.
13:41:21 TPump - ImportEnd: ImportNumber=1, Records Read=150000, Skipped=0, Rejected=0,
Sent=150000, Errors=1216.
13:41:21 TPump - TableStats: jobtype=I, Activity==148784,
TableName=continuous,DataBaseName=rms_per2
13:41:22 TPump - ErrTableDrop: Name=continuous.et1_rmh, Rows=1216.
13:41:25 TPump - Exit: TermCode=0.
13:41:25 TPump - Exit: FinalStatsHeader=init, begin(interval), open(interval), end(interval),cpuk, cpuu,
version, utilityname, username, errtbl,tablename, jobtype, dbasename, filename,activity, recs_in,
recs_skipped, recs_rejected,recs_out, recs_error, termcode.
2001-09-03 13:35:21.859 | 2001-09-03 13:35:25.437 | 0: 3.578 | 2001-09-03 13:35:26.468 | 0:01.031 |
2001-09-03 13:41:21.890 | 5:55.422 | 070.593 | 037.406 | Teradata Parallel Data Pump 01.03.01 | rmh |
continuous.et1_rmh | rms_per2 | I | continuous | QUEUE1 | 148784 | 150000 | 0 | 0 | 150000 | 1216 | 0
```

TPump Monitor Table

TPump provides a method of monitoring its own progress, which is expanded in this prototype. Only one row in the database is used by TPump for this purpose. If a specially-named monitor table has been provided in the database, and TPump can locate it during initialization, TPump will use this table at run time to both read from and write to. This table must be named **SysAdmin.TPumpStatusTbl**, and it can be created via a script provided in the TPump installation. If this monitor table is present, TPump inserts a row, then updates that same row once every minute, overlaying with each write the information it previously placed there.

What Kind of Data is Recorded in the Monitor Table?
A sampling of the columns include: Job start time, Last update time, Number of statements run, Records read, Records sent to the database, Records skipped, Records rejected.

By relying on database triggers, this prototype moves all information TPump writes to its monitor table into a history table which is also located in the Teradata database. This history table has been created specifically to hold images taken from the monitor table row. These database triggers are fired only when the single row in the TPumpStatusTbl changes, either because it is inserted (as it would be at job start), modified (once every minute during the job) or deleted (at end of job).

A subset of the collected contents of TPump's monitor table are shown below, as output from a query. The query ran against the history table. Each detail line represents the contents that were moved out of the monitor table at the time TPump updated it with new information. There are more columns (such as the log table name) than are displayed here.

User Name	Init StartDate	Init StartTime	Last UpdateDate	Last UpdateTime	Restart Count	Complete	Records Read	Records Errored
RMH	3/20/01	163613	3/20/01	163613	0	N	0	0
RMH	3/20/01	163613	3/20/01	163733	0	N	7	0
RMH	3/20/01	163613	3/20/01	163736	0	N	8	0
RMH	3/20/01	163613	3/20/01	163827	0	Y	8	0
RMH	3/20/01	165658	3/20/01	165658	0	N	0	0
RMH	3/20/01	165658	3/20/01	165658	2	N	0	0
RMH	3/20/01	165658	3/20/01	170924	1	N	0	0
RMH	3/20/01	165658	3/20/01	171226	1	N	4	0
RMH	3/20/01	165658	3/20/01	171230	1	N	4	0

Triggers

Teradata triggers were employed to automate the movement of data into a second location where it can be kept as long as it is useful. Without repetitively saving the information of the single row in the monitor table, it would be lost the next time the row is modified by TPump.

A mirror image of the status table was created as the repository of this statistical information. It has been named TpumpStatusLog and its layout is shown in the Appendix of this paper. A description of the three triggers defined for this prototype follow:

- TPUMPSTATUSINPUT trigger inserts a row into the TpumpStatusLog after the row is inserted into TPumpStatusTbl. The same data is copied with the column "ind" set to "i" for insert.

- TPUMPSTATUSUPDATE trigger inserts a row into the TpumpStatusLog after the row is updated on TPumpStatusTbl. The updated data is copied with the column "ind" set to "u" for update.

- TPUMPSTATUSDELETE trigger inserts a row into the TpumpStatusLog table before the row is deleted off of TPumpStatusTbl. The data is copied with the column "ind" set to "d" for delete.

All CREATE TRIGGER statements used in this prototype appear in the Appendix.

The extensive information being held in the TPumpStatusLog table allows interesting detail about TPump jobs to be saved and analyzed. The TPump job name could be added to the log table, so that each image of the monitor table that is captured can be identified with a particular job. Even without carrying the TPump job name, the job start time column can also be used to track to a particular TPump job, as TPump jobs in this prototype are named based on job start-date and start-time.

Other Uses of the Monitor Table Saved Output

The information triggered from the monitor table into a history table in the database offers some additional opportunities in a production environment. Since in a continuous TPump environment there is no end point, there is no one time at which it makes sense to review performance levels (such as at the end of each batch run). Being able to automatically alert the DBA when something unexpected is happening with the continuous TPump execution becomes more useful when a standard time for pausing and evaluating disappears, as batch windows shrink.

Alerts to the DBA could be useful when incidents such as these are noted, based on the historical statistical data:

- Two successive entries into the monitor table show no progress is being made
- A non-zero error count
- The row-processing rate is below a given threshold of acceptability

This prototype provides the foundation that underlies an alert system, which could be easily interface to several different third party monitoring and alert system tools.

Timestamps

As mentioned earlier, a timestamp is placed on each input record by the Sourcing Application when that row is placed in the queue. A second timestamp column is filled in when each row is actually inserted by TPump into the database. That column is defined in the definition of the table being loaded as:

```
db_time TIMESTAMP(6) DEFAULT CURRENT_TIMESTAMP(6)
```

Having these two timestamps on the database record itself allows latency comparisons to be made, and themselves are an additional type of statistical information that this prototype makes available. For example, by subtracting the queue timestamp from the database timestamp, you can get the amount of time that has elapsed between entry into the queue and insertion into the database.

Timestamps can also show you the TPump throughput rates as they have changed over time, or even during a particular day. This can be done by looking at how many rows in the base table have a database timestamp that falls within a given time interval. Also, by issuing an SQL statement that orders TPump-inserted base table rows by the second timestamp (time entered into the database), you can seek out specific periods of time when there were delays in your load process.

While analyzing timestamps, an anomaly appeared. After comparing the latency of all the rows loaded by a continuous TPump execution, it was noted that rows had a latency of either 1 or 2 seconds, or 19 or 20 seconds.

The latency was measured as the difference between the time inserted in the database and the time the row entered the queue, as represented by the two timestamps. Of the total rows loaded, about half took 19-20 seconds to be inserted and half took 1-2 seconds, as illustrated on the following chart.

Max – Min	Count(*)
1	3911
2	1510
19	2607
20	2821

In examining the time setting on each of the two nodes in the test configuration, an 18-second difference in clock time was discovered. Because an MPP system is being used as the server, tight synchronization of clocks on all nodes needs to be in place. A UNIX utility was run on this two-node prototype system that fixed the problem. Any production application depending on timestamps in such a way needs to validate that the time on one node is the same as the time on all nodes.

There is an additional benefit of these timestamps used in this prototype that could be relevant in a production environment. One side-effect of continuously loading data is that it may raise issues for users doing analysis if they require that their data be consistent to a certain point in time. Some applications need all the data they are operating on to appear as it did at 8 AM or 12 midnight. To manage data freshness in your database you may find it useful to rely on point-in-time views, which use this second timestamp (the time the row was inserted into the database) to filter out transactions that do not fall within the data freshness parameter.

Analyzing Prototype Latency

Included in this section are two snapshots taken during a series of rotating TPump jobs (after nodes were synchronized by time). The first illustrates the latency of rows moving from the queue into Teradata when there are 1% errors in the input file. The second shows a sample of the latency measurements with clean input data. The columns (including one row of data) in each snapshot look like this:

	# Rows			*** - - - - - - - - -**	**- - - For This**	**Minute - - -**	**- - - - - - - - ***
	Inserted	**Insertion**	**Cumulative**	**Min**	**Max**	**Average**	**Standard**
Minute	**this Minute**	**Rate/Sec**	**# of Rows**	**Latency**	**Latency**	**Latency**	**Deviation**
1	13,028	217	13,028	0.001	10.633	2.431	1.675

For each minute in the snapshot, the number of rows inserted is totaled, and an insertion rate is calculated, and a cumulative sum is given. Within that minute, the minimum, maximum, and average latency are reported, as well as the standard deviation of the latencies of all rows within that minute interval.

In both snapshots "relative" latency has been calculated, rather than reporting the exact turnaround times. This is because the clock on the client and the clock on the server were slightly out-of-sync (7 to 8 seconds difference between them). In the real world, multiple clients feeding the queue may have clocks that are not perfectly synchronized. Because latency measurements depend on subtracting the timestamp made when a row enters the queue from the timestamp when that same row is inserted into the database, this difference in clock times distorts actual latency values.

Relative latency was determined like this: For each run, the row carrying the minimum latency was identified. This value was re-interpreted as zero, and became the baseline for all other latencies within the job. All other latencies are reported relative to this minimum value of zero. The minimum latency value is subtracted from each reported latency to maintain their relative differences. By doing this the degree of variation across all rows is preserved, even though the clocks are slightly off.

The following chart represents 14 minutes taken from a TPump job with 1% errors, a pack factor of 10, using 20 sessions:

				* - - - - - - - - -	- - - For This	Minute - - -	- - - - - - - - *
Minute	**# Rows Inserted this Minute**	**Insertion Rate/Sec**	**Cumulative # of Rows**	**Min Latency**	**Max Latency**	**Average Latency**	**Standard Deviation**
1	13,028	217	13,028	0.001	10.633	2.431	1.675
2	13,074	217	26,102	0.001	8.943	2.472	1.569
3	13,127	218	39,229	0.004	7.413	2.741	1.450
4	13,093	218	52,322	0.004	7.861	2.641	1.415
5	13,058	217	65,380	0.003	9.119	2.644	1.468
6	12,937	215	78,317	0.003	9.690	2.601	1.411
7	13,076	217	91,393	0.002	6.864	2.604	1.427
8	13,065	217	104,458	0.001	8.250	2.519	1.487
9	13,152	219	117,610	0.002	7.691	2.566	1.626
10	12,937	215	130,547	0.001	8.960	2.613	1.615
11	13,042	217	143,589	0	9.931	2.623	1.510
12	12,910	215	156,499	0.001	7.528	2.523	1.468
13	12,854	214	169,353	0.001	7.139	2.659	1.634
14	13,113	218	182,466	0.003	8.373	2.685	1.330

The Sourcing Application was the bottleneck in this prototype. The average relative latency (latency adjusted for the differences between the client and server clocks) across all minute intervals is about 2.6 seconds. In reality, average latency is probably 2.8 or 2.9 seconds.

The standard deviation column represents how frequently the latencies within each minute interval appear above or below the average latency. A standard deviation of around 1.5 shows a good clustering around the mid-point and translates to fairly stable latencies.

The following chart represents 17 minutes taken from a TPump job with no errors, a pack factor of 10, using 20 sessions. This TPump job ran for 40 minutes, but this snapshot captures 17 minutes in the middle of the job.

	# Rows			*- - - - - - - For This	Minute - -	- - - - - - - - *	
	Inserted	Insertion	Cumulative	Min	Max	Average	Standard
Minute	this Minute	Rate/Sec	# of Rows	Latency	Latency	Latency	Deviation
1	13,050	217	13,050	0.005	2.959	1.803	0.631
2	13,075	217	26,125	0.006	2.846	1.733	0.658
3	13,100	218	39,225	0.009	2.772	1.701	0.648
4	13,025	217	52,250	0.007	2.781	1.673	0.684
5	12,725	212	64,975	0.005	3.132	1.637	0.727
6	13,050	217	78,025	0.002	2.603	1.582	0.714
7	13,025	217	91,050	0.002	2.473	1.527	0.728
8	13,125	218	104,175	0.016	2.382	1.485	0.717
9	13,075	217	117,250	0.014	2.239	1.468	0.714
10	13,075	217	130,325	0.007	2.272	1.465	0.707
11	13,050	217	143,375	0.007	2.247	1.458	0.701
12	13,125	218	156,500	0.007	2.274	1.511	0.732
13	13,175	219	169,675	0.002	2.247	1.514	0.733
14	13,225	220	182,900	0.002	2.247	1.488	0.723
15	13,150	219	196,050	0	2.335	1.465	0.722
16	13,050	217	209,100	0	2.157	1.444	0.725
17	13,125	218	222,225	0.002	2.635	1.528	0.740

In this case, the standard deviation shows an even tighter clustering around the average than in the case with 1% errors. This is what you would comparing a TPump instance with errors against one with clean data. The TPump instance with errors (reflected on the previous chart) will have some rows that have to be rolled back and inserted again because those buffers contain errors. Those rows that fall victim to error processing (are in the same buffer as the error row) will have a longer latency, maybe twice as long. These error rows will raise the average latency in the minute interval in which they occur.

Range of Seconds	1 % ERRORS Count(*)	CLEAN DATA Count(*)
0-1	30,533	49,207
1-2	41,617	120,258
2-3	39,782	65,510
3-4	42,440	25
4-5	23,615	
5-6	7,742	
6-7	1852	
7-8	785	
8-9	309	
8-10	70	

10-11	28	

In this TPump job with no errors, the average latency (the time it takes for a row to enter the queue and then be inserted into the database) is 1.5 seconds. The much smaller variance in latencies can be also seen in the chart above, which documents the number of rows that fall into each range of latencies. Close to 75% of the rows pass from the queue into Teradata in two seconds or less, when data is clean.

The table above shows how many rows in this snapshot were inserted within each of several different ranges of seconds, for both the 1% error execution and the execution with clean data. For example 30,533 rows were inserted with a relative latency of less than 1 second in the 1% error case. The variance in latency is significantly less with clean data.

Conclusions

Loading Teradata via queues in not futuristic. Using Access Modules, any of the three Teradata load utilities can draw information from a queue. And any of them can be executed in frequent batch modes, or continuous modes, as this TPump prototype has illustrated.

Several useful opportunities have been validated by this prototype. First, streaming input data into a load utility, without the need to pre-stage the data in flat files, was proven feasible. Second, the goal of data freshness in the warehouse was achieved with the prototype showing near real-time loads with less than 5 seconds from the time the record entered the queue until it was inserted into the database. Thirdly, operational simplicity and flexibility was established. Rotating instances of the load job allowed for discrete moments in time to handle errors, adjust parameters, and validate progress. And finally the ability to handle 7 x 24 loads was established. Not only can loading continue throughout the day, journaling the input records as was done in the prototype, illustrated a mechanism to archive and even re-load the data, if needed.

In addition, this entire prototype represents a starting point for more sophisticated combinations of functionality, customized around a users particular loading needs.

As more Teradata users explore ways to make their just-generated data available more quickly, a continuous load strategy fed by queues will become a more compelling and an extensible solution. This prototype suggests one approach to using a queue with TPump, a strategy consistent with breaking through the batch window barrier.

Appendix:

This appendix includes the syntax for the following:
1. Triggers for saving statistics history
2. Log table layout for holding statistics history

In addition a zip file is attached which contains the following code:
1. Visual Basic code for the Sourcing Application and the Rotater Application
2. Access Module C code for MSMQ and for MQSeries
3. Notify Exit code

Trigger Creation Syntax

```
REPLACE TRIGGER tpcd50g.TpumpStatusInsert
ENABLED AFTER INSERT
ON sysadmin.TPumpStatusTbl
REFERENCING NEW AS NewRow
FOR EACH ROW
  ( INSERT INTO tpcd50g.TPumpStatusLog
    VALUES (NewRow.LogDB,
              NewRow.LogTable,
              NewRow.Import,
              NewRow.UserName,
              NewRow.InitStartDate,
              NewRow.InitStartTime,
              NewRow.CurrStartDate,
              NewRow.CurrStartTime,
              NewRow.LastUpdateDate,
              NewRow.LastUpdateTime,
              NewRow.RestartCount ,
              NewRow.Complete,
              NewRow.StmtsLast,
              NewRow.RecordsOut,
              NewRow.RecordsSkipped ,
              NewRow.RecordsRejcted,
              NewRow.RecordsRead,
              NewRow.RecordsErrored,
              NewRow.StmtsUnLimited,
              NewRow.StmtsDesired,
              NewRow.SessnDesired,
              NewRow.TasksDesired,
              NewRow.PleasePause,
              NewRow.PleaseAbort,
              NewRow.RequestChange,
              NewRow.LogonSource,
              NewRow.LoadPID,
              NewRow.NumChildren,
              NewRow.ChildrenList,
              NewRow.ParentNode,
              'i');
  );
```

```
REPLACE TRIGGER tpcd50g.TpumpStatusUpdate
ENABLED AFTER UPDATE
ON sysadmin.TPumpStatusTbl
REFERENCING NEW AS NewRow
FOR EACH ROW
  (
    INSERT INTO tpcd50g.TPumpStatusLog
    VALUES (NewRow.LogDB,
              NewRow.LogTable,
              NewRow.Import,
              NewRow.UserName,
              NewRow.InitStartDate,
              NewRow.InitStartTime,
              NewRow.CurrStartDate,
              NewRow.CurrStartTime,
              NewRow.LastUpdateDate,
              NewRow.LastUpdateTime,
              NewRow.RestartCount ,
              NewRow.Complete,
              NewRow.StmtsLast,
              NewRow.RecordsOut,
              NewRow.RecordsSkipped ,
              NewRow.RecordsRejcted,
              NewRow.RecordsRead,
              NewRow.RecordsErrored,
              NewRow.StmtsUnLimited,
              NewRow.StmtsDesired,
              NewRow.SessnDesired,
              NewRow.TasksDesired,
              NewRow.PleasePause,
              NewRow.PleaseAbort,
              NewRow.RequestChange,
              NewRow.LogonSource,
              NewRow.LoadPID,
              NewRow.NumChildren,
              NewRow.ChildrenList,
              NewRow.ParentNode,
              'u');
  );

REPLACE TRIGGER tpcd50g.TpumpStatusdelete
ENABLED BEFORE DELETE
ON sysadmin.TPumpStatusTbl
REFERENCING old AS oldrow
FOR EACH ROW
  (
    INSERT INTO tpcd50g.TPumpStatusLog
    VALUES (OldRow.LogDB,
              OldRow.LogTable,
              OldRow.Import,
              OldRow.UserName,
              OldRow.InitStartDate,
              OldRow.InitStartTime,
              OldRow.CurrStartDate,
              OldRow.CurrStartTime,
                          OldRow.LastUpdateDate,
                          OldRow.LastUpdateTime,
                          OldRow.RestartCount ,
                          OldRow.Complete,
                          OldRow.StmtsLast,
                         OldRow.RecordsOut,
                         OldRow.RecordsSkipped ,
                         OldRow.RecordsRejcted,
                         OldRow.RecordsRead,
                         OldRow.RecordsErrored,
                         OldRow.StmtsUnLimited,
                         OldRow.StmtsDesired,
                         OldRow.SessnDesired,
                         OldRow.TasksDesired,
                         OldRow.PleasePause,
                         OldRow.PleaseAbort,
                         OldRow.RequestChange,
                         OldRow.LogonSource,
                         OldRow.LoadPID,
                         OldRow.NumChildren,
                         OldRow.ChildrenList,
                         OldRow.ParentNode,
                          'd');
       );
```

Because triggers were built on a SysAdmin table (TPumpStatusTbl) and because TPumpMacro (the macro the TPump executes that causes this trigger to fire) is the user, grant access with a grant option to TPumpMacro is needed for the table. The following command was issued:

```
GRANT ALL ON tpcd50g TO TPumpMacro WITH GRANT OPTION;
```

Status Log Table Layout

```
CREATE MULTISET TABLE tpcd50g.tpumpstatuslog ,NO FALLBACK ,
   NO BEFORE JOURNAL,
   NO AFTER JOURNAL
   (
    LogDB VARCHAR(32) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
    LogTable VARCHAR(32) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
    Import INTEGER NOT NULL,
    UserName VARCHAR(32) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
    InitStartDate DATE FORMAT 'YY/MM/DD' DEFAULT DATE ,
    InitStartTime FLOAT DEFAULT TIME ,
    CurrStartDate DATE FORMAT 'YY/MM/DD' DEFAULT DATE ,
    CurrStartTime FLOAT DEFAULT TIME ,
    LastUpdateDate DATE FORMAT 'YY/MM/DD' DEFAULT DATE ,
    LastUpdateTime FLOAT DEFAULT TIME ,
    RestartCount INTEGER NOT NULL DEFAULT 0 ,
    Complete CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC DEFAULT 'N',
    StmtsLast INTEGER DEFAULT 0 ,
    RecordsOut INTEGER DEFAULT 0 ,
    RecordsSkipped INTEGER DEFAULT 0 ,
    RecordsRejcted INTEGER DEFAULT 0 ,
    RecordsRead INTEGER DEFAULT 0 ,
    RecordsErrored INTEGER DEFAULT 0 ,
    StmtsUnLimited CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL DEFAULT 'Y',
    StmtsDesired INTEGER DEFAULT 0 ,
    SessnDesired INTEGER DEFAULT 0 ,
    TasksDesired INTEGER DEFAULT 0 ,
    PleasePause CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC DEFAULT 'N',
    PleaseAbort CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC DEFAULT 'N',
    RequestChange CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL DEFAULT 'N',
    LogonSource VARCHAR(128) CHARACTER SET LATIN NOT CASESPECIFIC,
    LoadPID CHAR(12) CHARACTER SET LATIN NOT CASESPECIFIC,
    NumChildren SMALLINT,
    ChildrenList CHAR(8192) CHARACTER SET LATIN NOT CASESPECIFIC,
    ParentNode CHAR(64) CHARACTER SET LATIN NOT CASESPECIFIC,
    ind CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC)
PRIMARY INDEX ( LogDB ,LogTable ,Import );
```