1. **Describe some ways to ensure balanced partitions.**

There are a few different ways to ensure balanced partitions, including partitioning by key range or key hash. Partitioning by key range will have limited success since there will be likely be unequal utilization of fixed key ranges, though this can be ameliorated by rebalancing on the key range once hot spots are detected or by using a key hash, allowing for a systematic randomization to create an index that will, in the long run, balance the data across the given partitions.

1. **Describe the two main approaches to partitioning a database with secondary indexes.**

***[This answer may be incomplete, please see Q&A response for more detail on issues I have with Term Partitioned Indexing]***

Document Partitioned Indexes are written locally to each partition, and only concerns the data kept on that partition. It allows for faster writes because the write only has to go to that partition but enforces a scatter/gather reading method (i.e. all partitions must be read when a user wishes to utilize a given secondary index).

Term Partitioned Indexes are stored on each partition but contains information about records stored on all partitions (including which partitions the records on). The phrase *Term Partitioned* comes from the idea that even the indexes are split based on the term, such that different parts of the same secondary index live on different nodes. Kleppmann illustrates this in Figure 6-5 in *Designing Data Applications,* where color and make both have secondary index entries on two illustrated partitions, separated by the value of the index (figure below). This means that when a user queries based on a given value of a secondary index, the query runs through the partition containing the index term, and then performs a “directed” scatter/gather query on partitions that are known to have the records (based on primary indexing).

A screenshot of a cell phone

Description automatically generatedFor example, in the figure below, if a user queries for a red car, the query is routed to Partition 0 since that’s where the secondary, term-based index keeps the term *red* for the *color* index. This then sends a request for information out to Partitions 0 and 1 since that is where the information is stored on red cars (knowing this because the primary index for 191 and 306 are on Partition 0, and the primary index for 768 is on Partition 1). The results are then aggregated and returned to the user.

1. **Describe some ways to rebalance partitions.**

Setting a fixed number of partitions when designing the database may work for balancing data because it allows for simple rebalancing operations, so long as each node contains an equal amount of data across partitions.

Dynamic partitioning allows for partition splits to be consistent with the size of the database, avoiding issues with fixed partitions in the case of key range partitioning. It splits the data into multiple partitions when growing, such that each partition receives half of the data – this allows for a growth in the number of partitions, which can be evenly distributed across nodes.

Partitioning proportionally to nodes allows for the number of partitions per node to remain constant, meaning that lightening the load of each partition (and node) is fixed on the number of nodes available. When a new node is added, there are new partitions available for rebalancing. This shifts the burden of rebalancing to a hardware issue as opposed to an allocation problem.

1. **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?**

Service discovery is the process of a client finding the appropriate connection to be able to perform the action (s)he wants to (whether that be a read or a write). This is prevalent in tools that require a connection to work, like an API or cloud-hosted service.

There are generally three approaches on who holds the knowledge of what connection to make, but the most important part is that all parts of the system agree on who (or what) is responsible for that knowledge. When the user is responsible for the knowledge, (s)he must know which node to connect to and the port associated. If the information is entered incorrectly, there may be an empty return set for a read when the information simply exists elsewhere. There is a distributed knowledge model where each node has the knowledge of where all data exists on every other node, and can reroute requests based on that knowledge. While this is great from a system redundancy point of view, this may lead to some latency in requests since the requests are routed randomly, and may lead to some hotspots in the cases of a biased method of a first direct. Lastly, there are routing tiers (that may or may not use an external application) that contains the knowledge of which node to query, and can be updated upon write from each node. This seems to be a common solution since the nodes can be utilized efficiently and it reduces read/write latency.

1. **(Algorithms) 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?**

Graphs have two main parts: vertices (nodes) and edges (connections). These each add equally to the complexity of a breadth-first search, as the time complexity is *O(V+E)*. Each new vertex requires that it be searched (and a cost associated with it). This adds complexity linearly. For each edge that a vertex has, however, additional complexity is added since the vertex on the other side of that edge must be searched, which also adds complexity linearly. If the number of vertices in a given graph stays the same, but there are more edges between them, the complexity rises in a linear fashion.

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

The goal of a depth-first search is to exhaust the options of a particular path in search for a valid response to end the search. A breadth-first search instead searches all paths in a parallel fashion, settling upon the shortest distance to the satisfactory case and ceasing operations afterwards. Depth-first search is particularly useful in cases where a solution may not be particularly close (to traverse an entire tree or graph may not be feasible based on hardware constraints) or where one choice may logically lead to another choice (such as in simulations like manufacturing systems or chess). Breadth-first search may be a better choice when the shortest path is the desired one, whether that invokes weighting or not. Breadth-first is a good option when there are limited branches (edges) from the origin and a close solution is the optimal one.