1 Loading Data

In a data warehouse environment, there's often a need to load massive amounts of data, usually at regular intervals (daily, weekly, monthly, etc.). These notes discuss some of the options and automation hints.

1.1 `INSERT` statements

An obvious way to load data is using `insert` statements. Obviously we can open a connection to the database, and simply dump many SQL statements into it.

The primarly limitation is usually speed. Every statement takes a certain amount of overhead to process (in many cases, the database will start a statement level transaction, etc.). There's also the cost associated with readin gand parsing the SQL input. Multiply that overhead many millions of times, and it becomes obvious why for large data loads, we should avoid SQL.

Another issue is that often, data exists in some funny database unfriendly format, such as CSV, or fixed width, or in some weird encoding (such as EBCDIC). Some tool is either used or created to turn that data into database friendly insert statements. An example may be something as simple as (`csv2sql.pl`):

```
while(chomp($_ = <>)){
    my @r = split/,/;
    print "INSERT INTO $ARGV[0] VALUES (".
        join(",", map { "'$_'" } @r).
    ");\n";
}
```

With the above style script, you can pipe CSV data in, and get SQL insert statements out. If for example your data is in `xdata.csv.gz` file, you can run:

```
zcat xdata.csv.gz | perl csv2sql.pl table1 > input.sql
```

If database supports piped in sql (as most do) you can generally just pipe that into the database directly, ie:

```
zcat xdata.csv.gz | perl csv2sql.pl table1 | dwsqltool ...
```

The above approach is slow, but it does work for all databases that claim to be SQL compatible. Some databases aren't designed for such inserts though, and it's not uncommon to find specialized data warehouse databases that take a *long* time to process each insert statement (something on the order of 1 second to do an insert statement—try inserting a million records like that).

1.2 Database Tools

1.3 MySQL's `mysqlimport` & `LOAD DATA INFILE`

MySQL comes with a command line utility `mysqlimport`. Internally, this command line works exactly like the "LOAD DATA INFILE" command, which is available through the `mysql` client.

The `mysqlimport` utility takes a huge number of options, specifying among other thing, the field delimiter, record terminator character, list of columns, etc.

```
mysqlimport [options] tname filename.csv
```

The `LOAD DATA INFILE` is somewhat similar to command line, but works from mysql client:

```
LOAD DATA INFILE 'filename.csv' INTO TABLE tname ...
```

(you can specify many options, etc.)

1.4 Oracle's `sqlldr`

Oracle's scheme to load data revolves around two files: the control file, and a parameter file. The control file specifies table name, options, column list, as well as data types and formats. The parameter file specifies the locations of log, bad, etc., files, as well as database connectivity information.

A control file looks like this:

```
LOAD DATA INFILE 'filename.csv'
APPEND
INTO TABLE tname
FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(   FIELD1  POSITION(1) CHAR TERMINATED BY ",",
    FIELD2  CHAR,
    FIELD3  DECIMAL EXTERNAL,
    FIELD4  INTEGER EXTERNAL,
    FIELD5  DATE "YYYYMMDD",
)
```

A parameter file may look like this:

```
userid = username/password
control = tname.ctl
log = tname.log
bad = tname.bad
direct = true
silent = (discards, feedback)
errors = 20
```

You run these things with:

```
sqlldr parfile=tname.par
```

1.5 PostreSQL `copy`

From within `psql` utility, you can run the ‘copy’ command:

```
COPY tname TO 'filname.csv' USING DELIMITERS ',';
```

```
COPY tname FROM 'filename.csv' USING DELIMITERS ',';
```

1.6 MSSQL `bcp` (Bulk Copy Program) & BULK INSERT

BCP is command line utility that comes with SQL Server:

```
BCP dbname..tname IN filename.csv -c -t ',' -r \n -U user -P pass
```

(obviously you’d need to specify login information)

```
BCP dbname..tname OUT filename.csv -c -t ',' -r \n -U user -P pass
```

BULK INSERT is a query, that is used to tell SQL Server to load data form a file, ie:

```
BULK INSERT tname FROM 'filename.csv'
WITH (
    DATAFILETYPE = 'char',
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '\n'
);
```

1.7 Netezza `nzload`

Netezza comes with `nzload` command line utility. You pipe .csv (or other delimited) data into the command.

1.8 Fixed width data

Once in a while you might need to work with fixed width data. That’s data usually coming from old software (like a cobol program outout), etc.

Many bulk database loaders aren’t very flexible when it comes to fixed width data, with the exception being Oracle’s `sqlldr`.

If the data volumes aren’t outragious, you might try the Perl approach, and get to enjoy the ‘pack’ and ‘unpack’ function.

If data volumes are huge, I suggest you write a C/C++ program to do the conversion (especially if you’re working with diffrent character sets, such as EBCDIC being converted to ASCII, etc.)

1.9 Piped Loads

In data warehouse environments, data files may often be too big to uncompress. Also, working with uncompressed files puts a lot of strain on the disk IO. By always keeping data in compressed form, you can offload some of the disk IO over to CPU (which is generally much faster).

With most databases, you can use a pipe to load data. We can create a ‘pipe’ file:

```
mknod /tmp/tname.pipe p
```

We can then specify this pipe object instead of the `.csv` file in any of the above database bulk loading tools, such as,

```
LOAD DATA INFILE '/tmp/tname.pipe'
APPEND
INTO TABLE tname
```

Then (in a separate process), you setup a write to the pipe, such as:

```
zcat myfile.csv.gz > /tmp/tname.pipe
```

Then you start the database bulk loading tool, and it will happily read from `/tmp/tname.pipe` as if it was a plain regular `.csv` file.

1.10 Parallel Loads

Most databases support parallel loads—which actually does often speed up data load times. For example, if your load is IO bound, you can load data from different drives. If your load is CPU bound, you can use multiple processors.

To setup multiple loads, simply start the loads in separate processes. That’s it. Setting up a Perl script to kick off multiple loads whenever files appear in some directory is usually best:

```
# go through all .complete files in upload dir, and
# start a job for each one.
$maxProcs=32;
$numProcs=0;
for my $file (glob "$uploaddir/*.complete"){
    $file =~ s/.*?([^\/]+)\.complete$/$1/i;
    while($numProcs >= $maxProcs){
        $numProcs-- if wait();
        if(fork()){
            $numProcs++;
        }else{
            loadFile($file);
            exit;
        }
    }
}
1 while(wait() > 0);
exit;
```

The `loadFile($file)` would do the obvious thing of invoking a bulk database loading tool on the filename. the `$maxProcs` can be adjusted, and should be about the number of CPUs in your server.

1.11 FTP Thingies

The above section hints at something really neat. Imagine a situation where you manage a data warehouse, and every night someone deposits (or you need to download) some data files, and load them into the database.

One approach is to setup an 'upload' folder, where everyone dumps their files. The files would have some identifying name, along with a '.complete' file (the .complete file is tiny file, used only as a flag that the parent file has finished uploading).

Lets say your upload folder is: `/uploads`. You have a cron script look through it every minute, if it finds any .complete files, it loads the data into the database (possibly using the parallel loader discussed above).

The sending party will first upload:

```
datafile.20080101.csv.gz
```

then once this upload is complete, they'll upload

```
datafile.20080101.csv.gz.complete
```

This will ensure that your cron script picks it up and loads it into the database within the next minute.

As an alternative to the .complete file, you can use the update time. If the file hasn't been updated in 5 minutes (rough estimate) then you assume it's finished uploading, and you can start loading it. Perl has a very useful "-M" function that returns the file update time in days.

1.12 Robust Transfers & Loads

Every file transfer should have some indicator that the file is complete. You should have a record count on each file somewhere, for example, if datafile.csv has 14382347 recods, all for 20080101 date, have that data be part of the filename, ie:

```
datafile.20080101.14382347.csv.gz
```

Now anyone can verify the data simply by doing either

```
gzip --test
```

or

```
zcat datafile.20080101.14382347.csv.gz | wc -l
```

The record count should match the one on the file. If not, this file should be discarded, re-transmitted, reloaded, etc.

Every operation should be assumed to fail. If it really does fail re-try that operation. For example, if for whatever reason, the database load filed, restart the load 30 minutes later. If it fails then, restart it again 30 minutes later, etc., so on and so forth.

Do not rely on exact times to run scripts. ie: having a cron job that runs every night at 3AM is a very very bad idea. What if your data is late? What if someone is doing maintenance on something? What if network is down?

Setup your scripts to check for data availability and perform whatever function is needed. If they cannot peform their function, have them try it again and again and again and again until they succeed. That way, if someone unplugged your server for the night, or data upload took an unusually long time, the data 'job will get done automatically' as soon as the data becomes available (and you won't have to 'restart' things manually to ensure it gets done).

1.13 Templates

Bulk database loaders aren't exactly easy commands to work with. It takes a while to create the exact set of commands that will work—and then usually these need to be multiplied by the number of tables you have, and executed every day—with logs going to appropriate places, etc.

You should automate the job of managing these files to scripts. You should create a general purpose template, and then simply pass in different parameters for different databases, files, etc.

Consider this little Perl script:

```
my %args;
map {my($a,$b)=split/=/; $args{$a} = $b ? $b:1 } @ARGV;
local $/=undef;
$_ = <>;
s/<\?=(.*?)\?>/$args{$1}/sgi;
print $_;
\end{document}
You can use such a script to provide templating functionality to your database files, ie
\begin{verbatim}
LOAD DATA INFILE '<?=fname?>'
APPEND
INTO TABLE tname_<?=tdate?>
```

Which you'll turn into a 'control file to run today' by simply piping it through the template script, ie:

```
cat tname.ctl_template | perl t.pl tdate='20080101' fname=/tmp/tmp.pipe
```

You can go much farther and implement your own mini-scripting language, ie (s.pl):

```
{ local $/=undef; $_ = <> }
$_ = "?>$_<?perl";
```

```
s{<\?=(.*?)\?>}{<?perl print($1); ?>}sgi;
s{\?>(.*?)<\?perl}{
    my $a=$1; $a=~s/\\/\\\\/sgi;
    $a=~s/'/\\'/sgi; "print '$a';\n"}sgie;
eval($_);
```

And you can write scripts that look eerily similar to PHP, ie (sample.template):

```
...
<?perl
for my $i (1..4){
    ?><p>Hello World <?=$i?></p><?perl
}
?>
...
```

You can run it via:

```
cat sample.template | s.pl > output.txt
```

Ie: the whole point of these things being that you can and should automate just about everything that deals with data.