A better option is to use a container block:

```
with tf.container("my_problem_1"):
    [...] # Construction phase of problem 1
```

This will use a container dedicated to problem #1, instead of the default one (whose name is an empty string `""`). One advantage is that variable names remain nice and short. Another advantage is that you can easily reset a named container. For example, the following command will connect to the server on machine A and ask it to reset the container named `"my_problem_1"`, which will free all the resources this container used (and also close all sessions open on the server). Any variable managed by this container must be initialized before you can use it again:

```
tf.Session.reset("grpc://machine-a.example.com:2222", ["my_problem_1"])
```

Resource containers make it easy to share variables across sessions in flexible ways. For example, Figure 12-7 shows four clients running different graphs on the same cluster, but sharing some variables. Clients A and B share the same variable `x` managed by the default container, while clients C and D share another variable named `x` managed by the container named `"my_problem_1"`. Note that client C even uses variables from both containers.

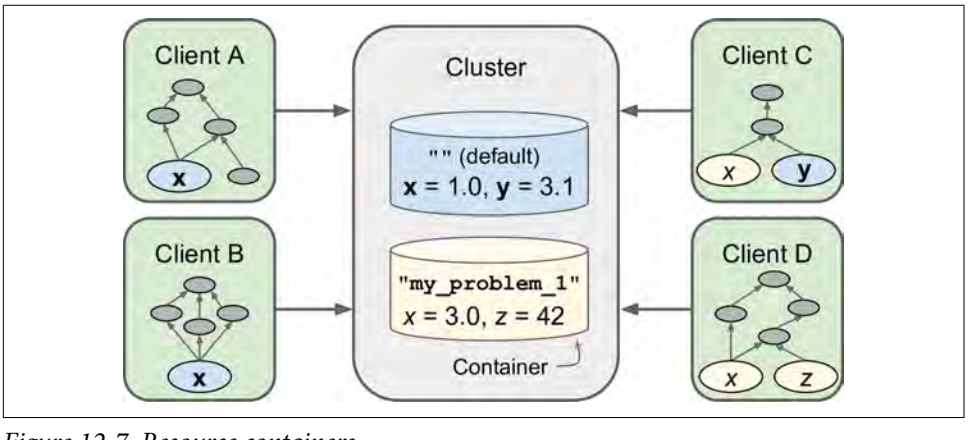

Figure 12-7. Resource containers

Resource containers also take care of preserving the state of other stateful operations, namely queues and readers. Let's take a look at queues first.

Asynchronous Communication Using TensorFlow Queues

Queues are another great way to exchange data between multiple sessions; for example, one common use case is to have a client create a graph that loads the training data and pushes it into a queue, while another client creates a graph that pulls the data from the queue and trains a model (see Figure 12-8). This can speed up training con-

siderably because the training operations don't have to wait for the next mini-batch at every step.

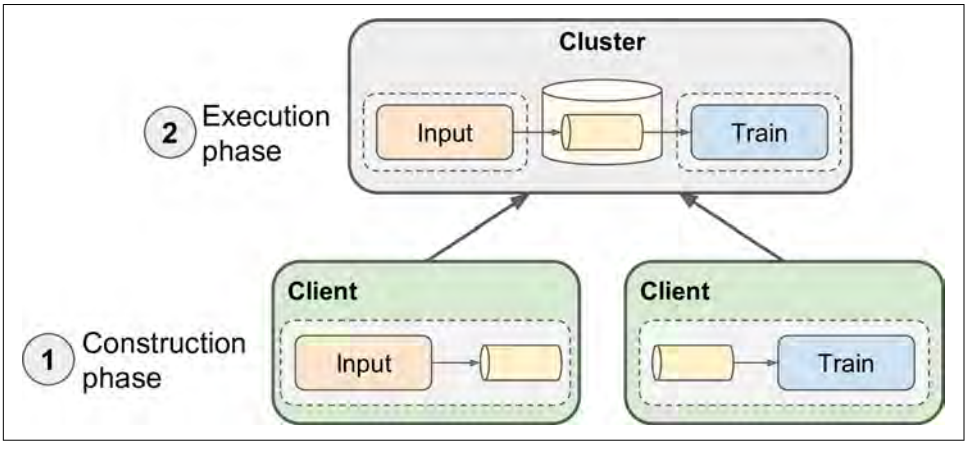

Figure 12-8. Using queues to load the training data asynchronously

TensorFlow provides various kinds of queues. The simplest kind is the *first-in first-out* (*FIFO*) queue. For example, the following code creates a FIFO queue that can store up to 10 tensors containing two float values each:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]],
                 name="q", shared_name="shared_q")
```

To share variables across sessions, all you had to do was to specify the same name and container on both ends. With queues TensorFlow does not use the `name` attribute but instead uses `shared_name`, so it is important to specify it (even if it is the same as the `name`). And, of course, use the same container.

Enqueuing data

To push data to a queue, you must create an `enqueue` operation. For example, the following code pushes three training instances to the queue:

```
# training_data_loader.py
import tensorflow as tf

q = [...]
training_instance = tf.placeholder(tf.float32, shape=(2))
enqueue = q.enqueue([training_instance])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
   sess.run(enqueue, feed_dict={training_instance: [1., 2.]})
   sess.run(enqueue, feed_dict={training_instance: [3., 4.]})
   sess.run(enqueue, feed_dict={training_instance: [5., 6.]})
```

Instead of enqueuing instances one by one, you can enqueue several at a time using an `enqueue_many` operation:

```
[...]
training_instances = tf.placeholder(tf.float32, shape=(None, 2))
enqueue_many = q.enqueue([training_instances])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue_many,
             feed_dict={training_instances: [[1., 2.], [3., 4.], [5., 6.]]})
```

Both examples enqueue the same three tensors to the queue.

Dequeuing data

To pull the instances out of the queue, on the other end, you need to use a `dequeue` operation:

```
# trainer.py
import tensorflow as tf

q = [...]
dequeue = q.dequeue()

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue))  # [1., 2.]
    print(sess.run(dequeue))  # [3., 4.]
    print(sess.run(dequeue))  # [5., 6.]
```

In general you will want to pull a whole mini-batch at once, instead of pulling just one instance at a time. To do so, you must use a `dequeue_many` operation, specifying the mini-batch size:

```
[...]
batch_size = 2
dequeue_mini_batch= q.dequeue_many(batch_size)

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue_mini_batch))  # [[1., 2.], [4., 5.]]
    print(sess.run(dequeue_mini_batch))  # blocked waiting for another instance
```

When a queue is full, the enqueue operation will block until items are pulled out by a dequeue operation. Similarly, when a queue is empty (or you are using `dequeue_many()` and there are fewer items than the mini-batch size), the dequeue operation will block until enough items are pushed into the queue using an enqueue operation.

Queues of tuples

Each item in a queue can be a tuple of tensors (of various types and shapes) instead of just a single tensor. For example, the following queue stores pairs of tensors, one of type `int32` and shape `()`, and the other of type `float32` and shape `[3,2]`:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32], shapes=[[],[3,2]],
                 name="q", shared_name="shared_q")
```

The enqueue operation must be given pairs of tensors (note that each pair represents only one item in the queue):

```
a = tf.placeholder(tf.int32, shape=())
b = tf.placeholder(tf.float32, shape=(3, 2))
enqueue = q.enqueue((a, b))

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={a: 10, b:[[1., 2.], [3., 4.], [5., 6.]]})
    sess.run(enqueue, feed_dict={a: 11, b:[[2., 4.], [6., 8.], [0., 2.]]})
    sess.run(enqueue, feed_dict={a: 12, b:[[3., 6.], [9., 2.], [5., 8.]]})
```

On the other end, the `dequeue()` function now creates a pair of dequeue operations:

```
dequeue_a, dequeue_b = q.dequeue()
```

In general, you should run these operations together:

```
with tf.Session([...]) as sess:
    a_val, b_val = sess.run([dequeue_a, dequeue_b])
    print(a_val) # 10
    print(b_val) # [[1., 2.], [3., 4.], [5., 6.]]
```

If you run `dequeue_a` on its own, it will dequeue a pair and return only the first element; the second element will be lost (and similarly, if you run `dequeue_b` on its own, the first element will be lost).

The `dequeue_many()` function also returns a pair of operations:

```
batch_size = 2
dequeue_as, dequeue_bs = q.dequeue_many(batch_size)
```

You can use it as you would expect:

```
with tf.Session([...]) as sess:
    a, b = sess.run([dequeue_a, dequeue_b])
    print(a) # [10, 11]
    print(b) # [[[1., 2.], [3., 4.], [5., 6.]], [[2., 4.], [6., 8.], [0., 2.]]]
    a, b = sess.run([dequeue_a, dequeue_b])  # blocked waiting for another pair
```

Closing a queue

It is possible to close a queue to signal to the other sessions that no more data will be enqueued:

```
close_q = q.close()

with tf.Session([...]) as sess:
    [...]
    sess.run(close_q)
```

Subsequent executions of `enqueue` or `enqueue_many` operations will raise an exception. By default, any pending enqueue request will be honored, unless you call `q.close(cancel_pending_enqueues=True)`.

Subsequent executions of `dequeue` or `dequeue_many` operations will continue to succeed as long as there are items in the queue, but they will fail when there are not enough items left in the queue. If you are using a `dequeue_many` operation and there are a few instances left in the queue, but fewer than the mini-batch size, they will be lost. You may prefer to use a `dequeue_up_to` operation instead; it behaves exactly like `dequeue_many` except when a queue is closed and there are fewer than `batch_size` instances left in the queue, in which case it just returns them.

RandomShuffleQueue

TensorFlow also supports a couple more types of queues, including `RandomShuffleQueue`, which can be used just like a `FIFOQueue` except that items are dequeued in a random order. This can be useful to shuffle training instances at each epoch during training. First, let's create the queue:

```
q = tf.RandomShuffleQueue(capacity=50, min_after_dequeue=10,
                          dtypes=[tf.float32], shapes=[()],
                          name="q", shared_name="shared_q")
```

The `min_after_dequeue` specifies the minimum number of items that must remain in the queue after a dequeue operation. This ensures that there will be enough instances in the queue to have enough randomness (once the queue is closed, the `min_after_dequeue` limit is ignored). Now suppose that you enqueued 22 items in this queue (floats `1.` to `22.`). Here is how you could dequeue them:

```
dequeue = q.dequeue_many(5)

with tf.Session([...]) as sess:
   print(sess.run(dequeue)) # [ 20.  15.  11.  12.   4.]   (17 items left)
   print(sess.run(dequeue)) # [  5.  13.   6.   0.  17.]   (12 items left)
   print(sess.run(dequeue)) # 12 - 5 < 10: blocked waiting for 3 more instances
```

PaddingFifoQueue

A `PaddingFIFOQueue` can also be used just like a `FIFOQueue` except that it accepts tensors of variable sizes along any dimension (but with a fixed rank). When you are dequeuing them with a `dequeue_many` or `dequeue_up_to` operation, each tensor is padded with zeros along every variable dimension to make it the same size as the largest tensor in the mini-batch. For example, you could enqueue 2D tensors (matrices) of arbitrary sizes:

```
q = tf.PaddingFIFOQueue(capacity=50, dtypes=[tf.float32], shapes=[(None, None)]
                        name="q", shared_name="shared_q")
v = tf.placeholder(tf.float32, shape=(None, None))
enqueue = q.enqueue([v])

with tf.Session([...]) as sess:
   sess.run(enqueue, feed_dict={v: [[1., 2.], [3., 4.], [5., 6.]]})       # 3x2
   sess.run(enqueue, feed_dict={v: [[1.]]})                               # 1x1
   sess.run(enqueue, feed_dict={v: [[7., 8., 9., 5.], [6., 7., 8., 9.]]}) # 2x4
```

If we just dequeue one item at a time, we get the exact same tensors that were enqueued. But if we dequeue several items at a time (using `dequeue_many()` or `dequeue_up_to()`), the queue automatically pads the tensors appropriately. For example, if we dequeue all three items at once, all tensors will be padded with zeros to become 3 × 4 tensors, since the maximum size for the first dimension is 3 (first item) and the maximum size for the second dimension is 4 (third item):

```
>>> q = [...]
>>> dequeue = q.dequeue_many(3)
>>> with tf.Session([...]) as sess:
...     print(sess.run(dequeue))
[[[ 1.  2.  0.  0.]
  [ 3.  4.  0.  0.]
  [ 5.  6.  0.  0.]]

 [[ 1.  0.  0.  0.]
  [ 0.  0.  0.  0.]
  [ 0.  0.  0.  0.]]

 [[ 7.  8.  9.  5.]
  [ 6.  7.  8.  9.]
  [ 0.  0.  0.  0.]]]
```

This type of queue can be useful when you are dealing with variable length inputs, such as sequences of words (see Chapter 14).

Okay, now let's pause for a second: so far you have learned to distribute computations across multiple devices and servers, share variables across sessions, and communicate asynchronously using queues. Before you start training neural networks, though, there's one last topic we need to discuss: how to efficiently load training data.

Loading Data Directly from the Graph

So far we have assumed that the clients would load the training data and feed it to the cluster using placeholders. This is simple and works quite well for simple setups, but it is rather inefficient since it transfers the training data several times:

1. From the filesystem to the client
2. From the client to the master task
3. Possibly from the master task to other tasks where the data is needed

It gets worse if you have several clients training various neural networks using the same training data (for example, for hyperparameter tuning): if every client loads the data simultaneously, you may end up even saturating your file server or the network's bandwidth.

Preload the data into a variable

For datasets that can fit in memory, a better option is to load the training data once and assign it to a variable, then just use that variable in your graph. This is called *preloading* the training set. This way the data will be transferred only once from the client to the cluster (but it may still need to be moved around from task to task depending on which operations need it). The following code shows how to load the full training set into a variable:

```
training_set_init = tf.placeholder(tf.float32, shape=(None, n_features))
training_set = tf.Variable(training_set_init, trainable=False, collections=[],
                           name="training_set")

with tf.Session([...]) as sess:
    data = [...]  # load the training data from the datastore
    sess.run(training_set.initializer, feed_dict={training_set_init: data})
```

You must set `trainable=False` so the optimizers don't try to tweak this variable. You should also set `collections=[]` to ensure that this variable won't get added to the `GraphKeys.GLOBAL_VARIABLES` collection, which is used for saving and restoring checkpoints.

This example assumes that all of your training set (including the labels) consists only of `float32` values. If that's not the case, you will need one variable per type.

Reading the training data directly from the graph

If the training set does not fit in memory, a good solution is to use *reader operations*: these are operations capable of reading data directly from the filesystem. This way the