GVA2020 - Class Review

Overview:

While the course home page serves as an organizational template for what you have done in the course, there are always going to be differences between what was on the webpage and what you did. Here we will create a document that will highlight exactly what tutorials you did to go along with the home page. Additionally, throughout the course you have been running anything of substance (ie programs and scripts) on iDev nodes, but as mentioned several times, using the idev nodes is not how you will typically want to interact with TACC.

Objectives:

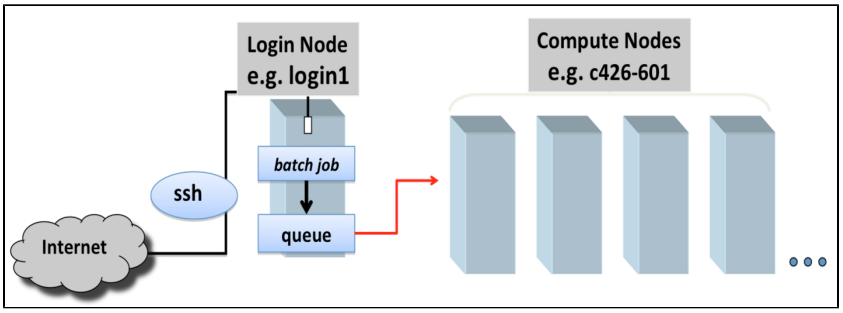
This tutorial aims to:

1. Review TACC's job submission architecture
2. Generate a commands file that will collect a list of what you

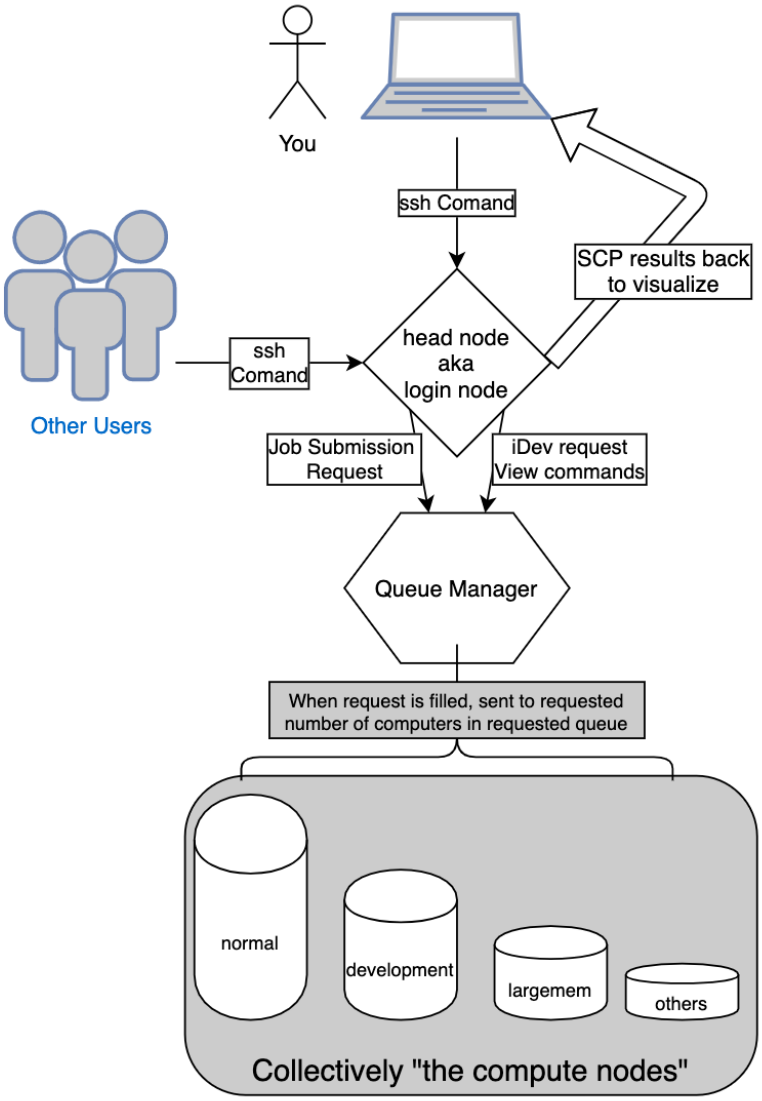
TACC Architecture:

In Thursday's class, there was some impromptu discussion/confusion about differences between the different types of compute nodes (development vs normal) and how the reservation we've been using this week related to the different types of nodes. The LS5 user guide includes some great information that tries to speak to people both well versed in computer science and those that aren't. They provide the following figure to describe how the resources at ls5 are laid out which while accurate and helpful doesn't describe 3 key things that are always helpful to think about:

1. How the head node or login node are shared between all users
2. Explain the different queues that are available and how their choice effects you
3. Differences between idev sessions and submitted jobs



After some internal debate, I decided to generate what I'm considering to be a similar figure that highlights the things that I think are important (specifically the 3 things I feel the above doesn't convey), and label things with the terms I use to think about them. As mentioned several times, my background is biology, not computer science, and ultimately I think slightly inaccurate descriptive terms may help clear up some of the confusion.



Different Queues

The LS5 user guide has detailed information about the queues and what the limits are, but similar to the figures above, as a biologist the information in those tables isn't really what I tend to think about. Instead I think about the queues as follows:

Queue	Max Time	Purpose/when to use
normal	48	this is my default queue, while 48 hours is the max time, in the history of LS5 I can only think of 1 instance I have ever needed to request more than 24 hours. Remember I work mostly with bacteria and up to 100s of millions of reads across dozens-400 samples at a time typically so your experience may vary.
development	2	Short jobs that I need output from, to start a larger job. In particular, read trimming (always) and FastQC/MultiQC (if more than 25ish samples or 50ish millions of reads)
largemem	48	Recently working with collaborator on sequencing project with very deep coverage, majority of reads related to a host organism, and minority of reads related to a poorly characterized bacteria. This required a the largemem node for a complex mapping command, and subsequent assembly attempt.
"other"	?	I know there are other queues that are available, but have never had need to figure out what they are or found nodes above to be lacking. As described in the read QC tutorial with not knowing about the '**`zgrep`**' command it is possible or even probable that using one of the other queues would be more efficient but I doubt it.

idev Vs submitted job

Submitting a job to the queue is my (and should be your) default position for several reasons:

1. Nearly all of the jobs I run take at least 30-60 minutes when running a reasonable number of samples, and most take at least double that. During that time, nothing happens that needs my intervention or input, the computer does the work for me. So running in an interactive mode where I can see output in real time would be a terrible use of my time.
2. I make routine use of redirecting the output and errors to log files with the **>&** command used in several of the advanced tutorials (Breseq, trimmo matic and maybe others) so I can view information from the run after the run is done or has had an error. (more on the **>&** below)
3. I am *always* working with multiple samples, and running multiple samples at the same time on a single node. This means even if I wanted to see what was going on, I wouldn't be able to decipher what lines came from what specific samples.

There are times that I do use idev nodes:

1. Jobs that I have submitted, keep failing early, but not immediately.
2. Job doesn't seem to be launching correctly as jobs.
3. I have several jobs to run in a row that I know will take <5 minutes each, but require sequential intervention.
    a. Specifically a pipeline for a bunch of plasmid sequencing the lab does that the analysis has 7 steps including: read trimming, breseq analysis, compare table generation, spades alignment, spades comparison, overall run statistics, and organization of results into different groups for individual researchers in the lab.
4. Working with a new analysis program and **downsampled** data still triggers a TACC warning to get off the head node. (Warning message shown below)

If idev is so rare why did we use it in the class:

First, running in interactive mode gives you some comparison of how different programs work. Think back on your runs this week, what programs /commands printed information to the screen that had useful information (read trimming, read mapping come to my mind), what programs didn't (mpileup, indexing, file conversions), and what programs had large amounts of information but wasn't directly needed or evaluated (breseq). This may help you decide when there are things you want to capture and when you should expect empty files.

Second, it speeds the class up. No matter what output is generated when you execute a command, you get your command prompt back when the command is done, and you can immediately interrogate the results rather than waiting and guessing when your job starts, and when it finishes.

Finally, throughout the course we made use of the reservation system which allowed us to skip the queue and immediately get an idev session or job running. In previous years where reservations weren't possible tutorials were planned around a:

- "hurry up and get the job started its going to sit for some amount of time in the queue"
- "ok let me tell you about those commands that are sitting around waiting to run"
- "DRAT! there is a typo in your commands file edit that command and go back to the end of the queue while we talk about the results you can't actually see"

I hope you can see that using idev nodes has enabled each of you to accomplish more tutorials than previous years while hopefully learning more. Generally, the feedback from students who have taken the course under this type of format has been positive, so if you find that you are overly reliant on idev nodes or have a hard time transitioning to submitting jobs, I'd love to hear the feedback so I can attempt to modify things further.

Good citizenship on the head node

When you log into lonestar using **ssh** you are connected to what is known as the login node or "the head node". There are several different head nodes, but they are shared by everyone that is logged into lonestar (not just in this class, or from campus, or even from texas, but everywhere in the world). Anything you type onto the command line has to be executed by the head node. The longer something takes to complete, or the more it will slow down you and everybody else. Get enough people running large jobs on the head node all at once (say a classroom full of summer school students) and lonestar can actually crash leaving nobody able to execute commands or even log in for minutes -> hours -> even days if something goes really wrong. To try to avoid crashes, TACC tries to monitor things and proactively stop things before they get too out of hand. If you guess wrong on if something should be run on the head node, you may eventually see a message like the one pasted below. If you do, its not the end of the world, but repeated messages will become revoked TACC access and emails where you have to explain what you are doing to TACC and your PI and how you are going to fix it and avoid it in the future.

Example of how you learn you shouldn't have been on the head node

```
Message from root@login1.ls4.tacc.utexas.edu on pts/127 at 09:16 ...
Please do not run scripts or programs that require more than a few minutes of
CPU time on the login nodes.  Your current running process below has been
killed and must be submitted to the queues, for usage policy see
http://www.tacc.utexas.edu/user-services/usage-policies/
If you have any questions regarding this, please submit a consulting ticket.
```

Recall this is the type of message that I have gotten when working with downsampled data on the head node and lead me to start an idev session to figure out what is going on.

Submitting jobs to the queue

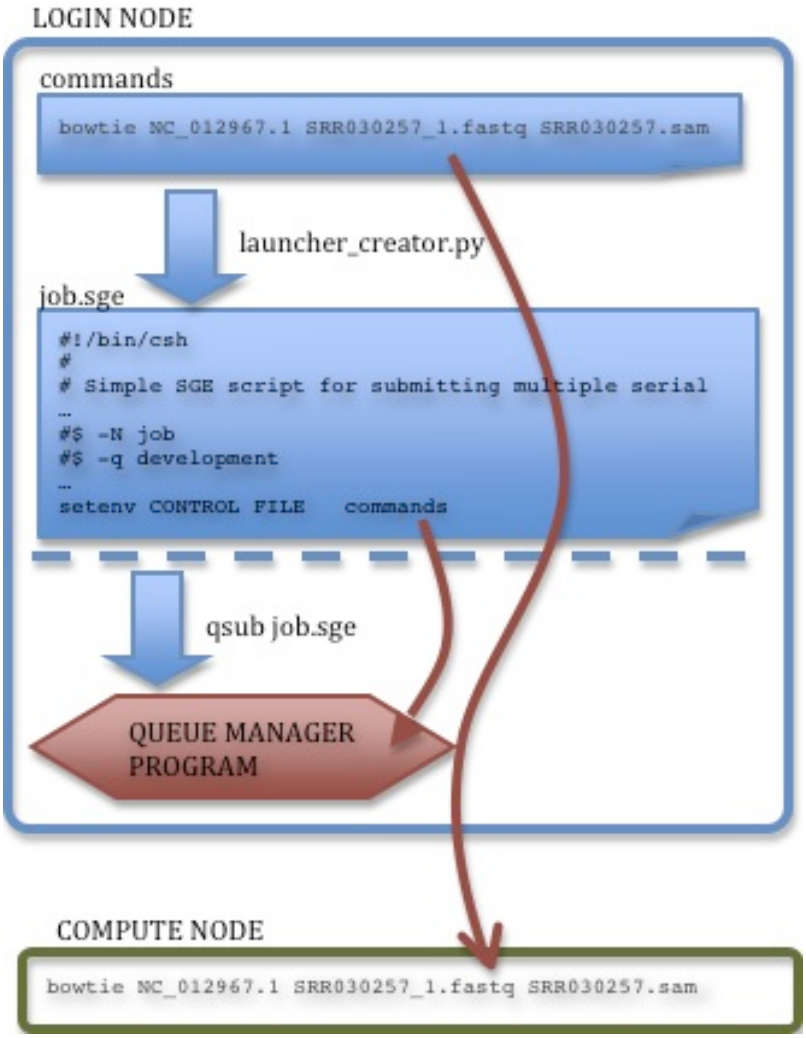
Every job you submit will have 2 parts:

1. A file with the commands you want to run
2. A control file that interacts with the queue system to do all the computer science stuff of executing the commands.

Access to nodes (regardless of what queue they are in) is controlled by a "Queue Manager" program. You can personify the Queue Manager program as: Heimdall in Thor, a more polite version of Gandalf in lord of the rings when dealing with with the balrog, the troll from the billy goats gruff tail, or any other "gatekeeper" type. Regardless of how nerdy your personification choice is, the Queue Manager has an interesting caveat: you can only interact with it using the **sbatch** command. "sbatch filename.slurm" tells the que manager to run a set job based on information in filename.slurm (i.e. how many nodes you need, how long you need them for, how to charge your allocation, etc). The Queue manager doesn't care **WHAT** you are running, only **HOW** you want to run it.

The following figure shows:

1. A commands file with a single bowtie command .
2. The launcher_cerator.py script generating a job.sge file.
    a. A sge file is analogous to a slurm file on different TACC systems.
3. Submitting that job.sge file to the que manager program with a qsub command
    a. This again is analogous to submitting a slurm file with sbatch on LS5.



The easiest way to deal with slurm files is using the launcher_creator.py script. In some cases I have copied slurm files back to my $HOME directory when I anticipate changing the information in it in specific ways in the future (such as changing the number of samples I will run through breseq)

Using `launcher_creator.py`

Running the launcher_creator.py script with the `-h` option to show the help message so we can see what other options the script takes:

How to display all available options of the launcher_createor.py script. Hopefully the idea of using a -h option to a program is second nature to you after this week.

```
launcher_creator.py -h
```

Short option	Long option	Required	Description
-n	name	Yes	The name of the job.
-t	time	Yes	Time allotment for job, format must be hh:mm:ss.
-j	job	j and/or b must be used	Filename of list of commands to be distributed to nodes.
-b	bash commands	j and/or b must be used	*Optional* String of Bash commands to execute before commands are distributed
-q	queue	Default: Development	The queue to submit to, like 'normal' or 'largemem', etc.
-a	allocation		The allocation you want to charge the run to.
-m	modules		*Optional* String of module management commands semi colon separated. ie "module load bowtie2; module load fastqc"
-w	wayness		*Optional* The number of jobs in a job list you want to give to each node. (Default is 12 for Lonestar, 16 for Stampede.)
-N	number of nodes		*Optional* Specifies a certain number of nodes to use. You probably don't need this option, as the launcher calculates how many nodes you need based on the job list (or Bash command string) you submit. It sometimes comes in handy when writing pipelines.
-e	email		*Optional* Your email address if you want to receive an email from Lonestar when your job starts and ends.
-l	launcher		*Optional* Filename of the launcher. (Default is `<name>.sge`)
-s	stdout		*Optional* Setting this flag outputs the name of the launcher to stdout.

The lines highlighted in green and yellow are what you should focus on:

1. **Job**: Not required by the script, but is what I always use. This file has a list of commands that you want to run, and is probably biggest part of what makes TACC great... letting you run a large number of commands all at once rather than having to run them 1 at a time on your own computer.
2. **Time and name**: They are required so obviously important for that reason. Also names are important as described on Monday: naming everything 'tacc_job' will make your life much more confusing at some point than using descriptive names like 'breseq_run_3-13-2020-genomes'.
3. **Wayness**: 12 is the default, 48 is the max on lonestar.
    a. This is a balance of memory and to a lesser extent speed against how long you want to wait for your job to run and SU cost.
    b. Imagine you have 96 commands you want to run. 12, 24, and 48 as the "w" option will require 8, 4, and 2 remote computers to run at the same time, 2 computers are typically available sooner than a set of 4 and 4 are available sooner than a set of 8. So your job is likely to spend less time 'in the queue' with larger numbers.
    c. We'll go over "SUs" below.
    d. For bacterial work, 48 is a much better choice in nearly all situations, unless you are experiencing problems or know you have a massive amount of data.
    e. The downside is if you pick a number that is too small, your commands may have errors and not actually produce results requiring you to start over with longer requests, or smaller "w" values
4. **queue**: As noted above, "normal" is a better choice in most all cases in my experience.

Commands files

How does Dan make commands files?

1. In several tutorials (MultiQC, Trimmomatic, Advanced Breseq) I gave example command line for loops that can be used to generate commands files for large numbers of samples.
2. Often I use a python script to generate files like this, as I am more fluent in python coding than I am in bash/command line scripting.
3. Sometimes I generate some commands in Excel
    a. An example might be a formula of `'="breseq -j 6 -p -r Ref.gbk -o Run_output/" & A1 & " " & B1 & " " & C1 & " >& runLogs/" & A1'` with sample name in A1, read1 in B1 and read2 in C1.
4. In a program like text wrangler/notepad to copy paste the same command multiple times and change it as appropriate then paste the text directly into a nano editor on TACC

Use of the '>&' symbology

Adding >& symbol to the end of a command followed by a file name is very useful. This results in redirecting what would normally print to the screen as either the standard output and the standard error streams to a file that follows. This can be very useful for determining where/why a particular sample failed as it sorts it into a specific file based on the sample being analyzed. This behavior is demonstrated in the trimmoatic tutorial as well as the advanced breseq tutorial.

The same information will be found in the .o<jobID> and .e<jobID>, but the output and error streams will be mixed together among all simultaneously running commands.

Tutorial: Running a job

Now that we have an understanding of what the different parts of running a job is, let's actually run a job. Our goal of this sample job will be to provide you with something to look back on and remember what you did while you were here. As a safety measure, you can not submit jobs from inside an idev node (similarly you can not run a commands file that submits new jobs on the compute nodes). So check if you are on an idev node (**showq -u** or **hostname**), and if you are on an idev node, **logout** before continuing.

Get some data

For this tutorial, the data that you will be using will be the tutorials you have already completed in the class. Similar to the "first job" you put together on the first day this "last day" job will generate a text file to remind you of what you did rather than highlighting basic commands.

Setting up your run

Navigate to the $SCRATCH directory before doing the following.

Commands to get to the where the data is and launch nano editor

```
cds  # move to your scratch directory
nano commands
```

Sample commands that can be pasted into nano

```
echo "My name is _____ and todays date is:" > GVA2020.output.txt
date >> GVA2020.output.txt
echo "I have just demonstrated that I know how to redirect output to a new file, and to append things to an
already created file. Or at least thats what I think I did" >> GVA2020.output.txt
echo "i'm going to test this by counting the number of lines in the file that I am writing to. So if the next
line reads 4 I remember I'm on the right track" >> GVA2020.output.txt
wc -l GVA2020.output.txt >> GVA2020.output.txt
echo "I know that normally i would be typing commands on each line of this file, that would be executed on a
compute node instead of the head node so that my programs run faster, in parallel, and do not slow down others
or risk my tacc account being locked out" >> GVA2020.output.txt
echo "i'm currently in my scratch directory on lonestar. there are 2 main ways of getting here: cds and cd
$SCRATCH" >>GVA2020.output.txt
pwd >> GVA2020.output.txt
echo "over the last week I've conducted multiple different types of analysis on a variety of sample types and
under different conditions. Each of the exercises was taken from the website https://wikis.utexas.edu/display
/bioiteam/Genome+Variant+Analysis+Course+2020" >> GVA2020.output.txt
echo "using the ls command i'm now going to try to remind you (my future self) of what tutorials I did" >>
GVA2020.output.txt
ls -1 >> GVA2020.output.txt
echo "the contents of those directories (representing the data i downloaded and the work i did) are as follows:
">> GVA2020.output.txt
find . >> GVA2020.output.txt
echo "the commands that i have run on the headnode are: " >> GVA2020.output.txt
history >> GVA2020.output.txt
echo "the contents of this, my commands file, which i will use in the launcher_creator.py script are:
">>GVA2020.output.txt
cat commands >> GVA2020.output.txt
echo "finally, I will be generating a job.slurm file using the launcher_creator.py script using the following
command:" >> GVA2020.output.txt
echo 'launcher_creator.py -w 1 -N 1 -n "what_i_did_at_GVA2020" -t 00:15:00 -a "UT-2015-05-18" -j commands' >>
GVA2020.output.txt
echo "this will create a what_i_did_at_GVA2020.slurm file that will run for 15 minutes" >> GVA2020.output.txt
echo "and i will send this job to the queue using the the command: sbatch what_i_did_at_GVA2020.slurm" >>
GVA2020.output.txt
```

THESE are not typed, these are the keys to press to save the file after you have entered it in nano

```
ctrl + o
# enter to save
ctrl + x
```

Use wc -l command to verify the number of lines in your commands file.

```
wc -l commands
```

If you get a number larger than 21 edit your commands file with nano so each command is a single line as they appear above. Several of the lines are likely long enough that they will wrap when you paste them in nano and cause problems

Sample commands that can be pasted into nano

```
launcher_creator.py -w 1 -N 1 -n "what_i_did_at_GVA2020" -t 00:15:00 -a "UT-2015-05-18" -j commands -m "module
load launcher/3.0.3"
sbatch what_i_did_at_GVA2020.slurm
```

Evaluating your job submission

Based on our example you may have expected 1 new file to have been created during the job submission (**GVA2020.output.txt**), but instead you will find 2 extra files as follows: **what_i_did_at_GVA2020.e(job-ID)**, and **what_i_did_at_GVA2020.o(job-ID)**. When things have worked well, these files are typically ignored. When your job fails, these files offer insight into the why so you can fix things and resubmit.

Many times while working with NGS data you will find yourself with intermediate files. Two of the more difficult challenges of analysis can be trying to decide what files you want to keep, and remembering what each intermediate file represents. Your commands files can serve as a quick reminder of what you did so you can always go back and reproduce the data. Using arbitrary endings (.out in this case) can serve as a way to remind you what type of file you are looking at. Since we've learned that the scratch directory is not backed up and is purged, see if you can turn your intermediate files into a single final file using the cat command, and copy the new final file, the .slurm file you created, and the 3 extra files to work. This way you should be able to come back and regenerate all the intermediate files if needed, and also see your final product.

make a single final file using the cat command and copy to a useful work directory

```
# remember that things after the # sign are ignored by bash
cat GVA2019.output.txt > end_of_class_job_submission.final.output
mkdir $WORK/GVA2019
mkdir $WORK/GVA2019/end_of_course_summary/  # each directory must be made in order to avoid getting a no such
file or directory error
cp end_of_class_job_submission.final.output $WORK/GVA2019/end_of_course_summary/
cp what_i_did* $WORK/GVA2019/end_of_course_summary/  # note this grabs the 2 output files generated by tacc
about your job run as well as the .slurm file you created to tell it how to run your commands file

cp commands $WORK/GVA2019/end_of_course_summary/
```

Interrogating the launcher queue

Here are some of the common commands that you can run and what they will do or tell you:

Command	Purpose	Output(s)
showq -u	Shows only your jobs	Shows all of your currently submitted jobs, a state of: "qw" means it is still queued and has not run yet "r" means it is currently running
scancel <job-ID>	Delete a submitted job before it is finished running **note**: you can only get the job-ID by using showq -u	There is no confirmation here, so be sure you are deleting the correct job. There is nothing worse than deleting a job that has sat a long time by accident because you forgot something on a job you just submitted.
showq	You are a nosy person and want to see everyone that has submitted a job	Typically a huge list of jobs, and not actually informative

There will be something you use quite often working on your own data.

SUs on TACC and allocations

SUs or "service units" are a currency TACC uses to control access and encourage everyone to engage in the best behaviors. For information on applying for an allocation at TACC, visit https://portal.tacc.utexas.edu/allocations-overview. It is my understanding (which may be flawed) that this is a very easy process for UT faculty, and possibly more difficult for external academic researchers, but still be possible potentially requiring you to go through https://portal.xsede.org/submit-request#/login which is linked from the allocations overview page above. In my experience, the people at TACC are awesome at helping people through these types of processes and ticket requests through the TACC user portal are a great way to start the process.

Each job (including idev sessions) uses SUs at a given rate. SUs are deducted from your balance based on the time, type, and number of nodes you *occup*
, NOT the time requested. More information can be found here. This represents an additional reason submitting a job is better practice than idev nodes most of the time: they cost less. Not because they are charged differently, but rather, a submitted job is charged based on exactly how long the commands take to execute while idev sessions have waiting between executing commands and you must remember to log out at the end of it to stop the billing.

Return to GVA2020 to work on any additional tutorials you are interested in.