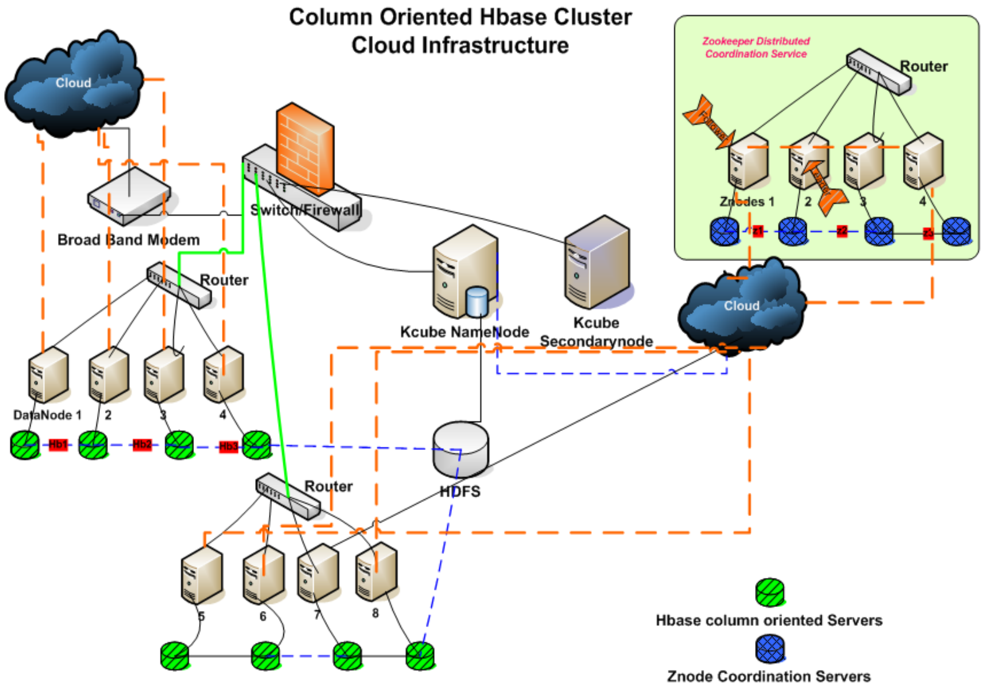
MapReduce and Big Data

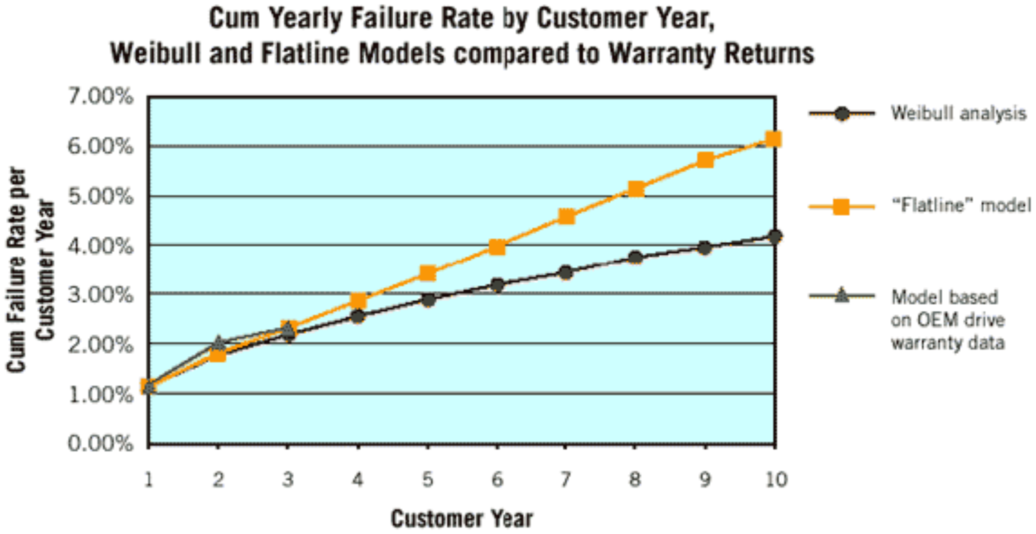
Cloud computing

- Leverage large numbers of consumer grade hardware.
- Failure modes
  - Network failure
  - CPU Failure
  - Hard drive failure



How big is big?

- Data is truly big if the probability of a node failing while the data is in process is nonnegligible.



MapReduce

- Map/Reduce is not an algorithm - it is a programming pattern

- The advantage is not is efficiency
  - Parallelizability
  - Dependency Analysis
  - Robustness

MapReduce

"Map" step: Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.

"Shuffle" step: Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.

"Reduce" step: Worker nodes now process each group of output data, per key, in parallel.

MapReduce

- Map
  - Take input data
  - Output (key, value) pairs
- Shuffle
  - Group data by (key), feed it back to processing node
- Sort
  - Order all data per key by some function
- Reduce
  - Process all values of key

MapReduce

1. **Prepare the Map() input** – the "MapReduce system" designates Map processors, assigns the input key value *K1* that each processor would work on, and provides that processor with all the input data associated with that key value.
2. **Run the user-provided Map() code** – Map() is run exactly once for each *K1* key value, generating output organized by key values *K2*.
3. **"Shuffle" the Map output to the Reduce processors** – the MapReduce system designates Reduce processors, assigns the *K2* key value each processor should work on, and provides that processor with all the Map-generated data associated with that key value.
4. **Run the user-provided Reduce() code** – Reduce() is run exactly once for each *K2* key value produced by the Map step.
5. **Produce the final output** – the MapReduce system collects all the Reduce output, and sorts it by *K2* to produce the final outcome.

MapReduce

The prototypical MapReduce example counts the appearance of each word in a set of documents:[11]

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += ParseInt(pc)
  emit (word, sum)
```

MapReduce in Java and KMeans

http://www.slideshare.net/andreaiacono/mapreduce-34478449

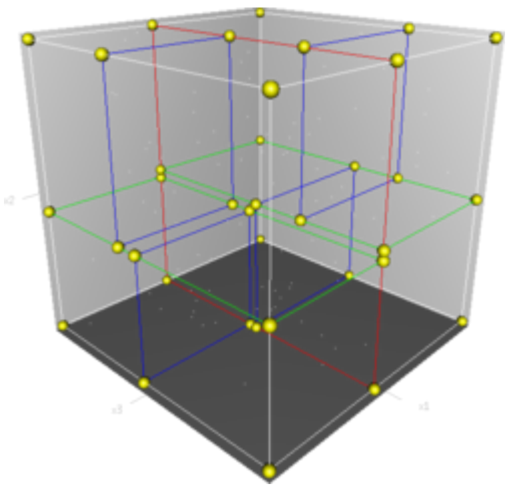
KD Tree

- KD Trees are multidimensional binary trees

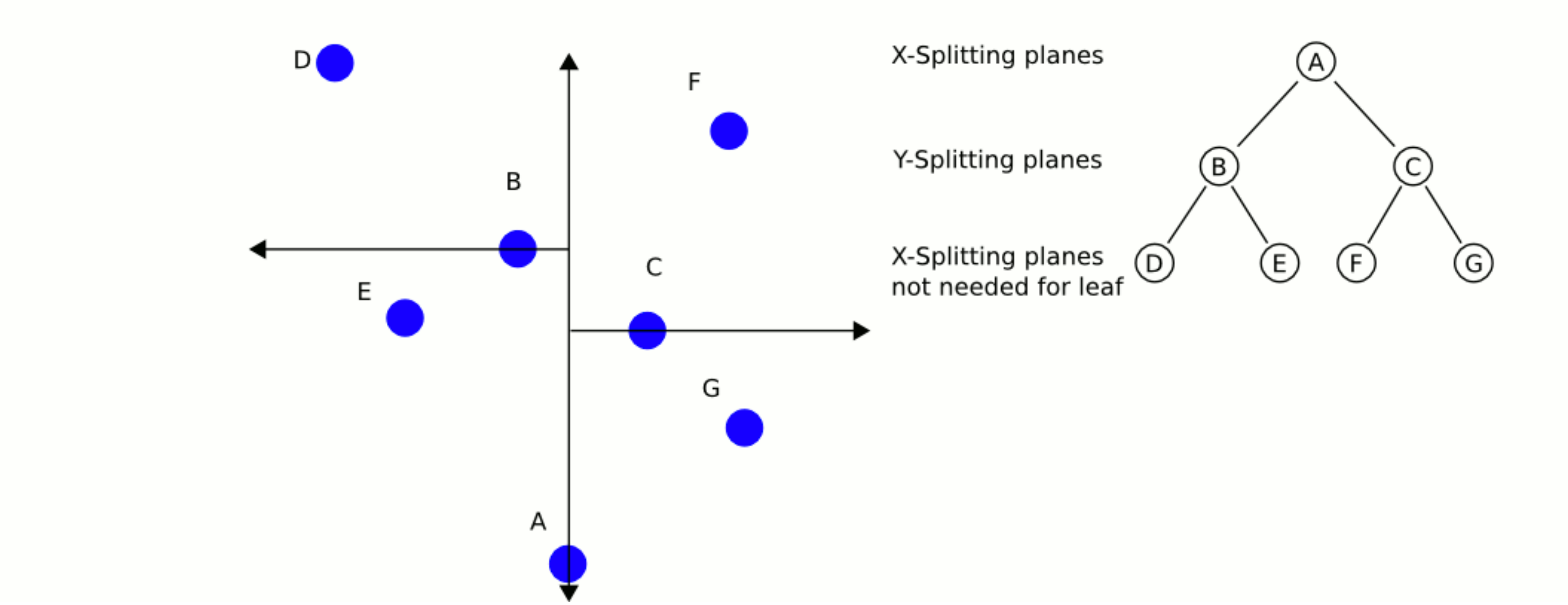
```
function kdtree (list of points pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtrees
    var tree_node node;
    node.location := median;
    node.leftChild := kdtree(points in pointList before median, depth+1);
    node.rightChild := kdtree(points in pointList after median, depth+1);
    return node;
}
```



KD Tree


X-Splitting planes
Y-Splitting planes
X-Splitting planes
not needed for leaf
A
B
C
D
E
F
G

KD Tree

Building a static *k*-d tree from *n* points has the following worst-case complexity:

$O(n \log^2 n)$ if an $O(n \log n)$ sort such as Heapsort or Mergesort is used to find the median at each level of the nascent tree;

$O(n \log n)$ if an $O(n)$ median of medians algorithm[3][4] is used to select the median at each level of the nascent tree;

$O(kn \log n)$ if *n* points are presorted in each of *k* dimensions using an $O(n \log n)$ sort such as Heapsort or Mergesort prior to building the *k*-d tree.[7]

- Inserting a new point into a balanced *k*-d tree takes $O(\log n)$ time.
- Removing a point from a balanced *k*-d tree takes $O(\log n))$ time.
- Querying an axis-parallel range in a balanced *k*-d tree takes $O(n^{1-1/k} +m)$ time, where *m* is the number of the reported points, and *k* the dimension of the *k*-d tree.
- Finding 1 nearest neighbour in a balanced *k*-d tree with randomly distributed points takes $O(\log n)$ time on average.

KD Tree

- KD Trees are multidimensional binary trees
- Distributed Kd-Trees for Retrieval from Very

Large Image Collections

  - http://vision.caltech.edu/malaa/publications/aly11distributed.pdf