DATASTAX

Best Practices for Running DataStax Enterprise within Docker

Table of Contents

Introduction

DataStax has a strategic imperative to help our customers become Internet Enterprises. For the past 5+ years, we have being delivering Apache Cassandra™ (with its genesis within large scale Internet companies such as Amazon, Google and Facebook) as part of a database platform purpose built for web, mobile and IoT applications.

Alongside Big Data's adoption within enterprises, there has been a movement to adopt agile development and deployment models utilized at Internet companies. Linux Containers have emerged as a crucial foundation for this DevOps style application lifecycle management. This meteoric rise is highlighted below trend analysis of number of searches on Google[1] for containers vs. virtual machines[2].

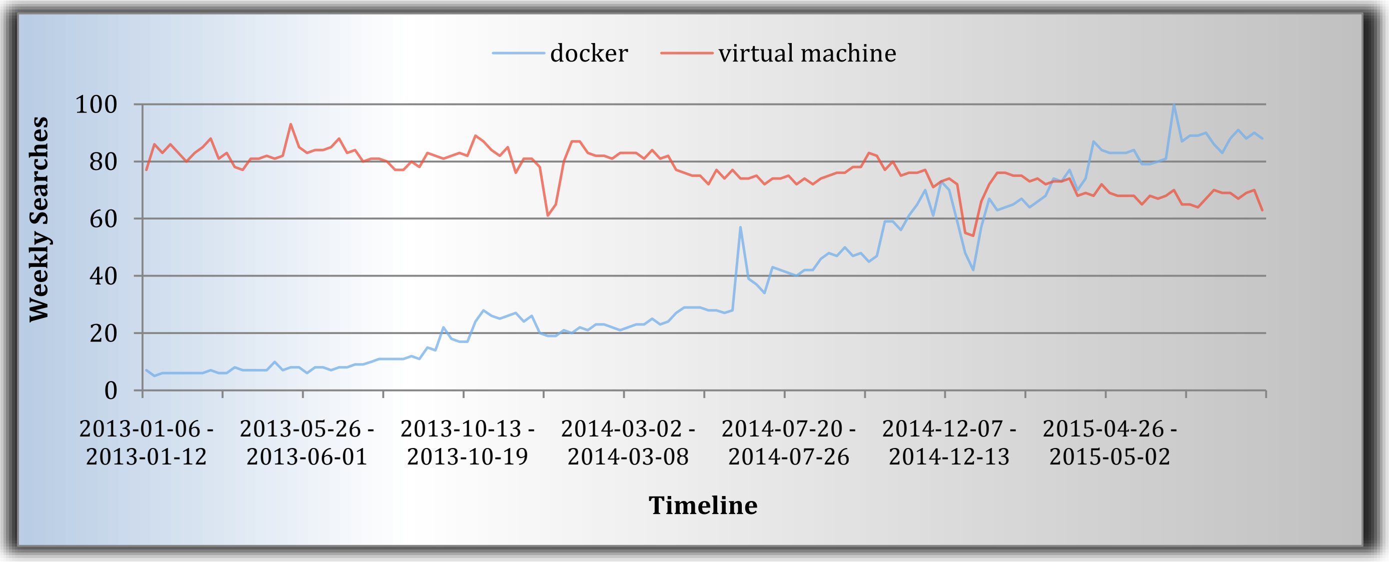

Figure 1 - Rise of container interest between 2013 and 2015; spearheaded by Docker

As it's clear from the trend above, the container movement (spearheaded by Docker) has recently overtaken virtual machines in search interest and is expected to grow further. This being the case, it's no surprise that customers have been asking for guidance on how to best run DataStax software with containers. This short paper describes best practices for running DataStax Enterprise (DSE) and OpsCenter within Docker environments, and includes trade-offs to be aware of and pitfalls to avoid[3].

[1] Sourced from Google Trends

[2] VMs were the dominant model of deployment within enterprises be considered as a predominant model of application deployment in the mid to late 2000's

[3] Please refer to linked resources to understand the basics of Docker, Linux containers etc.

Why Docker?

Before proceeding further, it's prudent to understand the evolution of application development and deployment paradigms in the past few decades as shown in diagram below.

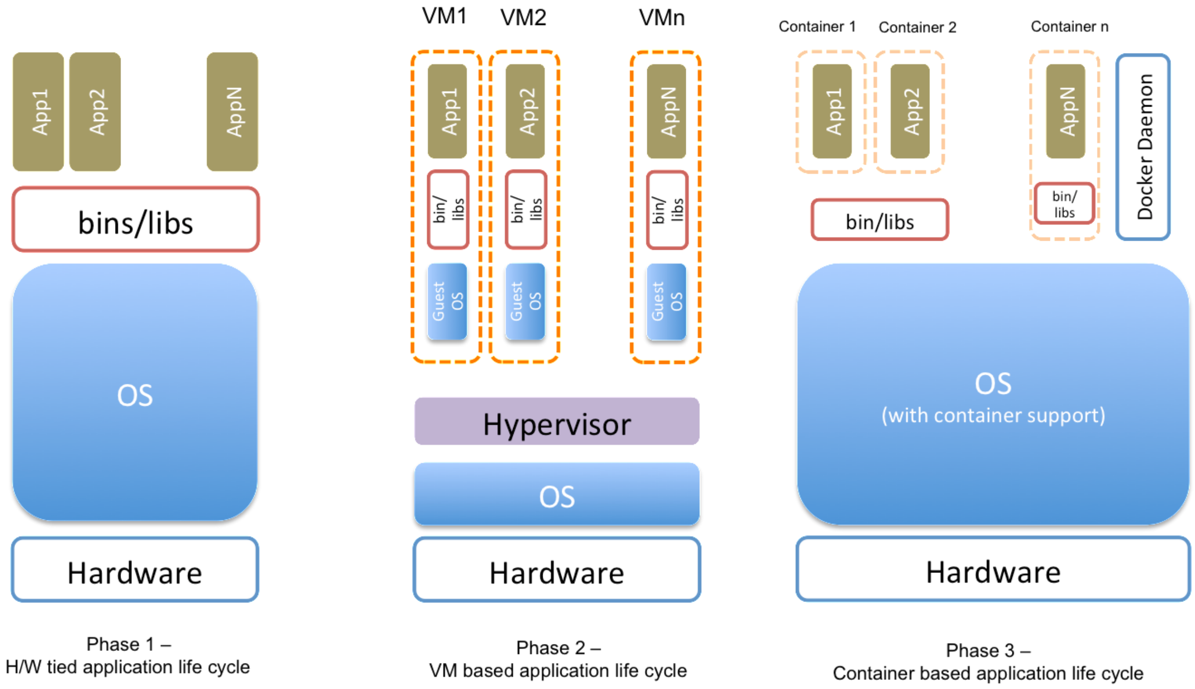

Figure 2 – The evolution of application life cycle deployment

While the evolution is pretty self explanatory, few aspects worth highlighting w.r.to containers:

- Containers provide "middle of the road" characteristics with respect to isolation and overhead
- Container based deployment requires an operating system with built in "container" support[4]
- It allows for flexibility in sharing binaries and libraries across applications if so desired
- Containers, unlike VMs, don't allow for heterogeneous operating system deployments to co-exist
- Docker makes the process of managing, maintaining and deploying containers turnkey

At a macro level, there are 2 key reasons why enterprises are adopting Docker (or containers more generally) for application life cycle management:

1) Build once, run anywhere – Any application packaged within the Docker framework is provided with a standards-based mechanism for managing its dependencies. This enables the packaged application to be deployed anywhere and moved around with ease
2) Better H/W Efficiency – As with the VM movement, underlying hardware has continued to grow at a non-linear pace. This enables a single physical host to masquerade as multiple virtual instances delivering significant total cost of ownership benefits. Containers bring the overhead of this virtualization to as close to bare metal as it can get[5]

[4] Unlike VMs which can be created on any OS

[5] This is possible as containers provide much lower levels of isolation as compared to VMs

Running DataStax Enterprise and OpsCenter with Docker

A typical DSE node[6] runs the following processes on a single instance within the cluster:

- A single core DSE JVM – Include Apache Cassandra, integrated DSE Search and Spark Master (for HA)
- One or more Spark executor processes
- A single Spark Worker process
- Multiple processes for the integrated Hadoop stack
- Multiple process which maybe started in an adhoc manner (e.g. Spark Job server, SparkSQL CLI, etc.)
- A single OpsCenter agent responsible for monitoring all processes on that DSE instance

The OpsCenter daemon is (logically) separate from the cluster and there is usually one[7] instance for the entire deployment.[8]

For the purposes of this document it is recommended that all the DSE processes be configured as 2 containers[9]:

- Container 1 - OpsCenter daemon
- Container 2 - All the JVMs running on a single DSE node (uniformly deployed across the each machine within the cluster)

Prerequisites

1. Install Docker based on instructions found here for the environment under consideration
2. Create a Docker file based on instruction here. An example Docker file (and related utilities) are provided in *Appendix 1 – Example Docker Files*

Steps to create an OpsCenter container

1. Download OpsCenter for your environment from here and place it in the same directory where the Docker file was created

2. Build the Docker image using the below command
```
a. docker build -t <opsc_image_name> .
```

3. Start the OpsCenter image using the below command and provide it a name that will be used for linking with the DSE container in the following section
```
a. docker run -d --name <opsc_container_name> <opsc_image_name>
```

4. Determine the OpsCenter container IP address using the below command
```
a. docker exec OPSCENTER_CONTAINER_ID hostname -I
```

5. At this point the OpsCenter daemon will be accessible within the browser @
   i. **http://<opcd_container_ip>:8888**

Steps to create a DSE Cluster

1. Download DSE for your environment from here and place it in the same directory where the Docker file was created in step 2 above (assuming its not already scripted)

2. Build the docker image using the below command
```
docker build -t <dse_image_name> .
```

[6] Node within this context is a single DSE instance

[7] We are ignoring any additional instances that can be used for OpsC HA (e.g. standby)

[8] Note that OpsCenter daemon can manage multiple DSE clusters

[9] In future we plan to provide guidance on utilizing advanced configurations such as splitting up Apache Spark JVMs and DSE Core JVMs within their own separate containers.

3. Start DSE on the specified Docker container using the below command

   ```
   docker run –link <opsc_conatiner_name> -d <dse_image_name> <options>
   ```

   - `–link <opsc_conatiner_name>` enables the DSE container to be linked with the OpsCenter container created in an earlier step. For more details w.r.to linking containers please refer to detailed documentation here
   - `<options>` to be used can be any that can be specified for regular DSE installations. For more details, please refer to DSE documentation here

4. Start the OpsCenter agent within the DSE container using the command below[10]

   ```
   docker exec DSE_CONTAINER_ID datastax-agent
   ```

5. Repeat steps 1 to 4 on all nodes within the cluster. At this point a dockerized DSE cluster and the corresponding OpsCenter setup is up and running

Configuration notes

While the steps above represent the bare minimum required to create a DSE cluster; certain highly recommended optimizations include the following:

- Given DSE is a distributed database; it is highly recommended that only one DSE node be configured to run per Docker host to achieve optimal networking configuration[11]. To do so, use the `--net=host` option. Please refer to documentation here for details about networking options. If it is necessary to specify the IP address of the networking interface where services are going to bind, do so using the following command:

  ```
  docker run -d -e IP=<ip address> <dse_image_name>
  ```

- If using Docker host networking isn't an option the IP addresses to be used can be listed by the following command:

  ```
  docker inspect --format '{{ .NetworkSettings.IPAddress }}'
  CONTAINER_ID
  ```

- To persist data, configuration and logs across container restarts, pre-create directories and map them via the Docker run command as shown below:

  Assuming the directories for data, conf and logs are

  ```
    i.   <some root dir>/<dse_image_name>-data for data
   ii.   <some root dir>/<dse_image_name>-conf for configuration
  iii.   <some root dir>/<dse_image_name>-logs for logs
  ```

  These can be mapped to the Docker instance as follows:

  ```
  docker run –v <some root dir>/<dse_image_name>-data:/data
             –v <some root dir>/<dse_image_name>-conf:/conf
             –v <some root dir>/<dse_image_name>-logs:/logs
             –d <dse_image_name>
  ```

- To provide cluster specific configuration, the following environment variables should be provided via the `Docker run` command:

  a. `CLUSTER_NAME:` the name of the cluster to create/connect to
  b. `SEEDS:` the comma-separated list of seed IP addresses, `e.g. SEEDS=127.0.0.2,127.0.0.3`

  Please refer to *Appendix 2 – Example Script to Start an Entire Cluster for* an example.

[10] For the purposes of this document it is assumed that the OpsC agent will run within the same container as rest of the DSE processes

[11] This is especially relevant for production deployments

- Once a cluster is up and running various tools used to manage DSE such as `nodetool`, `cqlsh`, or `dsetool` can be run with the `docker exec` command as noted below

  ```
  docker exec -it CONTAINER_ID nodetool ring
  ```

Important Caveats

- Data volumes are required for the `commitlog, saved_caches, and data` directories (everything in `/var/lib/cassandra`). The data volume must use a supported file system (usually `xfs or ext4`).
- Docker's default networking (via Linux bridge) is not recommended for the production use as it slows down networking considerably. Instead, use the host networking (`docker run --net=host`) or a plugin that can manage IP ranges across clusters of hosts. The host networking limits the number of DSE nodes per a Docker host to one, but this is the recommended configuration to use in production.
- Development and testing benefit from running DSE clusters on a single Docker host and for such scenarios the default networking is just fine. Use pipework or Weave if consistent IP address allocation is needed.
- Before running DSE within Docker (especially in production); it's prudent that various configuration options are adjusted for the Docker environment (for e.g. fine tuning of `dse.yaml`, `cassandra.yaml`, turning `swap` off on the Docker host, choosing where to manage Cassandra data, calculating optimal JVM heap size, choosing optimal garbage collector, etc).
- The default capability limits of Docker containers break `mlockall` functionality that Cassandra uses to prevent swapping and page faults. The simplest workaround is to add `-XX:+AlwaysPreTouch` to the JVM arguments and disable swap on the host OS.
- All containers by default inherit `ulimits` from the Docker daemon. DSE containers should have them set to unlimited or reasonably high values (for e.g. for `mem_locked_memory and max_memory_size`).
- There is no support for rack / data center placement for Dockerized nodes. To enable this capability, please refer to this example.
- Token ranges need to be calculated post run (by exposing a node's configuration via mounted volumes).
- Whenever a node starts up it will place its IP address within the `/data/ip.address`. This can be used to set of `SEEDS` to spawn nodes at a later time.
- Certain global state is kept within Cassandra tables (e.g. `system.peers`, cluster name, etc.). It is not advisable to change any of these whenever a new container is started which re-uses existing data files.
- Upgrading of DSE or OpsC software should follow similar procedures as documented in DataStax Enterprise documentation with the additional requirement of restarting the Docker instance in between upgrades.
- While Docker does provide as close to bare metal performance as possible, the overheads are certainly not zero. Hence, appropriate capacity must be planned for the necessary overhead that may occur.
- While the DataStax product teams validated best practices listed in this document, adaptations of these instructions may be necessary depending on the deployment under consideration. Hence, it is also highly recommended to execute rigorous tests for the use cases under consideration before deploying a Dockerised DSE installation within production.

DSE/Docker Futures

Deploying DSE within Docker isn't trivial, but with adequate guidance and pre-production validation, it's not that difficult. As the container ecosystem evolves, it is expected that future DSE releases will have additional guidelines to make the most of DSE installations under Docker. Some future areas that DataStax is investigating are:

- Further splitting up of DSE processes into separate containers (e.g. running Spark executors and DSE core JVM within a single container, and all other DSE processes within a separate containers)
- Integration of container based deployment with workload management infrastructure components such as Kubernetes, Mesos, etc.
- Enabling the deployment model on a variety of public and private clouds

Conclusion

DataStax acknowledges that containers have rapidly become one of the building blocks for becoming an Internet Enterprises. In this paper, we have attempted to provide tips, guidelines and examples to reduce the amount of time required to run DSE in Docker. For more resources and downloads of DataStax Enterprise, visit www.datastax.com today.

About DataStax

DataStax delivers Apache Cassandra™ in a database platform purpose built for the performance and availability demands of web, mobile and IoT applications, giving enterprises a secure always-on database that remains operationally simple when scaled in a single datacenter or across multiple datacenters and clouds.

With more than 500 customers in over 50 countries, DataStax is the database technology of choice for the world's most innovative companies, such as Netflix, Adobe, Intuit and eBay. Based in Santa Clara, Calif., DataStax is backed by industry-leading investors including Comcast Ventures, Crosslink Capital, Lightspeed Venture Partners, Kleiner Perkins Caufield & Byers, Meritech Capital, Premji Invest and Scale Venture Partners. For more information, visit DataStax.com or follow us @DataStax.

Appendix 1 – Example Docker File

Below is an example Docker file (and necessary utilities) for your reference. It is highly recommended that the contents of these be adjusted for the deployment under consideration[12].

There are 3 steps to running this example.

1. Create the `dse-entrypoint` file in the same directory as your docker file; to set up necessary directories, configuration files, and also enable starting of the DSE process when you run:

   ```
   docker exec -it <CONTAINER_ID> …
   ```

dse-entrypoint

```
#!/bin/sh

# Bind the various services
# These should be updated on every container start
if [ -z ${IP} ]; then
    IP=`hostname --ip-address`
Fi

echo $IP > /data/ip.address

# create directories for holding the node's data, logs, etc.
create_dirs() {
    local base_dir=$1;

    mkdir -p $base_dir/data/commitlog
    mkdir -p $base_dir/data/saved_caches
    mkdir -p $base_dir/logs
}

# copy the relevant sections of the config for a service (includes bin for
# selected services like cassandra or hadoop)
copy_config() {
    base_dst_dir=$1
    base_dir=$2
    service=$3

    src="$base_dir/resources/$service/conf"
    dst="$base_dst_dir/$service"

    cp -r $src $dst

}

# tweak the cassandra config
tweak_cassandra_config() {
    env="$1/cassandra-env.sh"
    conf="$1/cassandra.yaml"

    base_data_dir="/data"

    # Set the cluster name
    if [ -z "${CLUSTER_NAME}" ]; then
        printf "      - No cluster name provided; skipping.\n"
```

[12] Note that these scripts rely on common dependencies, which aren't explicit listed.

```
    else
        printf "      - Setting up the cluster name: ${CLUSTER_NAME}\n"
        regexp="s/Test Cluster/${CLUSTER_NAME}/g"
        sed -i -- "$regexp" $conf
    fi

    # Set the commitlog directory, and various other directories
    # These are done only once since the regexep matches will fail on subsequent
    # runs.
    printf "      - Setting up directories\n"
    regexp="s|/var/lib/cassandra/|$base_data_dir/|g"
    sed -i -- "$regexp" $conf


    regexp="s/^listen_address:.*/listen_address: ${IP}/g"
    sed -i -- "$regexp" $conf

    regexp="s/rpc_address:.*/rpc_address: ${IP}/g"
    sed -i -- "$regexp" $conf

    # seeds
    if [ -z "${SEEDS}" ]; then
        printf "      - Using own IP address ${IP} as seed.\n";
        regexp="s/seeds:.*/seeds: \"${IP}\"/g";
    else
        printf "      - Using seeds: $SEEDS\n";
        regexp="s/seeds:.*/seeds: \"${IP},${SEEDS}\"/g"
    fi
    sed -i -- "$regexp" $conf

    # JMX
    echo "JVM_OPTS=\"\$JVM_OPTS -Djava.rmi.server.hostname=127.0.0.1\"" >> $env
}

tweak_spark_config() {
    sed -i -- "s|/var/lib/spark/|/data/spark/|g" $1/spark-env.sh
    sed -i -- "s|/var/log/spark/|/logs/spark/|g" $1/spark-env.sh
    mkdir -p /data/spark/worker
    mkdir -p /data/spark/rdd
    mkdir -p /logs/spark/worker
}

tweak_agent_config() {
       [ -d "$OPSC_ADDR_DIR" ] && cat > $OPSC_ADDR_DIR/address.yaml <<EOF
       cassandra_conf: /conf/cassandra/cassandra.yaml
       local_interface: ${IP}
       hosts: ["${IP}"]
       cassandra_install_location: /opt/dse
       cassandra_log_location: /logs
       EOF
}

setup_resources() {
    printf "    - Copying configs\n"
    copy_config $1 $2 "cassandra"
    copy_config $1 $2 "dse"
    copy_config $1 $2 "hadoop"
    copy_config $1 $2 "pig"
    copy_config $1 $2 "hive"
    copy_config $1 $2 "sqoop"
    copy_config $1 $2 "mahout"
    copy_config $1 $2 "spark"
    copy_config $1 $2 "shark"
    copy_config $1 $2 "tomcat"
}

setup_node() {
    printf "* Setting up node...\n"
    printf "  + Setting up node...\n"

    create_dirs
    setup_resources "/conf" ${DSE_HOME}
    tweak_cassandra_config "/conf/cassandra"
    tweak_spark_config "/conf/spark"
    tweak_agent_config
    chown -R dse:dse /data /logs /conf
    printf "Done.\n"
}

# if conf dir is empty, generate config
[ -z "$(ls -A $CONF_DIR)" ] && setup_node

exec gosu dse $DSE_HOME/bin/dse cassandra -f "$@"
```

2. Create `dse-cmd-launcher` in the same directory as your docker file which ensures that DataStax Enterprise and related tools are run with the `dse` user, making them available from

    ```
    docker run -d --name <container_name> <image_name> …
    ```

dse-cmd-launcher

```bash
#!/bin/bash

cmd=$(basename $0)

# prefer the command in DSE_HOME/bin
full_cmd_path=$(find -L $DSE_HOME/bin $DSE_HOME -type f -name $cmd | head -1)

export CQLSH_HOST="$(hostname --ip-address)"

exec gosu dse $full_cmd_path "$@"
```

3. Create and build the docker file

   ```
   docker build -t <image_name> .
   ```

Dockerfile

```
FROM azul/zulu-openjdk:8

RUN export DEBIAN_FRONTEND=noninteractive && \
    apt-get update && \
    apt-get -y install adduser \
    curl \
    lsb-base \
    procps \
    zlib1g \
    gzip \
    python \
    python-support \
    sysstat \
    ntp bash tree && \
    rm -rf /var/lib/apt/lists/*

# grab gosu for easy step-down from root
RUN curl -o /bin/gosu -SkL "https://github.com/tianon/gosu/releases/download/1.4/gosu-$(dpkg --print-architecture)" \
    && chmod +x /bin/gosu

# tarball can be download into the folder where Dockerfile is
# wget --user=$USER --password=$PASS http://downloads.datastax.com/enterprise/dse-4.8.0-bin.tar.gz
# you may want to replace dse-4.8.0-bin.tar.gz with the corresponding downloaded package name
ADD dse-4.8.0-bin.tar.gz /opt

# keep data here. Optionally bind directly to a volume on the server. I.E.:
#VOLUME /data /data
VOLUME /data

ENV DSE_HOME /opt/dse
ENV DSE_ENV $DSE_HOME/bin/dse-env.sh
ENV CONF_DIR /conf

ENV DSE_CONF $CONF_DIR/dse
ENV CASSANDRA_CONF $CONF_DIR/cassandra
ENV HADOOP_CONF_DIR $CONF_DIR/hadoop
ENV TOMCAT_CONF_DIR $CONF_DIR/tomcat
ENV HIVE_CONF_DIR $CONF_DIR/hive
ENV SPARK_CONF_DIR $CONF_DIR/spark

RUN ln -s /opt/dse* $DSE_HOME

# keep configs here
VOLUME /conf
```

```
# and logs here
VOLUME /logs

# point C* logs dir to the created volume
RUN sed -i -- "s|/var/log/cassandra|/logs|g" $DSE_HOME/bin/dse.in.sh

# create a dedicated user for running DSE node
RUN groupadd -g 1337 dse && \
    useradd -u 1337 -g dse -s /bin/bash -d $DSE_HOME dse && \
    chown -R dse:dse /opt/dse*

VOLUME /opt/dse

# starting node using custom entrypoint that configures paths, interfaces, etc.
COPY dse-entrypoint /usr/local/bin/
RUN chmod +x /usr/local/bin/dse-entrypoint
ENTRYPOINT ["/usr/local/bin/dse-entrypoint"]

# Running any other DSE/C* command should be done on behalf dse user
# Perform that using a generic command launcher
COPY dse-cmd-launcher /usr/local/bin/
RUN chmod +x /usr/local/bin/dse-cmd-launcher


# link dse commands to the launcher
RUN for cmd in cqlsh dsetool nodetool dse cassandra-stress datastax-agent ; do \
        ln -sf /usr/local/bin/dse-cmd-launcher /usr/local/bin/$cmd ; \
  done

# the detailed list of ports
# http://docs.datastax.com/en/datastax_enterprise/4.7/datastax_enterprise/sec/secConfFirePort.html

# needed for datastax-agent
RUN locale-gen en_US en_US.UTF-8 && \
    ln -s /opt/dse/datastax-agent/conf /etc/datastax-agent

ENV OPSC_ADDR_DIR /opt/dse/datastax-agent/conf

# Cassandra
EXPOSE 7000 9042 9160

# Solr (assuming DSE Max)
EXPOSE 8983 8984

# Spark (assuming DSE Max)
EXPOSE 4040 7080 7081 7077

# Hadoop (assuming DSE Max)
EXPOSE 8012 50030 50060 9290

# Hive/Shark
```

```
EXPOSE 10000

# OpsCenter agent
EXPOSE 61621
```

Appendix 2 – Example Script to Start an Entire Cluster

```
#!/bin/bash

IMAGE=$1
NUM_NODES=$2
NODE_OPTS=$3

 [ -z "$CLUSTER_NAME" ] && CLUSTER_NAME="Test_Cluster"

docker run -d -e CLUSTER_NAME="$CLUSTER_NAME" --name node1 $IMAGE $NODE_OPTS

SEEDS=$(docker inspect --format '{{ .NetworkSettings.IPAddress }}' node1)

let n=1

while [ $n != $NUM_NODES ]; do
         let n=n+1
         docker run -d -e SEEDS=$SEEDS -e CLUSTER_NAME="$CLUSTER_NAME" --name node${n} $IMAGE
$NODE_OPTS
done
```

To start a 3-node DSE Search cluster run the above script with the following options –

```
        <script_name> <dse_image_name> 3 -s
```

If you want to run DSE Analyitcs nodes run –

```
        <script_name> <dse_image_name> 3 -k
```