1. De las tres características del teorema CAP, Consistency, Partition Tolerance & Availability, Hadoop elige Consistencia y tolerancia a particiones de la red.
La disponibilidad no está asegurada, ya que si fallan tres nodos del HDFS, puede darse el caso de que algunos archivos se tornen no disponibles.

CP: Hadoop y Zookeeper. Ambos eligen consistencia. La disponibilidad es mucho más cara de mantener que la consistencia. En estos sistemas los datos pueden tornarse no disponibles, pero cuando están, todos los ven de manera consistente.
AP: Ejemplos Amazon Dynamo, CouchDB, Cassandra, SimpleDB, Riak, Voldemort.
CA: Parece ser que en la práctica uno no puede elegir 'no soportar' particiones en la red. Mantener C y A sin P, es un lugar del espacio de diseño con poco valor práctico.

La realidad es que en la práctica no todas las capacidades son excluyentes de manera absoluta, sino que existe un contínuo entre las capacidades donde se puede comprometer partes de una característica en beneficio de alguna otra.
Parece ser que la realidad de CAP, al día de la fecha, para los diseñadores de sistemas distribuidos es ver como balancear la C y la A manteniendo P. Digamos, no se pueden salvar de encargarse de P.

2. En este caso, hice el esfuerzo de mantener el mismo dataset del práctico anterior, e incluso el mismo split entre train y test. Aquí están los datos separados entre training y test,

Los datos son de training son:
[1, 1, 1, 0]

[1, 0, 0, 1]

[1, 0, 1, 1]

[0, 0, 0, 0]

[0, 1, 0, 1]

[1, 1, 0, 0]

[1, 0, 0, 1]

[1, 1, 1, 0]

[1, 1, 1, 0]

[0, 0, 0, 0]

[1, 0, 1, 1]

[0, 0, 1, 0]

[0, 0, 0, 0]

[0, 0, 0, 0]

[0, 0, 0, 0]

[0, 1, 0, 1]

Los datos son de test son:
[1, 1, 0, 0]

[0, 0, 0, 0]

[0, 1, 0, 1]

[0, 1, 1, 1]

[0, 0, 0, 0]

[0, 1, 1, 1]

[1, 1, 0, 0]

[1, 0, 0, 1]

[0, 0, 0, 0]

[0, 1, 1, 1]

[0, 1, 1, 1]

[1, 0, 1, 1]

[1, 1, 1, 0]

[0, 1, 0, 1]

Este es el árbol que aprende ID3, con los resultados de precision & recall correspondientes,

```
 Attribute:    Value:
   Attribute: 0   Value: 0
      Attribute: 1   Value: 0
      Attribute: 1   Value: 1
   Attribute: 0   Value: 1
      Attribute: 1   Value: 0
      Attribute: 1   Value: 1
```

precision = 0.571428571429
recall = 1.0

En el caso de Random Forest, utilicé scikit-learn. Y como era de esperar, los resultados varían según la corrida, estos son algunos resultados,

precision = 0.666666666667
recall = 0.5

precision = 1.0
recall = 1.0

precision = 1.0
recall = 0.5

precision = 0.8
recall = 1.0

precision = 1.0
recall = 0.375

Los resultados anteriores para ID3 son siempre los mismos, por supuesto.

4. Descomposición LU.
La implementación MapReduce de este algoritmo tiene tres partes:

1. Se crean algunos archivos de control en el HDFS utilizando el master node. Estos archivos son utilizados como entradas a los mappers de todos los trabajos MapReduce.
2. La partición de los datos de acuerdo al procedimiento recursivo que se explica en la tesis.
3. Una serie de trabajos MapReduce que calculan los diferentes pedazos de la matriz resultante.

Algunos parámetros del algoritmo:
N^2 = 10.000x10.000
m0 = número inicial de archivos / número de nodos de cómputo
nb = 3200. El límite de N para un solo nodo.

Si bien el procedimiento es recursivo, la profundidad de la recursión y el número de MapReduce jobs puede calcularse de antemano, utilizando la fórmula de la tesis,

nb = 3200 => 4 jobs.

Dibujo1.

(1) Se arranca el algoritmo creando m0 archivos y se los almacena en Root/MapInput/. Cada archivo contiene solo un entero. A.0 contiene 0, A.1 contiene 1, hasta A.m0-1 que contiene m0-1. Los mappers utilizan estos archivos para controlar el cómputo y van produciendo las distintas partes de la salida para hacer la inversión.

(2) La partición se hace con un trabajo MapReduce que solo contiene mappers que particionan los datos. Los mappers de este trabajo leen los datos desde Root/MapInput. El entero que leen de cada archivo le sirve a cada mapper para saber que pedazo de la matriz tienen que
La partición de la matriz inicial se hace en el directorio Root/ y después se repite el procedimiento volviendo a crear los archivos de control adentro de otros subdirectorios debajo de Root/. El resultado es como se muestra abajo,

Dibujo 2.

Por ejemplo, A3 se particiona por filas, y es así como se va a leer durante el cómputo en paralelo. Por ejemplo, Root/A1/A1/A.0, se obtiene partiendo la matriz “top level” en cuatro, tomando el primer cuarto(A1) y volvéndolo a partir en cuatro.

(3) Una vez que A está particionada, y todos los archivos están en HDFS, y se empieza el cómputo leyendo A1, A2, A3, y A4 desde disco. Se lanza un trabajo MapReduce para computar U2, L2' y B.
La función map computa U2 y L2' y el reduce computa B = A4 -L2'U2.
Cada mapper emite un (key, value) = (j,j), donde j es el integer que leyó de su archivo de entrada. Esos j se utilizan para controlar la parte de B que cada reducer debe computar. En el caso de la tesis, el worker j computa el bloque j de B, por eso esos (key, value).
Este dibujo(*) no está en la tesis, pero me parece útil para ver como se mueven los datos,

(*) Cada flecha en este dibujo puede representar más de un mapper/reducer. No se explican detalles de las particiones de los files en el HDFS.