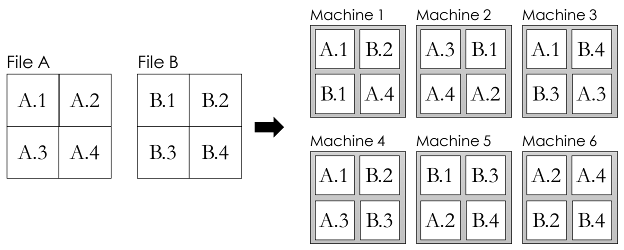
DS 100: Principles and Techniques of Data Science **Date: August 5, 2019**

Discussion #12

Name:

1 Big Data & Distributed File System

1. Consider the following layout of the files `A` and `B` onto a distributed file-system of 6 machines.



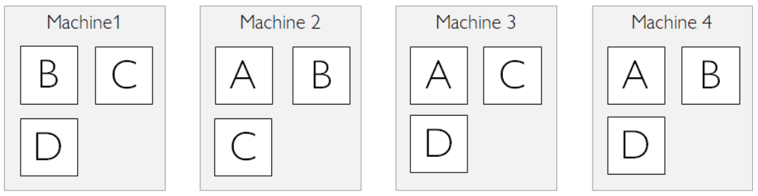
Assume that all blocks have the same file size and computation takes the same amount of time.

(a) If we were to lose machines $M1$, $M2$, and $M3$ which of the following file or files would we lose (select all that apply)?

A. File `A` B. File `B` **C. We would still be able to load both files.**

(b) If each of the six machines fail with probability p, what is the probability that we will lose block $B.1$ of file `B`?

A. $3p$ **B.** p^3 C. $(1-p)^3$ D. $1-p^3$



2. The figure above shows four distinct file blocks labeled A, B, C, and D spread across four machines, where each machine holds exactly 3 blocks.

(a) For the figure above, at most, how many of our machines can fail without any data loss? ___**2**___

(b) Suppose that instead of 4 machines, we have only 3 machines that can store 3 blocks each. Suppose we want to be able to recover our data even if two machines fail. What is the maximum total number of distinct blocks we can store? ___**3**___

(c) Same as part b, but now suppose we only need to be able to recover our data if one machine fails. What is the maximum total number of distinct blocks we can store? ____**4**____

3. As described in class, the traditional data warehouse is a large tabular database that is periodically updated through the ETL process, which combines data from several smaller data sources into a common tabular format. The alternative is a data lake, where data is stored in its original natural form. Which of the following are good reasons to use a data lake approach?

   ☐ The data is sensitive, e.g. medical data or government secrets.

   ☐ To maximize compatibility with commercial data analysis and visualization tools.

   √ **When there is no natural way to store the data in tabular format.**

   ☐ To ensure that the data is clean.

2 Distributed/Parallel Computing & Ray

2.1 Overview

Ray is a distributed execution engine. The same code can be run on a single machine to achieve efficient multiprocessing, and it can be used on a cluster for large computations. When using Ray, several processes are involved.

1. Multiple **worker** processes execute tasks and store results in object stores. Each worker is a separate process.
2. **Task** is a *stateless* function that can be executed on a remote worker. **Actor** is a *stateful* object that lives in a remote process.
3. One **object store** per node stores immutable objects in shared memory and allows workers to efficiently share objects on the same node with minimal copying and deserialization.
4. One **raylet** per node assigns tasks to workers on the same node.
5. A **driver** is the Python process that the user controls. For example, if the user is running a script or using a Python shell, then the driver is the Python process that runs the script or the shell. A driver is similar to a worker in that it can submit tasks to its raylet and get objects from the object store, but it is different in that the raylet will not assign tasks to the driver to be executed.
6. A **Redis server** maintains much of the system's state. For example, it keeps track of which objects live on which machines and of the task specifications (but not data). It can also be queried directly for debugging purposes.

2.2 Asynchronous Computation in Ray

Ray enables arbitrary Python functions to be executed asynchronously. This is done by designating a Python function as a **remote function**.

For example, a normal Python function looks like this.

```python
def add1(a, b):
    return a + b
```

A remote function looks like this.

```python
@ray.remote
def add2(a, b):
    return a + b
```

2.2.1 Remote functions

Whereas calling `add1(1, 2)` returns `3` and causes the Python interpreter to block until the computation has finished, calling `add2.remote(1, 2)` immediately returns an object ID and creates a **task**. The task will be scheduled by the system and executed asynchronously (potentially on a different machine). When the task finishes executing, its return value will be stored in the object store.

```python
x_id = add2.remote(1, 2)
ray.get(x_id) # 3
```

The following simple example demonstrates how asynchronous tasks can be used to parallelize computation.

```python
import time

def f1():
    time.sleep(1)

@ray.remote
def f2():
    time.sleep(1)

# The following takes ten seconds.
[f1() for _ in range(10)]

# The following takes one second (assuming the system has at least
    ten CPUs).
ray.get([f2.remote() for _ in range(10)])
```

2.3 Parameter Server

Say we are training a *large* logistic regression model that have `500,000,000` rows of data. We use gradient descent to find the optimal parameters. Each worker can compute gradient for the model for only `10,000,000` rows of data, it takes `10ms` to compute the gradient. We want to train for 10 iteration to achieve good result.

(a) What's the ideal duration to finish compute gradients for one iteration?

☐ 1s [✓ **5s**] ☐ 10s

Solution: **NOTE:** This question was poorly worded and the question statement should say "ten iterations" instead of "one iteration." The solution is 5 seconds when we run our program serially, but when we run our program in parallel, our ideal duration is 0.1 seconds.

(b) Why can't we achieve the ideal duration?

Solution: There will be synchronization and communication costs.

(c) We use the parameter server pattern, given the code:

```python
@ray.remote
class ParameterServer(object):
    def __init__(self, learning_rate):
        self.net = model.SimpleLR(learning_rate=learning_rate)

    def apply_gradients(self, *gradients):
        self.net.apply_gradients(np.mean(gradients, axis=0))
        return self.net.variables.get()

    def get_weights(self):
        return self.net.variables.get()

@ray.remote
class Worker(object):
    def __init__(self, worker_index, batch_size=50):
        self.worker_index = worker_index
        self.batch_size = batch_size
        self.data = input_data.read_data_sets()
        self.net = model.SimpleLR()

    def compute_gradients(self, weights):
        self.net.variables.set(weights)
        xs, ys = self.data.train.next_batch(self.batch_size)
        return self.net.compute_gradients(xs, ys)
```

Fill in the code below, how would we complete the training loop?

```
current_weights = randomly_initialized_weight
while True:
   gradients = [worker.____.remote(current_weights)
             for worker in workers]
   current_weights = ps.____.remote(*gradients)
```

```
current_weights = randomly_initialized_weight
while True:
   gradients = [worker.compute_gradients.remote(current_weights)
             for worker in workers]
   current_weights = ps.apply_gradients.remote(*gradients)
```