KEVIN JOHNSON

MODULE 6 READING COMPREHENSION

1. **Describe some ways to ensure partitions. (2 Pts)**

   The goal of partitioning is to spread the data and the query load evenly across nodes. One way of partitioning is to assign a continuous range of keys to each partition. If you know the boundaries between the ranges, you can easily determine which partition contains a given key. If you also know which partition is assigned to which node, then you can make your request directly to the appropriate node. Another way to ensure portioning is a hash function. A good hash function takes skewed data and makes it uniformly distributed. Even if the input strings are very similar, their hashes are evenly distributed across that range of numbers.

2. **Describe the two main approaches to partitioning a database with secondary indexes. (2 Pts)**

   The two main approaches to partitioning a database with secondary indexes: document-based partitioning and term-based partitioning. In document-based indexing, each partition is completely separate. Each partition maintains its own secondary indexes, covering only the documents in that partition. Whenever you need to write to the database, you only need to deal with the partition that contains the document ID that you are writing. A document-partitioned index is also known as a local index.

   Rather than each partition having its own secondary index, a global index can be constructed that covers data in all partitions. A global index is partitioned in term-based partitioning. In this instance, the term we are looking for determines the partition of the index. The example uses red cars, which appear under color: red in the index. However, the index is partitioned so that colors starting with the letters a to r appear in partition 0 and colors starting s to z appear in partition 1.

3. **Describe some ways to rebalance partitions. (2 Pts)**

   Rebalancing is the process of moving load from one node in the cluster to another. There are several strategies for rebalancing:

   - Fixed number of partitions: Create many more partitions than there are nodes and assign several partitions to each node. If a new node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again. Only entire partitions are moved between nodes. The number of partitions does not change, nor does the assignment of keys to partitions. The only thing that changes is the assignment of partitions to nodes.
   - Dynamic partition: Key-range partitioned databases create partitions dynamically. When a partition grows to exceed a configured size, it is split into two partitions so that approximately half of the data ends up on each side of the

split. Conversely, if lots of data is deleted and a partition shrinks below some threshold, it can be merged with an adjacent partition.

- Partitioning proportionally to nodes: There are a fixed number of partitions to nodes. The size of each partition grows proportionally to the dataset size while the number of nodes remains unchanged. When you increase the number of nodes, the partitions become smaller again. Since a larger data volume generally requires a larger number of nodes to store, this approach also keeps the size of each partition fairly stable.

4. **Describe the concept of service discovery and some ways of dealing with it. What are some existing products that deal with this concept in terms of distributed data systems. (2 Pts)**

As partitions are rebalanced, the assignment of partitions to nodes changes. Somebody needs to stay on top of those changes in order to know which IP address and port number to connect when reading or writing a key. The main question of service discovery is "How does the component making the routing decision learn about changes in the assignment of partitions to nodes?" There are three approaches described for this problem:

1. Allow clients to contact any node. If that node owns the partition to which the request applies, it can handle the request directly. Otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply on to the client.
2. Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does itself handle any requests but only acts as a partition-aware load balancer.
3. Require that clients be aware of the partitioning and the assignment of partitions to nodes. The client can connect directly to the appropriate node without any intermediary.

Distributed data systems rely on a separate coordination service to keep track of the cluster metadata. One such service is Zookeeper. LinkedIn, HBase, SolrCloud, and Kafka all use ZooKeeper to track partition assignment

5. **What are the different parts of a graph and how do they affect the complexity (in terms of O notation) of the Breadth First Search Algorithm? (1 Pt)**

A graph is made up of nodes and edges. A node is each item in the graph, while an edge is the connection between each node. A node can be directly connected to many other nodes, which are called its neighbors. If you search the entire network, that means you will follow each edge. So the running time will be at least O(number of edges). You also keep a queue of every item to search. Adding one item to the queue takes constant time O(1). Doing this will take O(number of nodes) total. Therefore, Breadth-first search takes O(number of edges + number of nodes), which is more commonly written as O(V+E).

6. **What is the difference between a Breadth First Search and a Depth First Search? When should someone use one over the other? (1 Pt)**

   The difference between breadth-first and depth-first search lies in the order each one searches through a list. Breadth-first is a vertex based technique for finding the shortest path in the graph. Breadth first looks at all nodes at one level before moving onto the next level. Depth-first searches all the way down one path before going back and searching the next path. Breadth-first is more suitable for searching vertices that are closer to the given source, while depth-first is more suitable where there are solutions away from the source.