Midterm Examination

CSE 455 / CIS 555 – Internet and Web Systems

Spring 2009

Zachary Ives

Name: _Solution____

6 questions, 100 pts, 80 minutes

1. *(20 pts)* Compare Hadoop (plus HDFS) to the Chord DHT. (a) What are the differences in how they *partition* data?

HDFS partitions by pages, according to the structure of the file. (It can also partition using other schemes, but this is the default.)

Chord partitions key-value pairs by hash key, allocating to the node whose value is the successor (modulo 2^{160}) of the key's hash value.

(b) How is *communication/coordination* achieved, in order to perform a computation?

Hadoop expresses computation using map and reduce operations. The Hadoop runtime allocates map operations to individual nodes, and their output is to temporary files. The results of map operations are sorted and merged by keys, then reduce operations are performed.

Chord uses messages (common P2P API) or get/put calls as its means of communication. In essence messages are routed to form communication.

(c) How is *failure recovery* done?

Hadoop will monitor nodes for failures, and restart jobs that did not complete. It uses the HDFS fielsystem as a means of ensuring the appropriate outputs are available to be merged. Data is replicated in HDFS to ensure availability.

Chord only handles failures in the sense that it replicates data – an outstanding request could fail and Chord will not restart. But replica sets are maintained at the successors of a node.

2. *(20 pts)* Suppose we were to reimplement the DNS using a hash-based scheme, rather than a directory-based scheme. How could this be achieved in a scalable way, given the technologies and algorithms we have seen in the course? What would the differences be, both functionally and administratively?

   One could have a set of DNS nodes in a Chord/Pastry ring, or any other distributed partitioned environment. Now names do not need to be hierarchical – hashing would be used to distribute them across nodes. Chord/Pastry would even allow for replication and for the set of DNS nodes to change over time.

   It is possible to still use caching in such an environment – the local DNS resolver could cache.

   What substantially changes is that administrative control on the system is less obvious: it isn't clear who gets permission to "publish" DNS entries, and what relationship these entries need to have to one another. There's no longer a need to use hierarchical naming, where each domain or subdomain administers its member subdomains. But in fact one could still impose this structural naming convention even with a hash-based scheme.

   One other aspect that changes is that the allocation of names to nodes is done in a way that doesn't consider administrative structure. In the current DNS, a domain has a structural relationship with all of its subdomains, and in principle it holds the subdomains' data. This becomes less true in a hash-based scheme – one might hold data from other sites.

3. (15 pts) Given two XML files with the following form:

<u>Students.xml</u>

```
<students>
  <student>
    <sid>{student ID}</sid>
    <name>{person}</name>
  </student>
…
</students>
```

<u>Classes.xml</u>

```
<classes>
  <takes>
    <sid> { student ID } </sid>
    <course> { Course name } </course>
  </takes>
  …
</classes>
```

Write an XQuery that outputs, for each student (in tag <student>), the student’s name (with the tag <name>) and courses taken (with the tag <takes>).

```
for $s in doc(“Students.xml”)/students/student,
      $n in $s/name,
      $i in $s/sid/text()
return <student>
            { $n }
            {
              for $c in doc(“Classes.xml”)/classes/takes,
                    $i2 in $c/sid/text(),
                    $crs in $c/course/text()
               where $i = $i2
               return <takes> { $crs } </takes>
            }
           </student>
```

4. (15 pts) Explain the intuitions for the three cornerstones of a Google-like keyword search and rank system: (a) the vector space model, (b) TF/IDF, (c) PageRank.

(a) The vector space model creates a *document vector* for each document, and a *query vector* for a keyword query. Each element in the vector represents a score related to a keyword – and keywords are assumed to be independent (and modeled as orthogonal dimensions in the vector space). The vector space model takes the cosine of the angle between the query vector and a given document vector. The rationale is that query + document combinations that are well-aligned will have parallel vectors (angle 0, cosine 1) and combinations that have no similarity will have perpendicular vectors (angle 90 degrees, cosine 0).

(b) The TF/IDF model looks at the significance of keyword occurrences within a document. Intuitively, the more times a term appears in a document, the more representative the keyword should be of the document (this is TF). Yet the more common the keyword across all documents, the less significant the keyword (this is the IDF).

(c) PageRank is based on the "random surfer model", assessing how likely a surfer who randomly follows out-links is to come to a given document. Intuitively, the better-linked the document, the more likely – but the popularity of the sites where the links originate also matters. Hence, documents linked to by high-PageRank documents have high PageRank. There is a small probability of randomly jumping to another document in the system – necessary to ensure PageRank's stability with cycles, dead-ends, and other idiosyncrasies in the graph.

5. (10 pts) The B+ Tree is a balanced tree index structure. Provide pseudocode showing how insertions are handled in the B+ Tree with order *d*.

```
Insert(v)
  1. Let n ≔ find (v)
  2. If n is full then:
       a. Create a new node n2
       b. Let L ≔ sort all ((v’ in n) union v)
       c. Put the first d / 2 entries of L in n, and the remainder in n2
       d. CopyUp(middle value in the sorted list)
  3. Else
       a. Put v into its place in n
  4. End if

CopyUp(v, n)
  1. If n is full then:
       a. Create a new node n2
       b. Let L ≔ sort all ((v’ in n) union v)
       c. Put the first d / 2 entries of L in n, and the last d / 2 in n2
       d. PushUp(middle value in the sorted list)
  2. Else
       a. Put v into its place in n
  3. End if

PushUp(v, n)
  1. If n is full then:
       a. Create a new parent node n2 and child node n3
       b. Let L ≔ sort all ((v’ in n) union v)
       c. Put the first d / 2 entries of L in n, and the last d / 2 in n3
       d. Give n2 the middle value in the sorted list
       e. Add n and n3 as children of n2
  2. Else
       a. Put v into its place in n
  3. End if
```

6. *(20 pts)* Assume we are given a task to determine the most popular Web site (domain name) given a log of access requests, each of the form "URL requestor". Assume you are given a function **String getDomain(URL)** that retrieves the domain name from each URL. Show pseudocode for the map and reduce functions for a MapReduce operation to do this computation.

```
Map(URL, requestor)
    Let d ≔ getDomain(URL)
    Emit (d, 1)
```

```
Reduce(domain, {sequence of counts})
    Let L ≔ length of the sequence
    Emit(d, l)
```

Now we need to scan through the Reduce outputs and find the max domain/count pair.