

# TPCx-V and TPCx-HCI User's Guide

Version 2.1.5, April, 2019

---

## [0.0 Two Benchmarks, One User's Guide](#)

### [1.0 Overview](#)

### [2.0 Setting Up the SUT Environment](#)

#### [2.0.1 Using the pre-configured OVF template](#)

#### [2.0.2 Run as root or postgres?](#)

### [2.1 Create the Virtual Machines](#)

#### [2.2 Install the Operating System](#)

#### [2.3 Configure the Operating System](#)

### [2.4 Install PostgreSQL and the Kit](#)

#### [2.4.1 Clone Virtual Machines](#)

### [2.5 Configure a Tier B Virtual Machine](#)

#### [2.5.1 Create Tablespace Directories](#)

#### [2.5.2 Setup PostgreSQL](#)

#### [2.5.3 Create the File System for the Flat Files](#)

#### [2.5.4 Create the Flat Files](#)

#### [2.5.5 Create the TPCx-V Database](#)

#### [2.5.6 Back up the TPCx-V Database](#)

#### [2.5.7 Restoring the TPCx-V Database](#)

### [2.6 Configure a Tier A Virtual Machine](#)

#### [2.6.1 Setup the ODBC Environment](#)

## [3.0 Setting Up the Driver Environment](#)

### [3.1 Building the Benchmark Driver](#)

### [3.2 Copying Benchmark Driver Files](#)

### [3.3 Editing the Configuration File](#)

## [4.0 Using the supplied bash scripts to run a test](#)

## [5.0 Manually starting a Test Run](#)

### [5.1 Starting the Component Processes](#)

#### [5.1.1 Starting the VConnector Process](#)

#### [5.1.2 Starting the VMee Process](#)

#### [5.1.3 Starting the VCe Process](#)

#### [5.1.4 Starting the VDM Process](#)

### [5.2 Starting the Prime Driver Process](#)

## [6.0 Post-Run Reporting](#)

[6.1 Mix Log File Sorting](#)

[6.2 Mix Log File Merging](#)

[6.3 Transaction Mix and Audit Check Log File Creation](#)

[6.4 Executive Summary Creation](#)

[6.5 Advanced Reporter Features](#)

[7.0 TPCx-V Auditing](#)

[8.0 Tips](#)

---

## 0.0 Two Benchmarks, One User's Guide

The two benchmarks TPCx-V and TPCx-HCI share the same Benchmark Kit, and are very similar in installation, set-up, and execution. The main differences between the two benchmarks are that TPCx-V is a single-system benchmark, but TPCx-HCI runs on multi-node HCI clusters, and there are significant differences in how the number of Tiles is calculated for them. Although the same Benchmark Kit may be used for both TPCx-HCI and TPCx-V benchmarks, the results of the TPCx-HCI and TPCx-V benchmarks may not be compared against each other.

**The benchmark name used throughout this Guide is TPCx-V, but the instructions apply equally to TPCx-V or TPCx-HCI.**

## 1.0 Overview

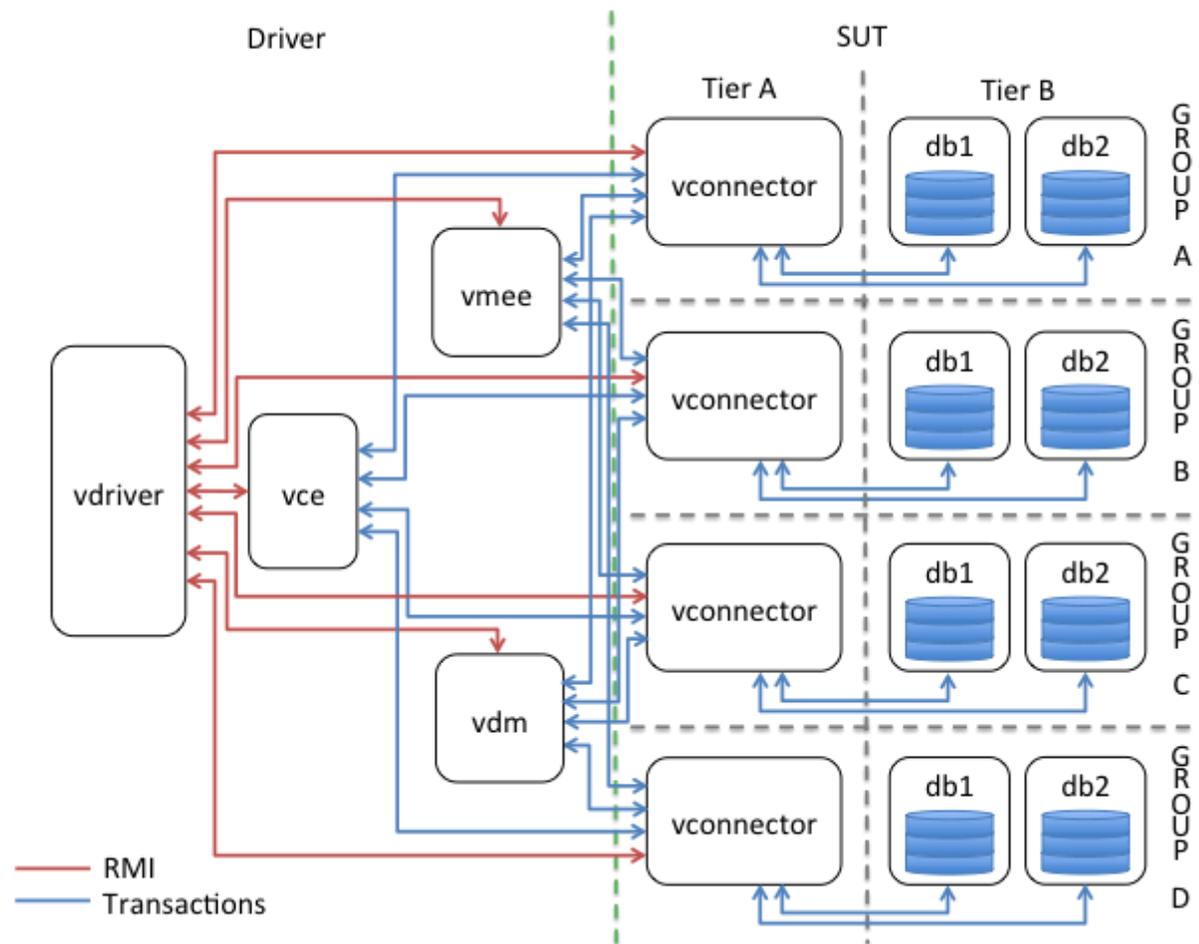
This document will help you set up one TPCx-V Tile using Virtual Machines running the following software configuration.

Operating System: RHEL 7.6

Database: PostgreSQL 10.5

**Other Linux distributions that meet the TPC Pricing Specification's support requirements are also acceptable, as well as other supported releases of PostgreSQL. In this Guide, we use RHEL 7.6 and PostgreSQL 10.5, which happens to be the latest release of PostgreSQL that Red Hat distributes as part of a supported *Software Collections*.**

The following figure represents all of the components of the benchmark driver and the corresponding network connections between them. This figure may be a useful visual reference as you work through these instructions. Note that this figure represents a System Under Test (SUT) with a single-tile test configuration using only a single driver. For each additional tile, an additional twelve SUT VMs, potentially additional vmee and vce processes, and corresponding connections to those SUT VMs and the vdriver prime driver would be required.

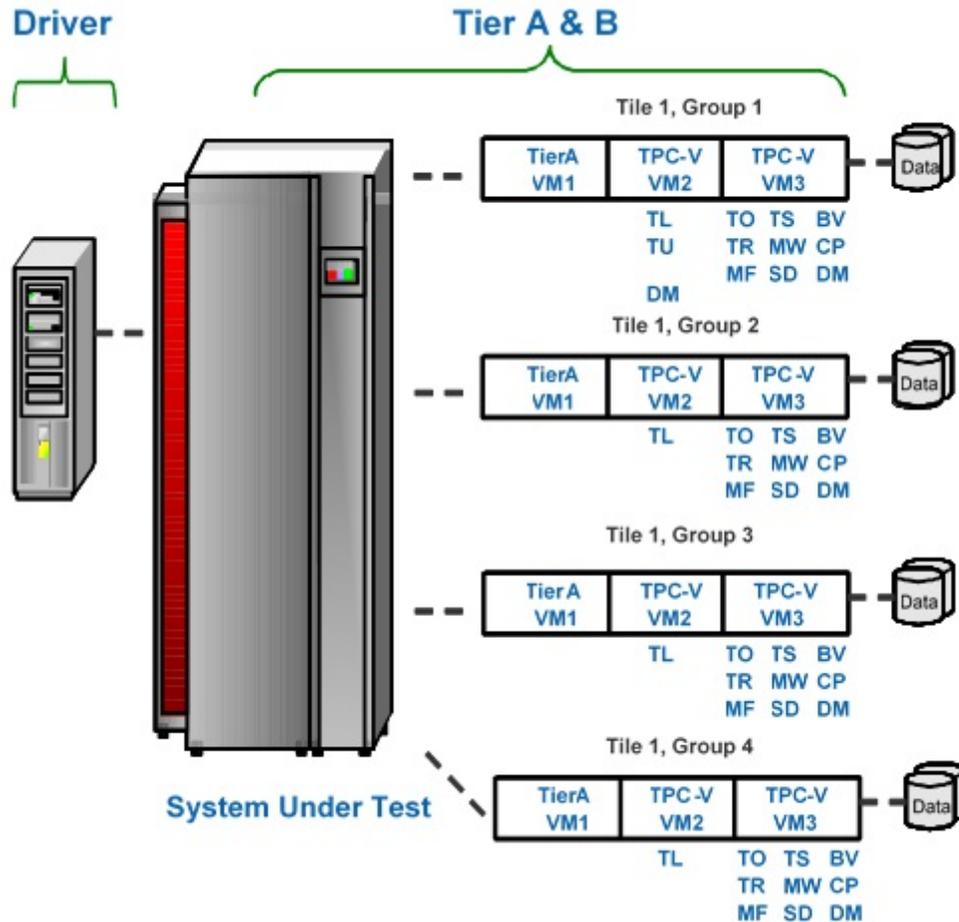


## 2.0 Setting up the SUT Environment

A general note: the TPCx-V benchmark started out life under the name "TPC-V", without the letter "x". The name of the database, names of some files in the benchmark structure, etc. still bear names such as tpcv, TPC-V, etc.

As you can see from the diagram in Section 1.0, a single-tile SUT configuration will consist of 12 different virtual machines. The instructions in this section will describe how to set up a single virtual machine. You can either run through the instructions 12 times to set up the various virtual machines or you can set up one Tier A virtual machine and one Tier B virtual machines using the instructions, clone those images as needed, and then make the necessary modifications to the cloned images to customize them for their purpose (unique hostname, unique ip address, secondary disk devices, etc).

The figure below show a sample 1-Tile SUT configuration, showing the 3 VMs per Tile, as well as the division of transaction types between the two Tier B VMs.



## 2.0.1 Using the OVF template

A preconfigured OVF template is available to download from the TPC website. This template has all the necessary software installed and configured. The root password of this VM is "tpc-v" (note: no letter "x" in the password). If you have downloaded and deployed this template, you can skip to section [2.4.1 Clone Virtual Machines](#)

## 2.0.2 Run as root or postgres?

Once the VMs are configured, most of the commands are run as user postgres. When we install the postgres software, we get a user postgres in /usr/passwords. But it does not have a password, and cannot be directly logged into. You can give it a password, or log into the VM as root, and then do a `su - postgres`

Unless otherwise noted, the commands in sections [2.1](#) to [2.5.3](#) are meant to be run as the user root.

## 2.1 Create the Virtual Machines

Create a virtual machine with the following minimum characteristics

- 2 virtual cpus
- 4GB memory
- 6GB primary disk device
- 1 network interface

### 2.1.1 Disk Space for Database Files

The 6GB of space in the root file system should be enough to install the operating system and all the software packages. Ordinarily, you should install all the software and finish configuring the first VM, then clone it into Tier A and Tier B VMs. At that point, you will need the extra space for the databases in the Tier B VMs.

The Tier B virtual machines will need additional disk storage for the database, backups, and flat files. The amount of storage needed will depend on the Group that will be running in the VM. The VM3 Tier B VMs will require slightly more space than VM2 Tier B VMs. Use the chart below to determine the extra disk space needed. The primary disk device of Tier B VMs can be sized to include enough space for the database, backup, and flat files (the flat files may be created on a single VM and nfs-mounted on all other VMs), or other storage devices can be added to provide the needed size.

**The values in the table below are for the minimum valid configuration of 50 LUs divided between the 4 Groups of one Tile (see Clause 2.4.1.3 of the benchmark specification). The valid increment size is 10 Load Units at a time. Each Load Unit will require approximately 16GB of disk space. So as an example, a 1-Tile, 800-Load Unit server will need around 12.5TB of *database* disk space with VM2 and VM3 of Group 4 each using around 2.5TB of disk space. The flat files will need an additional 1TB of disk space.**

**Even if you choose to not create a backup copy of the database, you should still allocate the space that backup would have used. That additional space in the file system gets used to hold the temporary data that is generated during the index creation phase of database load.**

**Storage requirements of the minimal 50-LU configuration**

	Number of Load Units	VM2 Database Size (GB)	VM2 Backup Size (GB)	VM3 Database Size (GB)	VM3 Backup Size (GB)	Flat Files (GB)
--	----------------------	------------------------	----------------------	------------------------	----------------------	-----------------

Group 1	5	37	7.4	38.5	7.7	15
Group 2	10	70.5	14.1	73.5	14.7	30
Group 3	15	105	21	110	22	45
Group 4	20	138	28	144	29	60
Total	50	860GB for all 8 VMs				60 (Total is equal to Group 4's)

### Storage requirements for a 10-LU increment

	Number of Load Units	VM2 Database Size (GB)	VM2 Backup Size (GB)	VM3 Database Size (GB)	VM3 Backup Size (GB)	Flat Files (GB)
Group 1	1	7.4	1.5	7.7	1.6	3
Group 2	2	14.1	2.8	14.9	3.0	6
Group 3	3	21	4.2	22	4.4	9
Group 4	4	27.6	5.5	28.8	5.8	12
Total	10	172GB for all 8 VMs				12 (Total is equal to Group 4's)

## 2.2 Install the Operating System

Install the 64-bit, **server** version of RHEL 7 or CentOS 7 in the virtual machine on the primary disk device of the VM. In the "INSTALLATION SUMMARY" window, click on "SOFTWARE SELECTION", and under "Base Environment", choose "Server with GUI". Under "Add-Ons for Selected Environment", choose the following:

1. File and Storage Server [to get NFS]
2. Network File System Client
3. Large Systems Performance

4. Performance Tools
5. Compatibility Libraries
6. Development Tools

Install more packages with the following yum commands. **Again, note that this is the procedure we used to install PostgreSQL 10.5 as part of a Red Hat Software Collections. You can use any method as long as the OS and the DBMS meet the TPC Pricing Specification's support requirements.**

1. yum clean all
2. subscription-manager register
3. subscription-manager attach
4. subscription-manager list --available | grep -iE 'Subscription Name|pool'
  - Find the relevant subscription (in this case, Red Hat Enterprise Linux), and replace the XYZ below with the Pool ID listed after that subscription
5. subscription-manager attach --pool=XYZ
6. subscription-manager list --consumed
  - Confirm that the subscription is listed as consumed
7. subscription-manager repos --list
  - There may be hundreds of repos. But it is actually simple: grep for rhscl, and there are a few repos and the obvious one is rhel-server-rhscl-7-rpms
8. subscription-manager repos --enable rhel-server-rhscl-7-rpms
  - Find the *Software Collections component* name [here](#), e.g., rh-postgresql10
9. yum install rh-postgresql10 rh-postgresql10-postgresql-server rh-postgresql10-postgresql-devel postgresql-odbc
  - Sometimes we find out later that we are missing /usr/lib64/psqlodbcw.so and have to re-install postgresql-odbc
10. yum erase postgresql.x86\_64 postgresql-docs.x86\_64 postgresql-server.x86\_64 postgresql-libs.x86\_64
11. yum install rh-postgresql10-syspaths
12. yum -y install java-1.8.0-openjdk java-1.8.0-openjdk-devel
13. yum -y install unixODBC unixODBC-devel
14. yum -y install ntp
15. If you want to speed up the compression of result log files, also install:
  1. yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
  2. yum install lbzip2

## 2.3 Configure the Operating System

1. Increase the limits in /etc/security/limits.conf

```
* soft nproc 578528
* hard nproc 578528
* soft memlock 33554432
* hard memlock 33554432
```

```
postgres soft nproc 578528
postgres hard nproc 578528
postgres soft memlock 536870912
postgres hard memlock 536870912
```

## 2. Add these to /etc/sysctl.conf

```
kernel.sem = 500 64000 400 512
fs.file-max = 500000
vm.min_free_kbytes = 65536
kernel.shmall = 134217728
kernel.shmmax = 549755813888
kernel.shmmni = 4096
fs.file-max = 6815744
fs.aio-max-nr=1048576
kernel.core_uses_pid = 1
kernel.wake_balance = 0
kernel.perf_event_paranoid = -1
vm.swappiness = 0
vm.dirty_background_bytes = 10000000
```

## 3. Disable the firewall:

```
systemctl stop firewalld
systemctl disable firewalld
systemctl stop iptables
systemctl disable iptables
```

## 4. In /etc/selinux/config, set SELINUX=disabled. Otherwise, systemctl start rh-postgresql10-postgresql fails

## 2.4 Install PostgreSQL and the Kit

You should now have a user *postgres* on the VM; if not, create it. Perform the following steps as user *postgres*

1. Put the TPCx-V kit tar files *xVAudit.tgz*, *VGen.tgz*, *VDb.tgz*, and *VDriver.tgz* into the */opt* directory
2. Untar the kit files to their target locations

```
tar xzvf /opt/VGen.tgz --directory=/opt
tar xzvf /opt/VDB.tgz --directory=/opt
```

```
tar xzvf /opt/VDriver.tgz --directory=/opt
tar xzvf /opt/xVAudit.tgz --directory=/opt
chown -R postgres:postgres /opt/VDb /opt/VDriver /opt/VGen /opt/xVAudit
```

3. Add the following lines to the end of `~postgres/.bash_profile` to set these values for future logins.

Note the setting for PGDATA to `/dbstore/pgsql/tpcv`, which is used throughout this Guide. If you want to store the postgres data and configuration files in a different location, change PGDATA accordingly.

```
export PGDATA=/dbstore/pgsql/tpcv
```

4. We use the `pgstatspack` package to collect performance data. **As user postgres**, download the file `pgstatspack_version_2.3.1.tar.gz` from [pgfoundry.org/](http://pgfoundry.org/), untar it in postgres home directory, cd to it, and run the `install_pgstats.sh` script.

## 2.4.1 Clone virtual machines

The run-time bash scripts depend on the prime driver being able to ssh into all 13 VMs, as both root and postgres, using the file `~.ssh/authorized_keys`. It's a good idea to set this up on the first (template) VM before making the clones

```
ssh-keygen # You can simply hit the Return Key in response to all the questions
ssh-copy-id root@nameofmyvm # Use the Network Name of your VM here
ssh-copy-id postgres@nameofmyvm
```

Test this by logging in as user postgres and running a simple command like `"ssh nameofmyvm date"`. If it still asks for a password, run the following command as `root`:

```
restorecon -R -v /root/.ssh
```

At this point, your VM has all the required software. You can now clone it 12 (or 24, etc.) times for all the VMs in a TPCx-V configuration, and customize the Tier A and Tier B VMs by following the instructions below

On the VMware vSphere platforms, you need to install the `deployPkg` plug-in as described in KB article 2075048. You can then clone VMs by simply providing the new VM's network name and IP address.

If you clone the Tier B VMs from a single source, remember to properly resize the virtual disk that holds the `/dbstore` file system, and run `resize2fs(8)` after powering up the VM.

You can use the `VDriver/scripts/cloneVMs.ps1` PowerCLI script to clone the VMs in a VMware vSphere environment. You will need to invest time in installing VMware PowerCLI on a Windows PC (or VM!), and customizing the script for your environment. That time will be well-spent as it will automate the cloning process, and will save a lot of time in the long run.

## 2.5 Configure a Tier B Virtual Machine

Use the instructions in this section to finish configuring a VM for use in Tier B. Note that Tier B VMs require additional virtual disk space per table in section 2.1.1

## 2.5.1 Create Tablespace Directories

**Run this step as root on all database VMs.**

If a separate file system will be used to store the postgres data files, create it with the command below (change /dev/sdb to whatever virtual disk will hold the database):

```
mkfs.ext4 -F -v -t ext4 -T largefile4 -m 0 -j -O extent -O dir_index /dev/sdb  
and mount it as /dbstore with mount arguments: nofail,noatime,nodiratime,nobarrier
```

It is convenient to label the file system with a label such as DBSTORE to aid in mounting it. You can add a line to /etc/fstab to have the OS mount the file system every time the system reboots.

So, for example, label the file system with the command `e2label /dev/sdb DBSTORE`, and add the following line to /etc/fstab:  
`LABEL=DBSTORE /dbstore ext4 nofail,noatime,nodiratime,nobarrier 2 2`

Run the following commands on the Tier B VM:

```
rm -rf /dbstore/tpcv* /dbstore/pgsql  
rm -rf /pg_wal/pgsql  
mkdir -p /dbstore/pgsql/tpcv  
mkdir -p /dbstore/pgsql/tpcv-data  
mkdir -p /dbstore/pgsql/tpcv-index  
mkdir -p /dbstore/pgsql/tpcv-temp  
mkdir -p /pg_wal/pgsql  
chown -R postgres:postgres /dbstore/pgsql /pg_wal/pgsql;
```

## 2.5.2 Setup PostgreSQL

**Run this step as root on all database VMs.**

1. Since we have customized the location of PostgreSQL configuration files to /dbstore/pgsql/tpcv, we need to let the operating system know where to find these files

1. Create a /etc/systemd/system/rh-postgresql110-postgresql.service file with the following contents:

```
.include /lib/systemd/system/rh-postgresql110-postgresql.service  
Environment=PGDATA=/dbstore/pgsql/tpcv
```

```
2. mkdir /etc/systemd/system/rh-postgresql10-postgresql.service.d
```

3. Create a /etc/systemd/system/rh-postgresql10-postgresql.service.d/restart.conf file with the following contents:

```
[Service]
Environment=PGDATA=/dbstore/pgsql/tpcv
```

2. Run the following commands on the Tier B VM to set up PostgreSQL:

```
systemctl enable rh-postgresql10-postgresql.service
/usr/bin/postgresql-setup --initdb --unit rh-postgresql10-postgresql
systemctl start rh-postgresql10-postgresql.service
```

3. Edit the \$PGDATA/pg\_hba.conf file to make the connection security levels trust rather than peer and ident. To do that, add an entry such as the first line below to this file, and modify the METHOD for the other 3 lines to "trust". **Obviously, use a CIDR block address range appropriate for your particular installation.**

```
host all all 10.196.00.00/16 trust
local all all trust
host all all 127.0.0.1/32 trust
host all all ::1/128 trust
```

4. Update the postgresql.conf file (\$PGDATA/postgresql.conf) to have PostgreSQL listen on the local ip address and not just localhost. Update (may need to uncomment) the listen\_addresses parameter and add the ip address of the server. A setting such as listen\_addresses = '\*' would allow any client system to connect to the database. If you have security concerns, you can instead specify the IP address of the Tier A VM1 client in each of the Tier B VM2/VM3.

The following postgresql.conf changes are useful when loading the database

```
shared_buffers = 1024MB
wal_sync_method = open_datasync
wal_writer_delay = 10ms
checkpoint_segments = 30
checkpoint_completion_target = 0.9
```

5. Restart the PostgreSQL service so the changes will take affect

```
systemctl restart rh-postgresql10-postgresql
```

### 2.5.3 Create the File System for Flat Files

All Tier B VMs use the same flat files for populating the databases, even in a multi-Tile configuration. Each Group 1/2/3/4 loads some or all the flat files. So there is no need to generate identical flat files on all 8 database VMs and waste disk space. The 8 database VMs can nfs-mount /vgenstore from one VM that has generated the flat files. To create the file system for the flat files, run the following command (change /dev/sdb to whatever virtual disk will hold the flat files) as user root **on the single VM that will load the flat files**:

```
mkfs.ext4 -F -v -t ext4 -T largefile4 -m 0 -j -O extent -O dir_index /dev/sdb
```

It is convenient to label the file system with a label such as VGENSTORE to aid in mounting it. You can label the file system with the command `e2label /dev/sdb VGENSTORE`, and add the following line to /etc/fstab:

```
LABEL=VGENSTORE /vgenstore ext4 defaults 2 2
```

After you mount the new file system as /vgenstore, change its permissions:

```
chown postgres:postgres /vgenstore
```

## 2.5.4 Create the Flat Files

**Run this step as user postgres on the VM that will load the flat files in its /vgenstore file system (all other VMs will nfs-mount this VM's /vgenstore).**

Build the VGen binaries by executing the following commands:

```
cd /opt/VGen/prj/GNUMake
make
```

The space for the flat files (60GB for the minimum-sized database) can be anywhere as long as all Tier B VMs can access it, and as long as the "-f" argument to create\_TPCx-V\_flat\_files.sh and the parameter flatfile\_dir in VDb/pgsql/scripts/linux/env.sh are the same path. Here we assume the flat files are in /vgenstore/flat\_out

The script /opt/VGen/scripts/create\_TPCx-V\_flat\_files.sh creates the flat files. **The example below will create the flat files for 20 LUs, which will be loaded into Group 4 VMs. Groups 1-3 will load 5, 10, and 15 LUs, respectively. Overall, the Tile will have a total of 50 LUs.**

```
/opt/VGen/scripts/create_TPCx-V_flat_files.sh -f /vgenstore/flat_out -l 20
```

Flat files created with the above script creates flat files for all four Groups with the minimum customer counts of

```
Group A: 5000
Group B: 10000
Group C: 15000
Group D: 20000
```

and uses 60GB of disk space.

## 2.5.5 Create the TPCx-V database

**Assuming you have created a single set of flat files on one of the VMs, nfs-mount that /vgenstore file system on all database VMs. Run the rest of this step as user postgres on all database VMs.**

1. Make sure the parameter `flatfile_dir` in `/opt/VDb/pgsql/scripts/linux/env.sh` points to where the flat files were loaded. Change the parameter `scaling_tables_dirs` in `/opt/VDb/pgsql/scripts/linux/env.sh` to `"/Scaling[1-1]"`, `"/Scaling[1-2]"`, and `"/Scaling[1-3]"` for G1, G2, and G3 VMs, respectively.

**One of the most common issues users run into is missing this step, loading the full set of data files on all VMs, and filling up the file systems on G1/G2/G3 VMs**

2. On each Tier B VM, run the following to populate and set up the TPCx-V database:

```
cd /opt/VDb/pgsql/scripts/linux
./setup.sh
```

This phase will take about one hour when loading a default set of flat files. You will see some errors when we create the objects role and language, which already exist, or when we try to drop objects (tables and indexes) that were gone when we dropped the database. These errors are benign.

## 2.5.6 Back up the TPCx-V database

Backing up and restoring the databases is optional, but is much faster than reloading the database from flat files. One can use the built-in PostgreSQL commands to back up and restore. But we have found that simple Linux commands work much faster and more reliably.

If you want to back up the database, on each Tier B VM, run the following commands **as user root**:

```
systemctl stop rh-postgresql10-postgresql.service
cd /dbstore/pgsql
(find tpcv tpcv-data tpcv-index tpcv-temp | grep -v tpcv/pg_wal; find tpcv/pg_wal -follow) | cpio -o | lbzip2 -c >
/dbstore/pgsql/tpcv-backup.cpio.bz2
systemctl start rh-postgresql10-postgresql.service
```

## 2.5.7 Restoring the TPCx-V database

On each Tier B VM, run the following commands **as user root**:

```
systemctl stop rh-postgresql10-postgresql.service
cd /dbstore/pgsql
rm -rf tpcv/pg_wal/* tpcv tpcv-data tpcv-index tpcv-temp
lbzip2 -d /dbstore/pgsql/tpcv-backup.cpio.bz2 -c | cpio -idm
```

```
chown -R postgres:postgres /dbstore/pgsql
systemctl start rh-postgresql10-postgresql.service
```

## 2.6 Configure a Tier A Virtual Machine

Use the instructions in this section to finish configuring a VM for use in Tier A

### 2.6.1 Setup the ODBC environment

**Run this step as user *root*.**

This section only needs to be done on the Tier A virtual machines.

1. Populate the `odbc.ini` file from the kit

```
cat /opt/VDb/pgsql/prj/osfiles/odbc.ini >> /etc/odbc.ini
```

2. Edit `/etc/odbcinst.ini` and set

```
Driver64=/usr/lib64/psqlodbcw.so
```

3. Edit `/etc/odbc.ini` and set `ServerName` to the name of the two Tier B VMs that serve this Tier A. Note that you need to create two *Data Sources*, one that points to VM2, and one that points to VM3. You can name these Data Sources anything that you want as long as they match `VCONN_DSN_LABELS` in `vcfg.properties`; `PSQL2` and `PSQL3` are good choices.

4. Test out the TPCx-V and ODBC connections. First let's make sure the Tier B VM is reachable on the network, and the database is up and running. Run the following steps as user `postgres`. Set `Pghost` to the network name of a Tier B VM, then run a simple database query from the Tier A, such as the following command with its expected output:

```
-bash-4.2$ export Pghost=THE_NETWORK_NAME_OF_YOUR_TIERB
-bash-4.2$ psql tpcv -c "select count(*) from sector"
count
-----
12
(1 row)
```

Now, check that we can reach the same database via ODBC. Do this by using the `tradesstatus` test application in `/opt/VDb/pgsql/dml/test_programs`, which connects to the database on the datasource name `PSQL1` (if you have used `PSQL2` and `PSQL3` in `/etc/odbc.ini`, replace the `PSQL1` string with `PSQL2` or `PSQL3`, and re-make the application).

```
cd /opt/VDb/pgsql/dml/test_programs
make tradestatus
./tradestatus
```

## 3.0 Setting Up the Driver Environment

**Run this step as user postgres.**

These instructions assume you have already used Section 2 of this document to set up four Tier A VMs, each connected to two Tier B database VMs, for a total of 12 VMs.

Optionally, you can create an extra clone VM to be the prime driver. Although this is not required, if done, it will simplify the process since the prime driver will immediately have all the required software installed.

These instructions are intended to assist the user in setting up the TPCx-V prime driver, all subordinate drivers, market exchange emulators, and database connectors used to generate the database load. They also assume you have read the TPCx-V Benchmark Kit Design Document, Clause 10 of the benchmark Specification, and are familiar with concepts and terminology introduced there.

### 3.1 Building the Benchmark Driver

The following build instructions assume you are building the driver on Linux using GNU C and the OpenJDK Java Development Kit. Other environments may use different build scripts and build steps, though the fundamental build process should be very similar.

1. Copy the benchmark driver kit file(s) first to your prime driver in the desired directory. (The OVF template already has a copy of the kit; but it may not be up-to-date.)
2. If a Java Development Kit has not already been installed, install one on the prime driver in the desired directory.
3. From the base directory of the benchmark driver kit on the prime driver, go to the VDriver/prj/GNUMake directory.
4. Modify the included Makefile as appropriate for your environment. Most fields should require no modification. However, be certain that JAVA\_HOME points to the correct base directory for your Java development kit (JDK). Likewise, confirm that the C++ compiler and flags are correct for your environment.
5. Run "make" from the GNUMake directory to build the benchmark driver.

When the benchmark driver has built successfully, you should find a file named "vdriver.zip" in the /opt/VDriver/dist directory.

### 3.2 Copying Benchmark Driver Files

The vdriver.zip file contains all of the necessary benchmark driver files. You can copy the zip file to the desired directory on each Tier A SUT VM, as well as to each system that will be used as a CE driver and Market Exchange Emulator, and extract all files to each system. For example,

copy `vdriver.zip` from the `/opt/VDriver/dist` directory of the prime driver to the `/opt/VDriver/dist` directory of each of the 12 VMs, then on each VM run the following commands:

```
cd /opt/VDriver/jar
unzip -o ../dist/vdriver.zip -x vcfg.properties
```

However, the following instructions detail how to copy only the necessary files for each component.

1. Unzip `vdriver.zip` on the prime driver (or any other system from which you can copy the extracted files to all of the target systems).
2. Copy `vconnector.jar` and the "lib" directory to the desired directory on all Tier A VMs.
3. Copy `vmee.jar` and the "lib" directory to the desired directory on all systems that will be used as market exchange emulators.
4. Copy `vdm.jar` and the "lib" directory to the desired directory on the system that will be used to drive the data maintenance transactions.
5. Copy `vce.jar` and the "lib" directory to the desired directory on all systems that will be used as customer emulators.
6. If the system on which `vdriver.zip` was extracted is not the system that will be used as the prime driver or is not in the desired location on the prime driver, then copy `vdriver.jar`, `reporter.jar`, the "lib" directory, `vcfg.properties`, and `testbed.properties` to the desired directory on the prime driver.

Finally, confirm that you have a Java runtime environment (JRE) or development kit (JDK) installed on all systems that will be running these benchmark driver components, and that the installed version is the same major revision level or newer than the one used to build the benchmark driver.

### 3.3 Editing the Configuration File

The configuration file (`vcfg.properties` in `/opt/VDriver/jar`) is where the benchmark run configuration is defined. A copy of `vcfg.properties` will be archived in `driver.zip` when the kit gets compiled and built. This section deals only with the most commonly modified parameters.

**The benchmark requires the setting of a potentially large number of port numbers. You can specify the port numbers individually (e.g., `VCONN_TXN_PORT[2][3][6] = "48123"` for Tile 2, Group 3, `vconnector` process 6); or you can specify the first port number without any indexes (e.g., `VCONN_TXN_PORT[] = "44000"`), and let the benchmark automatically assign consecutive port numbers for all the listening ports. In the normal use case, you can use the simpler second method.**

#### VM\_TILES

Set this to the number of Tiles being used for the test. (These setup instructions assume a single-Tile configuration.)

#### VDRIVER\_RMI\_HOST

#### VDRIVER\_RMI\_PORT

Set these values to the network interface hostname and port on which the prime driver will listen for RMI commands.

#### NUM\_DRIVER\_HOSTS

Set this value to the number of CE driver processes used in a test. There are two common reasons for using multiple CE driver processes:

1. You can drive the benchmark from multiple systems. Each system needs at least one CE driver process

2. Sometimes the results log files get very large, and the reporter process has a hard time dealing with them. With multiple CE driver processes, we have more, smaller log files, which are easier to handle.

#### NUM\_CE\_DRIVERS

##### NUM\_CE\_DRIVERS[x]

Modify NUM\_CE\_DRIVERS in order to change the total number of CE driver *threads* used to generate load in a test. Note that these threads are evenly distributed between all CE driver processes. In order to override this default distribution of threads, use the indexed version to specify how many driver threads are run in each driver process.

##### VCE\_RMI\_HOST[x]

##### VCE\_RMI\_PORT[x]

Set these values to the network interface hostname and port on which each vce client process will listen for RMI commands. Note that any vce processes not defined in this way will default to the value assigned to the non-indexed form of these parameters.

#### NUM\_VCONN\_PER\_GROUP

The "vconnector" is the process on the Tier A VM (VM1) that receives transactions from the driver and submits them to the database. There can be one or more vconnector processes on each Tier A.

NUM\_VCONN\_PER\_GROUP is the number of VConnector processes running on each Tier A VM (The requests will be distributed across all of these processes). Each process is multi-threaded, and one process may be enough. But if you see odbc contention issues on the Tier A VM1, increase this value

##### VCONN\_RMI\_HOST[tile][group]

##### VCONN\_RMI\_PORT[tile][group][vconn]

Set these values to the network interface hostname and port on which each vconnector process will listen for RMI commands. The common practice is to set VCONN\_RMI\_HOST[tile][group] (since each Tier A for [tile][group] runs on a different VM with a different network name); but leave the selection of the port number to the benchmark by simply setting a single unindexed VCONN\_RMI\_PORT[] (empty brackets []) for the first port number for whole configuration.

##### VCONN\_TXN\_HOST[tile][group]

##### VCONN\_TXN\_PORT[tile][group][vconn]

Set these values to the network interface hostname and port on which each vconnector process will accept transactions from the vce or vmee processes. The common practice is to set VCONN\_TXN\_HOST[tile][group] but leave the selection of the port number to the benchmark by simply setting a single unindexed VCONN\_TXN\_PORT[].

##### VCONN\_DSN\_LABELS[tile][group]

Set this to the label strings for the databases used, as defined in your Tier A VM's odbc.ini file, with each label delimited by a comma. There must be as many label names as Tier B databases in use. The common case is to set an unindexed VCONN\_DSN\_LABELS[] = "PSQL2,PSQL3" to cover the whole SUT.

##### VCONN\_NUM\_DBS[tile][group]

Set this to the number of databases being used by the Tier A VM of the same Tile and Group indexes. The common case is to set an unindexed VCONN\_NUM\_DBS[] = "2" to cover the whole SUT.

VMEE\_RMI\_HOST[index]

VMEE\_RMI\_PORT[index]

Set these values to the network interface hostname and port on which each vmee process will listen for RMI commands. Note that any vmee processes not defined in this way will default to the value assigned to the non-indexed form of these parameters.

**All VMEE processes must run on the same system (VM or physical server). But different VMEE\_RMI\_HOST parameters may be specified to allow the networking traffic to flow over different NICs.**

MEE\_TXN\_HOST[tile][group][vconn]

MEE\_TXN\_PORT[tile][group][vconn]

The network interface hostname and port on which the individual MEE threads accept transactions. The MEE threads are divided between the VMEE processes. There is a 1-1 mapping between vconnector processes on Tier A VMs and MEEs. The VMEE process will have one MEE thread for each vconnector process.

**All MEE threads must run on the same system (VM or physical server) as the VMEE process(es) but setting MEE\_TXN\_HOST[] to a different value allows the networking traffic to flow over a different NIC.**

**Although it is possible to set the host name and port number for each individual MEE, it is a lot easier to use the unindexed (empty brackets []) parameters. By specifying a single port number in MEE\_TXN\_PORT[], the kit will automatically select as many port numbers as necessary for all MEEs starting with that value.**

VDM\_RMI\_HOST

VDM\_RMI\_PORT

Set these values to the network interface hostname and port on which the vdm process will listen for RMI commands.

RUN\_ITERATION\_SEC

Set this value to the number of seconds that a measurement interval to run. (Modifying this value from the default will result in a non-compliant run.)

DRIVER\_SCALEUP\_SEC

RAMPUP\_SEC

DRIVER\_RAMPDN\_SEC

RAMPUP\_SEC is the number of seconds of ramp-up at the beginning of the *first* iteration of the test run. Ramp-up for a valid run has to be at least 12 minutes to ensure we have reached the steady-state mode before starting the measurement interval. DRIVER\_RAMPDN\_SEC is the ramp-down period to gradually reduce the load at the end the benchmark run. Included in Ramp-up is DRIVER\_SCALEUP\_SEC, which is the number of seconds to gradually increase the load to the full level at the beginning of a run. We are at full load after DRIVER\_SCALEUP\_SEC. A 30-60 second DRIVER\_SCALEUP\_SEC is usually adequate.

After transactions start executing at full load, it takes 6 minutes for limit-order Trade-Results transactions to reach their steady-state throughput. So you want the difference between RAMPUP\_SEC and DRIVER\_SCALEUP\_SEC to be at least 6 minutes

NUM\_RUN\_PHASES

Set this to the number of run phases you wish to run in a single measurement interval. There must be a `GROUP_PCT_DIST_PHASE` defined for as many run phases as specified. (Modifying this value from the default will result in a non-compliant run.)

#### `NUM_RUN_ITERATIONS`

Set this to the number of times to run a full set of run phases.

Other parameters such as `CUST_CONFIGURED[]`, `CUST_ACTIVE[]`, `NUM_VMEE_PROCESSES`, and `NUM_RUN_PHASES` are described in the comments in `vcfg.properties`

## 4.0 Using the supplied bash scripts to run a test

### Run this step as user `postgres`.

There are a number of shell scripts in the directory `/opt/VDriver/scripts/rhel6` that facilitate the running of a test and post processing the results.

`runme.sh` is the main script. It starts the component processes on all VMs, runs the test, post processes the logs to produce results all the way to a TPCx-V Executive Summary, and can optionally collect stats, run the consistency tests, and run the Trade-Cleanup transaction before and after the run. The results of each run are saved in the directory `/opt/VDriver/results/RUN_ID` with a unique `RUN_ID` for each run. Run `runme.sh -h` to get the list of options. Of particular note is `[-j size|--java_heap size]`. Sometimes the reporter app runs out of memory when processing large log files. Use this option to increase the java heap size.

`rerun_reporter.sh` repeats the last post-processing step of running the reporter app on the final sorted log of all recorded transactions. This is useful, for example, after a multi-iteration run where you want to experiment with different Measurement Intervals to find the peak throughput.

`iostat_summary.sh` averages the `iostat` output of each VM over the Measurement Interval, and prints a line for each database (virtual) disk of each Tier B VM.

`killrun.sh` logs into all VMs, and kills the benchmark processes. Note that `killrun.sh` gets the network names of the VMs for a given run from the file `/opt/VDriver/results/RUN_ID/allhosts`, which is created by `runme.sh`. If for some reason this file is not created (e.g., you hit CTRL-C before `runme.sh` had a chance to create `allhosts`; or because of syntax issues in `vcfg.properties`), `killrun.sh` won't work, and you will need to manually log into each VM, and kill the benchmark processes.

`summary.sh` gives a 1-line summary of the performance for a given run number.

`trade-cleanup.sh` runs the Trade-Cleanup transaction for the last completed run. Alternatively, the `-t` option to `runme.sh` will automatically run Trade-Cleanup after every run.

`collect_config.sh`, `tools.sh`, `finish_tools.sh`, `killlocal.sh`, and `run_consistency.sh` are auxiliary bash scripts that get invoked by other scripts.

## 5.0 Manually starting a Test Run

If instead of using the supplied bash scripts, you need to manually launch individual component processes, follow the instructions below. **Note that running the kit in this manner is typically for debugging purposes only. You would normally just use the runme.sh script.**

### 5.1 Starting the Component Processes

As the diagram, above, illustrates, the vconnector, vmee, and vce processes all receive RMI calls from the prime driver, and as such, the hostname of the network interface on which the user intends to communicate with the prime driver must be specified when invoking these processes. Likewise, unless the user intends to listen on the default RMI network port for these RMI calls, the network port must also be specified when invoking these processes.

#### 5.1.1 Starting the VConnector Process

Go to the directory in which the vconnector jar file is installed on each Tier A VM and start each vconnector process using the following invocation string format:

```
java -jar [-Djava.library.path=<path_to_c_libs>] vconnector.jar -rh <hostname> [-rp <port_num>]
```

An example invocation string:

```
java -jar -Djava.library.path=lib vconnector.jar -rh tpcv_vm1 -rp 30000
```

#### 5.1.2 Starting the VMee Process

Go to the directory in which the vmee jar file is installed on each system being used as a market exchange emulator and start each vmee process using the following invocation string format:

```
java -jar [-Djava.library.path=<path_to_c_libs>] vmee.jar -rh <hostname> [-rp <port_num>]
```

An example invocation string:

```
java -jar -Djava.library.path=lib vmee.jar -rh tpcv_mee1 -rp 32000
```

Be certain to specify different RMI ports for each vmee process on the same system and using the same network interface.

#### 5.1.3 Starting the VCe Process

Go to the directory in which the vce jar file is installed on each system being used as a customer emulator and start each vce process using the following invocation string format:

```
java -jar [-Djava.library.path=<path_to_c_libs>] vce.jar -rh <hostname> [-rp <port_num>]
```

An example invocation string:

```
java -jar -Djava.library.path=lib vce.jar -rh tpcv_ce1 -rp 31000
```

Be certain to specify different RMI ports for each vce process on the same system and using the same network interface.

#### 5.1.4 Starting the VDM Process

Go to the directory in which the vdm jar file is installed on the system being used as a data maintenance transaction driver and start the vdm process using the following invocation string format:

```
java -jar [-Djava.library.path=<path_to_c_libs>] vdm.jar -rh <hostname> [-rp <port_num>]
```

An example invocation string:

```
java -jar -Djava.library.path=lib vdm.jar -rh tpcv_dm1 -rp 34000
```

Once all of these driver components are started, they are listening on their RMI ports and waiting for the prime driver to send them the needed configuration information and to start the test run.

## 5.2 Starting the Prime Driver Process

After confirming the values in the configuration file are complete and correct for the desired run characteristics, go to the directory in which the vdriver jar file is installed on the prime driver and start the vdriver process using the following invocation string format:

```
java -jar [-Djava.library.path=<path_to_c_libs>] vdriver.jar -i <flat_file_location>
```

An example invocation string:

```
java -jar -Djava.library.path=lib vdriver.jar -i /opt/tpcv/VGen/flat_in
```

This will start the test run, and all vce, vmee, and vconnector processes, as well as the vdriver process should terminate at the end of the run.

## 6.0 Post-Run Reporting

The `runme.sh` script will automatically invoke the `reporter.jar` application multiple times to sort the log files, merge them into a single log file, produce an ASCII file with the required performance and compliance data, and finally generate an Executive Summary. (Run this step as user `postgres`.)

Alternatively, you can follow the instructions below to invoke `reporter.jar` to post-process the logs.

During the measurement interval, each vce and vmee process records the response time for each request type on a *per-phase* basis in a transaction mix log. At the end of the run, the log files from all of these vce and vmee processes must be pre-sorted and then merged into a single mix log file using the reporter. (Note that if you set `SORT_MIX_LOGS = "1"` in `vcfg.properties`, the vce and vmee processes will sort their log files by timestamp at the end of the run. Otherwise, it must be done by the reporter.)

The following instructions assume you have collected all of the vce and vmee log files and placed them in the same directory as the `reporter.jar` file. They also assume that there is a "results" subdirectory that includes the timestamped subdirectories for each run, and in which subdirectory exists a `runtime.properties` file that includes the specific configuration information for the run that the reporter is intended to process.

There are four flags that you may need to use when invoking the reporter:

- `-s`: this flag tells the reporter that the file or files that it will process need to be *sorted* by timestamp
- `-m`: this flag tells the reporter to *merge* all sorted mix files into a single, time-sorted file
- `-r`: this flag tells the reporter to write a final transaction rate *report* based on the log entries from the single, merged, time-sorted file

- `-es`: this flag tells the reporter to write an HTML-formatted *Executive Summary* report based on the log entries from the single, merged, time-sorted file as well as the information contained in the `testbed.properties` file

If you pass the reporter none of these flags, it expects that you have also passed it the name of a single, sorted, merged log file that only requires writing a final transaction rate report. Otherwise, you should pass it the combination of these four flags that reflects the tasks you would like the reporter to perform.

The reporter process is invoked using the following format:

```
java -jar [-Djava.library.path=<path_to_c_libs>] reporter.jar -c <path_to_runtime.properties> [-t  
<path_to_testbed.properties>] [-h] [-s ] [-m] [-r] [-es] [-o <merged_log_name/path>] [-i  
<desired_sample_interval_seconds>] [-si <measurement_starting_iteration>] [-sp <measurement_starting_phase>] [-mp  
<total_measurement_phases>] <mix_log_file_name(s)>
```

## 6.1 Mix Log File Sorting

In order to process the mix log files from the CE and MEE processes, these files have to be sorted (by timestamp), and then merged. The following is an example reporter invocation string that might be used for sorting a single mix log file:

```
java -jar -Djava.library.path=lib reporter.jar -c results/20151118-142511/runtime.properties -s CE_Mix-0.log
```

This would create a new log file named `CE_Mix-0-sorted.log` that is ready to be merged with any other pre-sorted logs from the same run.

## 6.2 Mix Log File Merging

Once all of your CE and MEE mix logs are sorted, they are ready to be merged into a single, combined mix log file. The following is an example reporter invocation string that might be used for merging multiple mix log files:

```
java -jar -Djava.library.path=lib reporter.jar -c results/20151118-142511/runtime.properties -m -o mixlog-merged.log  
CE_Mix-0-sorted.log CE_Mix-1-sorted.log MEE_Mix-0-sorted.log MEE_Mix-1-sorted.log
```

This would merge the four (pre-sorted) mix logs into a single sorted and merged log file named `mixlog-merged.log`. (A single, combined mix log file is required for creating the transaction mix validation and audit check log files as well as an HTML-formatted Executive Summary document.)

## 6.3 Transaction Mix and Audit Check Log File Creation

Once you have a single sorted and merged log file to use as input, you can then create the transaction mix validation and audit check log files by using a reporter invocation string similar to the following:

```
java -jar -Djava.library.path=lib reporter.jar -c results/20151118-142511/runtime.properties -i 30 mixlog-merged.log
```

Note that the "-i 30" was not actually required because 30 seconds is the default time sampling interval. Additionally, the "-r" flag was not used in this invocation. That flag is only needed when including the these log files creation with the previous merging step. But if you provide the reporter with a single, pre-sorted and pre-merged mix log file, the creation of this performance log will occur with or without this flag's inclusion in the invocation string. Note also that the reporter will try to write these log files to the same directory as the directory in which the driver wrote the runtime.properties file.

## 6.4 Executive Summary Creation

The HTML-formatted Executive Summary is created using a similar format to that of the performance summary log, above. In fact, the performance summary log is created as a part of the Executive Summary creation, making it unnecessary to create the former prior to creating the Executive Summary pages. However, the Executive Summary requires additional details about the hardware and software configuration used for the test, as well as component pricing information. So the user must also fill out the testbed.properties file, and optionally, the sample\_pricing.xls file as an easier alternative method for providing pricing data.

If you choose to use the provided sample\_pricing.xls file to fill in the test system's pricing data, after filling in the information required, you will need to save the file in csv format, because the reporter expects a csv-formatted file if a pricing file is passed in separately. You can alternatively choose to provide the pricing information in the testbed.properties file. You will find the pricing section commented out in this file. (This is considerably more difficult, and therefore prone to error, and is therefore commented out by default.)

Pass these files names and locations to the reporter using the -t and -p flags, respectively, when invoking it to create an Executive Summary. The reporter invocation string would look similar to the following in format:

```
java -jar -Djava.library.path=lib reporter.jar -c results/20151118-142511/runtime.properties -t testbed.properties -p pricing.csv mixlog-merged.log
```

If unsure how to fill out the testbed.properties file, the user can first invoke the reporter using the sample testbed.properties file provided with the kit, and the resulting Executive Summary Report should help clarify where and how these properties are being used in the report.

## 6.5 Advanced Reporter Features

Up to here, all of the reporter instructions have implicitly been assuming that the test that was run was a single iteration of ten phases. However, neither the benchmark kit nor the reporter is so limited. If the user runs multiple iterations of the 10-phase test, any ten consecutive phases may be selected as the measurement interval to use for the report (or even greater or fewer than ten phases for a non-compliant test).

To use this reporter functionality the user will include one or more of the following flags when invoking the reporter:

- -si <starting\_iteration>: This flag is used to specify the iteration that is the beginning of the measurement interval chosen. (Default: 0)
- -sp <starting\_phase>: This flag is used to specify the phase number that is to be the first phase of the measurement interval chosen. (Default: 0)
- -mp <total\_phases>: This flag is used to specify the total number of phases that are to be included in the measurement interval. (Default: 10; note that any number other than 10 will make this measurement interval non-compliant)

As an example, assume that a test is run with two iterations of ten phases, and it seems as though the user may have used too short of a warmup time, because prior to the first phase of the first iteration the log data suggests that the system has still not reached a steady state condition. So the user wants to create a report starting at the *second* phase of the first iteration and continue through the first phase of the second iteration. The reporter invocation would look similar to:

```
java -jar -Djava.library.path=lib reporter.jar -c results/20151118-142511/runtime.properties -t testbed.properties -si 0 -sp 1 mixlog-merged.log
```

Note that the `-mp` flag was not included because the default value is being used (and must be used for a compliant run). Similarly, because the default starting interval is also 0, the `-si` flag could have been left out, as well.

## 7.0 TPCx-V Auditing

One of the components of the TPCx-V benchmark kit is the self-audit module. A test sponsor can execute the various tests implemented in the xVAudit application, and provide the outputs to the auditor. The file `/opt/xVAudit/doc/README` lists these tests along with short descriptions of them.

The xVAudit application executes the audit tests on all the databases in the testbed, so it needs to know about all these databases, and know how to access them. That information is collected during the run by the `runme.sh` script, and is deposited in the file `tierb.properties` in the results directory of the run. The `tierb.properties` file should be passed to the xVAudit application with the `-c` option.

The xVAudit application needs a jdbc connection driver, such as `postgresql-9.4-1201.jdbc41.jar` (or newer), which you can download from `jdbc.postgresql.org`. The version number is often embedded in the driver file name, so we cannot know in advance what the file name is. So, the `runme.sh` script assumes it is simply called "postgresql-jdbc.jar".

Please copy the jdbc driver to the `/opt/xVAudit/jar` directory, rename it to `postgresql-jdbc.jar`, and run the Audit test by following the example below:

```
cd /opt/xVAudit/prj/GNUMake
make
cd /opt/xVAudit/jar
java -cp xVAudit.jar:postgresql-jdbc.jar xVAudit.StoredProcs.StoredProcAudit -c
/opt/VDriver/results/600/tierb.properties
```

The full list of xVAudit tests is documented in `/opt/xVAudit/doc/README`

## 8.0 Tips

- The benchmark specification requires that the clocks on all the drivers and SUT VMs be in sync. An easy way of doing that is installing the Network Time Protocol (ntp) rpm, and letting the prime driver act as the server.
  - On the ntp server (prime driver), comment out the lines that start with "server", and insert the following

```
server 127.127.1.0  
fudge 127.127.1.0 stratum 10
```

- On the ntp clients (all other VMs), comment out the lines that start with "server", and insert the following with the *NTPSERVER* being the IP address or network name of the prime driver

```
server NTPSERVER iburst
```

- Then run the following commands on the prime driver and all the VMs:

```
systemctl enable ntpd; systemctl start ntpd
```