CSE 490H: Scalable Systems: Design, Implementation and Use Of Large Scale Clusters

Autumn Quarter 2008

Term Test

Your name: SOLUTION SET

Question 1	_____ /	**6**
Question 2	_____ /	**6**
Question 3	_____ /	**9**
Question 4	_____ /	**6**
Question 5	_____ /	**6**
Question 6	_____ /	**4**
Question 7	_____ /	**8**
Question 8	_____ /	**16**
Question 9	_____ /	**9**
Question 10	_____ /	**6**
Question 11	_____ /	**10**
Question 12	_____ /	**14**
Total	_____ /	**100**

Question 1 (6 points)

Google has built a fault-tolerant, highly-available, recoverable, scalable search application using software techniques.

A (3 points): Identify a situation in which the Google search application uses data partitioning to achieve scalability.

Indexes and documents each are partitioned into “shards” – subsets that contain a certain “range.”

B (3 points): Identify a situation in which the Google search application uses replication to achieve reliability.

Each shard is replicated. If one replica becomes unavailable, another can be used.

Question 2 (6 points)

Describe two ways in which BigTable has less functionality than a traditional relational database system.

It doesn’t support a SQL interface. Data is semi-structured – no type guarantees. Transactions only apply to a single row.

Question 3 (9 points)

A (3 points): What does the IP network protocol accomplish – what does it do? (One sentence)

IP does best-effort routing of single packets from source to destination across multiple heterogeneous networks.

B (3 points): What does the TCP network protocol accomplish – what does it do? (One sentence)

TCP provides reliable delivery of multi-packet messages.

C (3 points): Are TCP packets encapsulated within IP packets, or are IP packets encapsulated within TCP packets?

TCP is encapsulated within IP: {Physical{IP{TCP{payload}}}}

Question 4 (6 points)

A Google File System cluster has a single Master, which holds metadata, and a large number of Chunkservers, which hold file data. GFS uses the Chubby coarse-grained lock service to elect a new Master in the event of a failure.

A (3 points): How is Chubby used for this purpose? That is, how do GFS computers determine a new Master using Chubby? (Just a sentence or two.)

Chubby provides atomic file creation and exclusive locking. The various GFS computers vie to create a new file with a designated file name, writing their name in the file. One succeeds, the others fail.

B (3 points): Like GFS, Chubby uses replication. In the event of the failure of the Master in a Chubby cell, how is a new Master determined? (Just a sentence or two.)

Chubby uses the Paxos algorithm.

Question 5 (6 points)

You don't "need" MapReduce (and the components on which it is built, such as GFS and Chubby) to build applications on top of a huge cluster of commodity computers. MapReduce eases the task, though, by taking care of a large number of headaches that you would otherwise have to write code to deal with yourself. Identify two different major headaches that MapReduce takes care of.

Failure handling. Load balancing. Distribution of data among workers. Incremental scalability.

Question 6 (4 points)

The major components of a computer are CPU, RAM, network, and disk. Which component(s) are typically the rate-limiting ones in a MapReduce process? Explain why. (Just a few sentences.)

Network and disk. MapReduce algorithms are inevitably I/O-bound.

Question 7 (8 points)

Given the following web link graph:

```
A -> [B, C]
B -> [A, C]
C -> [D, B, A]
D -> [A]
```

where `X -> [Y, Z]` means page `X` links to pages `Y` and `Z`, show <u>one iteration</u> of the PageRank algorithm. Assume d = .85. Before iteration 1, initialize each pagerank to 0.15.

```
Map
     A:
          A -> [B, C]
          B -> 1/2(0.15)          C -> 1/2(0.15)
     B:
          B -> [A, C]
          A -> 1/2(0.15)          C -> 1/2(0.15)
     C:
          C -> [D, B, A]
          D ->  1/3(0.15)         B -> 1/3(0.15)          A -> 1/3(0.15)
     D:
          D -> [A]
          A -> 0.15

Reduce
                       B        C        D
     A:  0.15 + 0.85(0.15/2 + 0.15/3 + 0.15)  =  0.38375

                       A       C
     B:  0.15 + 0.85(0.15/2 + 0.15/3)  =  0.25625

                      A        B
     C:  0.15 + 0.85(0.15/2 + 0.15/2)  =  0.2775

                      C
     D:  0.15 + 0.85(0.05)  =  0.1925
```

Question 8 (16 points)

Given these input data structures:

```
class Foo implements Writable {
   int fooIdentificationKey;
   int someFooData;
   float importantFooMagic;

   void write(DataOutput out) { } // elided
   void readFields(DataInput in) { } // elided
}

class Bar implements Writable {
   int barIdentificationKey;
   String barString;
   int relatedFooItem;

   void write(DataOutput out) { } // elided
   void readFields(DataInput in) { } // elided
}
```

A (4 points): Create a datatype that has the following properties:

- It can represent the contents of either a `Foo` or a `Bar` object.
- A `Bar` object should be able to be joined with the `importantFooMagic` field of the corresponding `Foo` object it references.
- We must be able to distinguish between `Bar` objects that have been through this join process and those that have not.

Show all the fields the object requires; also show the `write()` method body. (You do *not* need to show the `readFields()`, `compareTo()`, `equals()`, `toString()`, or `hashCode()` methods.) For reference, assume the following interface:

```
interface DataOutput {
   public void writeInt(int x);
   public void writeLong(long x);
   public void writeFloat(float x);
   public void writeDouble(double x);
   public void writeString(String x);
   public void writeBoolean(bool x);
   public void writeChar(char x);
}
```

```
class FooOrBar implements Writable {
  enum { FOO, BAR, JOINED_BAR };
  int TAG;

  // ident key + TAG is common to both; ok to have
  // separate fooIdentification, barIdentification
  int identificationKey;

  int someFooData;
  float importantFooMagic; // reused by JOINED_BAR

  String barString;
  int relatedFooItem;

  public void writeFields(DataOutput out) {
    out.writeInt(TAG);
    out.writeInt(identificationKey);
    if (TAG == FOO) {
      out.writeInt(someFooData);
      out.writeFloat(importantFooMagic);
    } else if (TAG == BAR) {
      out.writeString(barString);
      out.writeInt(relatedFooItem);
    } else if (TAG == JOINED_BAR) {
      out.writeString(barString);
      out.writeInt(relatedFooItem);
      out.writeFloat(importantFooMagic);
    }
  }
}
```

(It’s OK to nest Foo and Bar and defer to their writeFields() instead of inlining. You must still include the type discrimination tag.)

B (6 points): Write the mapper and reducer code which reads in objects of your combined data type, and emits them back out; `Foo` objects should be unchanged, but `Bar` objects should have had the magic data from their related `Foo` objects joined in.

Assume that `Foo`-style values (magically) always arrive "first in line" at a reducer ahead of any `Bar`-style values.

Assume that the key arriving at the mapper is irrelevant.

```
void map(SomeKeyType key, FooOrBar val, OutputCollector out,
  Reporter r) {
  if (val.TAG == FOO) {
    out.collect(val.identificationKey, val);
  } else if (val.TAG == BAR || val.TAG == JOINED_BAR) {
    out.collect(val.relatedFooItem, val);
  }
}

void reduce(int fooId, Iterator<FooOrBar> vals,
  OutputCollector out, Reporter r) {
  float magicFooVal;

  foreach (FooOrBar val : vals) {
    if (val.TAG == FOO) {
      magicFooVal = val.importantFooMagic;
      gotFoo = true;
      out.collect(fooId, val);
    } else if (val.TAG == BAR) {
      val.TAG = JOINED_BAR;
      // remember, foo always came first; this is ok
      val.importantFooMagic = magicFooVal;
      out.collect(fooId, val);
    } else if (val.TAG == JOINED_BAR) {
      out.collect(fooId, val); // unchanged
    }
  }
}
```

C (3 points): Why is it important for the `Foo`-style values to arrive at the reducers before the `Bar`-style values?

Otherwise, we would have to buffer the Bars before the joining Foo came. If there were a lot of Bars, then this might overflow memory on a single node, so there would be a limit on its scalability.

D (3 points): What is the general relationship (the "contract") between the implementations of the `compareTo`, `equals`, and `hashCode` methods? Why is this important for MapReduce? (Just a few sentences.)

```
x.equals(y) ⇔ x.comparesTo(y) == 0  =>
               x.hashCode() == y.hashCode()
```

```
MapReduce uses hashCode() to select the reduce shard, and
compareTo() to sort keys.  If compareTo and hashCode don't work
sanely, keys may not all arrive at the same reducer in the
correct order and the reducer won't get the correct set of
values together.
```

Question 9 (9 points)

A (6 points): Why is data not lost when a single machine fails in an HDFS cluster? Describe the steps the system takes to ensure this.

```
HDFS keeps three replicas at all times, so a single failure does
not cause data loss.  Machines are heart-beat, so the system
knows when a machine is down.  When this happens, a surviving
replica is copied to another machine.
```

B (3 points): Under what conditions could HDFS lose data permanently?

```
If the wrong three machines failed simultaneously, you could
lose all replicas of a file.  Also, the NameNode is a single
point of failure – you could lose the metadata.
```

Question 10 (6 points)

A (3 points): Assuming a Paxos cluster of 7 nodes, at most how many nodes can fail and leave the system remaining consistent (functioning correctly)?

```
No more than 3.
```

B (3 points): Why can Paxos not support more failures than this?

```
You need to receive confirmation from more than half of the
original number of nodes.
```

Question 11 (10 points)

Virtual machine monitors have recently found a “new life” for server consolidation (multiple services on a single server).

A (2 points): Identify one key characteristic of VMM’s that makes them particularly suitable for this task.

Guest OS’s are isolated from one another. Deployment of applications can be simple and automated.

B (8 points): Trace the steps that occur when an application running on a guest operating system in a virtual machine attempts to do a file operation – identify each transition among application, guest OS, VMM, and hardware, and identify the mechanism that causes each transition.

- The application executes a privileged instruction in the syscall stub.
- The hardware reflects this up to the VMM, which reflects it up to the appropriate guest OS, causing its syscall code to be invoked.
- The guest OS executes a privileged I/O operation on behalf of the application.
- The hardware reflects this up to the VMM, which “simulates” the operation on the appropriate virtual disk of the appropriate guest OS. Probably this requires the VMM to execute a privileged I/O instruction, which actually causes the physical disk to do something because the VMM is executing in kernel mode so the privileged instruction executes rather than trapping.
- When the physical I/O completes, the completion interrupt is handled in the VMM, which reflects the completion up to the guest OS (as a virtual completion interrupt), which can then return from the original syscall.

Question 12 (14 points)

Implement `Variance(X)` using MapReduce.

The `Variance` of `n` values of the variable `X` is defined as

$$\text{Variance(X)} = \sum_{i=1}^{n} (x_i - \mu)^2$$

where μ is the arithmetic mean of the values.

The input to your program is a file including several intermixed datasets. A dataset is the multiple values for a single variable. Each line in the file consists of a key (the name of the variable) and a single value. The same values may repeat within a dataset. Thus, the input file looks like:

```
K1   Value1_for_K1
K1   Value2_for_K1
K2   Value1_for_K2
K1   Value3_for_K1
K2   Value2_for_K2
Etc.
```

The output of your program should have a `(Key, Variance)` pair for each key (each variable) in the input dataset.

What are the scalability limits, if any, of your solution?

There were three basic approaches to this problem, at different points on the scalability and efficiency spectrum, worth 5, 9 and 14/14 points (as well as a possible super-bonus point) each.

5/14: Buffering Solution

```
map(k, v) => emit(k, v)    // identity mapper

reducer(k, iter<v> vals) {
  let arr = new ArrayList();
  sum = 0;
  count = 0;
  for (val : vals) {
    sum = sum + val;
    count++;
    arr.add(val.copy()); // buffer iterator input
  }

  mu = sum / count;
  variance = 0;
  for (val : arr) {
    variance += (val - mu)^2;
  }

  emit(k, variance);
}
```

Scalability limits: This program buffers all the values associated with a key in RAM; thus a lot of values for the same key will crash the reducer.

+1 if you realized that you could do the next solution, but didn't actually write the [pseudo]code for it.

-1 if you tried to use the iterator twice. You can't reset the iterator in Hadoop. You must use a separate array.

<u>9/14: Two pass solution</u>

```
map1(k, v) => emit(k, v); //id mapper
reduce1(k, iter<v> vals) {
  sum = 0;
  count = 0;
  for (val : vals) {
    sum += val;
    count++;
    emit(k, VAL(val));
  }
  emit(k, MU(sum/count));
}

// assume we can perform secondary sorting so MU-tagged elements
// arrive at reducer before VAL-tagged elements
map2(k, v) => emit(k, v); //id mapper over output of reduce1
reduce2(k, iter<v> vals) {
  variance = 0;
  for (val : vals) {
    if isMu(val) {
      mu = val;
    } else {
      // mu element is always first, so mu will be defined here.
      variance += (val - mu)^2;
    }
  }

  emit(k, variance);
}
```

Scalability limits: None. You just take an extra read-write cycle.

<u>14/14: Streaming variance</u>

```
Observe:
sigma^2 == Sum (x_i - mu)^2 ==
Sum ( x_i^2 - 2x_i mu + mu^2 ) ==

   x_i^2 - 2x_i mu + mu^2
+  x_j^2 - 2x_j mu + mu^2
====================================
factor vertically into three tallys:

*  (x_i^2) + (x_j^2) + ...      // x_squared
*  - (2x_i) - (2x_j) + ...
  == -2 * mu ( x_i + x_j + ...) // x_sum
* mu * (1 + 1 + ...)            // count

map(k, v) => emit(k, v) // id mapper
reduce(k, iter<v> vals) {
  x_sum = 0;
  x_squared = 0;
  count = 0;
  for (val : vals) {
    count++;
    x_sum += val;
    x_squared += val^2;
  }

  variance = x_squared - (2 * x_sum * mu) + (mu * mu);
  emit(k, variance);
}
```

Scalability limits: The algorithm has virtually unlimited scalability, although we're still using a fair amount of bandwidth due to the identity mapper. To handle this, we would need:

15/14: Fully tree-reduced variance

```
Mapper(k, v) => emit (k, (mu=v, cnt=1, var=0))
Combiner(k, iter<v> vals)  emits (k, (mu, count, variance)) for
subset
Reducer(k, iter<v> vals) emits final
  (k, (mu_total, count_total, var_total)
```