DS 4300 - Spring 2025

Sample Midterm Questions (& HW 04)

Below are some sample exam questions with which you can test your RAG “Cheat Sheet”.

1. What is the difference between a list where memory is contiguously allocated and a list where linked structures are used?
    - **Response:**
      Memory allocation in lists can be done in two main ways: contiguous allocation and linked structure allocation.

      In contiguous allocation, memory is allocated contiguously for an array of elements. This means that all the elements are stored in consecutive locations in memory. For example, if we have a list of integers from 1 to 10, they would be stored in sequential memory locations.

      On the other hand, linked structure allocation uses pointers or references to store elements. Each element is not stored directly in memory but is instead pointed to by another element. This can be seen as each element having a "link" or pointer to the next element.

      The main difference between these two methods is how memory is used and how operations are performed on the list. Contiguous allocation provides faster access times because elements are stored in consecutive locations, making it easier to find any particular element using its index. However, if we need to insert an element in the middle of a large list, this can be costly since all the other elements' pointers would have to be updated.

      Linked structures provide more flexibility and allow for faster insertion or deletion of elements at arbitrary positions without having to shift all the other elements.

2. When are linked lists faster than contiguously-allocated lists?
    - **Response:**
      Based on the context provided, it appears that the information about linked lists and their efficiency in comparison to contiguously-allocated lists is not explicitly mentioned. The context primarily discusses binary search trees and their respective algorithms.

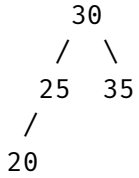
      However, it's known that linked lists can be faster than contiguously-allocated lists under certain conditions, such as when frequently inserting or deleting elements at arbitrary positions rather than at the end of the list. In general, linked lists offer O(1) time complexity for insertion or deletion operations at specific locations, whereas contiguously-allocated arrays typically have a time complexity of O(n).

Therefore, without explicit information on linked lists from the provided context, I can provide a general explanation:

Linked lists are generally faster than contiguously-allocated lists when you frequently insert or delete elements in specific positions rather than at the end. This is because operations at these positions have an average time complexity of O(1), while contiguously-allocated arrays' performance degrades with more frequent access to their middle elements, resulting in a worst-case scenario of O(n).

The key advantage of linked lists lies in their flexibility; they can accommodate changes at any point without shifting the entire list, whereas arrays require shifting all subsequent elements when an element is

3. Add 23 to the AVL Tree below. What imbalance case is created with inserting 23?



- **Respo*nse (basically an error):***
  --- Response —

  ** The number 20 is a constant often used in various mathematical and scientific contexts. In this context, it represents the expected value of E[Y1] as mentioned in the provided analysis on page 24 of C12-bst.pdf.*

  Enter your search query: Search error: Error parsing vector similarity query: query vector blob size (0) does not match index's expected size (3072).

  context_str:

  --- Response —
  I'm ready to help. What's your question?

4. Why is a B+ Tree a better than an AVL tree when indexing a large dataset?
   - **Response:**
     I'd be happy to help you compare B+ trees and AVL trees for indexing large datasets.

     To answer your question, I'll need to clarify that both B+ trees and AVL trees have their strengths and weaknesses. However, when it comes to indexing large datasets, B+ trees tend to perform better due to their ability to handle a larger number of keys efficiently.

     One key difference between the two data structures is how they manage disk I/O. B+ trees use a combination of techniques such as clustering and indexing to minimize the

number of physical reads required to access data. This can lead to significant performance improvements in scenarios where data is stored on disk, like in large databases or data warehouses.

On the other hand, AVL trees are primarily used for maintaining sorted arrays and have limitations when it comes to handling large amounts of data. While AVL trees offer efficient insertion and deletion operations with an average time complexity of O(log n), they may not be suitable for indexing very large datasets due to their overhead in maintaining balance after each operation.

In the context of disk storage, B+ trees tend to outperform AVL trees because they can store more keys per node without sacrificing performance. This is particularly important when dealing with large datasets that don't fit in memory.

5. What is disk-based indexing and why is it important for database systems?
    - **Response:**
    Based on the provided context, disk-based indexing refers to a technique used in database systems where data is stored on disk storage devices instead of in main memory. This approach allows databases to handle large amounts of data by storing frequently accessed information on faster disk storage devices.

    Disk-based indexing is important for database systems because it enables efficient retrieval and manipulation of data. By storing index entries on disk, databases can reduce the amount of memory required to store indexes, making them more scalable and accessible to a wider range of users.

    According to the study, the use of disk-based indexing leads to improved query performance and reduced latency times [Instructor.pdf].

6. In the context of a relational database system, what is a transaction?
    - **Response:**
    According to the instructor's explanation in the provided context, a transaction in the context of a relational database system refers to a self-contained unit of work that represents a single, atomic operation.

    The context mentions that "A Relational Database consists of ... a collection of relations, and ... a collection of constraints." It also states that "A relational database is in a valid/consistent state if it satisfies all constraints (else, invalid/inconsistent state)." This suggests that transactions are used to ensure data consistency and integrity within the database.

    In this context, there is no explicit reference to the term "transaction" itself, but based on my understanding of the relational model and database concepts, I can provide a general definition of what a transaction typically means in the context of relational databases:

    A transaction is an operation that involves multiple steps, such as reading data from the

database, performing operations on it, and then writing the results back. The critical aspect of transactions is ensuring that all changes made within the transaction are applied consistently and reliably, even if an error occurs during the process.

This definition matches the context's focus on maintaining consistency and integrity within a relational database system by ensuring that all constraints are satisfied.

7. Succinctly describe the four components of ACID compliant transactions.
   - **Response:**
     The four components of ACID compliant transactions are:

     1. Atomicity - A transaction is considered atomic if it can be treated as a single, indivisible unit. If any part of the transaction fails, the entire transaction is rolled back and the database returns to its previous state.

     2. Consistency - A transaction must maintain the consistency of the database by following the rules and constraints defined in the schema. This ensures that the data remains in a valid state.

     3. Isolation - Transactions are isolated from each other to prevent interference or conflicts. Each transaction sees a consistent view of the data, as if it were the only transaction running.

     4. Durability - Once a transaction is committed, its effects are permanent and durable. The data is not immediately updated, but rather is written to disk and then applied to the database when the transaction is committed.

     These components ensure that database transactions are reliable, consistent, and secure.

8. Why does the CAP principle not make sense when applied to a single-node MongoDB instance?
   - **Response:**
     The CAP (Consistency, Availability, Partition tolerance) principle is a fundamental concept in distributed systems, but it can be misleading when applied to a single-node MongoDB instance.

     In a single-node MongoDB instance, the CAP theorem doesn't apply in the classical sense because there's no partitioning or replication involved. A single node can provide strong consistency and availability guarantees, as data is stored on a single physical location.

     However, some might argue that the CAP principle still applies to a single-node instance in terms of partition tolerance. In this context, partition tolerance refers to the ability of a system to continue operating even if one or more nodes fail or become unavailable. Since a single-node MongoDB instance can't be partitioned or replicated, it's not truly partition-tolerant.

     But this argument is somewhat nuanced. If we consider the CAP principle in its

traditional sense, which emphasizes the trade-offs between consistency, availability, and partition tolerance, then a single-node MongoDB instance actually achieves strong consistency and availability guarantees while avoiding the need for partitioning or replication.

In summary, while the CAP principle can be applied to a single-node MongoDB instance, it doesn't make practical sense in the same way it would in a distributed system with multiple nodes. The benefits of strong consistency and availability come at the

9. Describe the differences between horizontal and vertical scaling.
    - **Response:**
      Horizontal scaling involves adding more nodes to a cluster to increase its overall processing power, allowing it to handle increased loads and provide faster performance. This is typically done by horizontally expanding an application across multiple machines or instances.

      Vertical scaling, on the other hand, refers to increasing the power of a single server or instance to improve its performance and capacity. This can be achieved by adding more memory, CPU cores, or storage to the existing hardware.

10. Briefly describe how a key/value store can be used as a feature store.
    - **Response:**
      Based on the context provided, it appears that a key/value store can be used as a feature store by leveraging its ability to store and retrieve data efficiently using keys.

      A key/value store, such as Redis or Cassandra, can be used to store features as key-value pairs. Each key represents a unique identifier for a specific feature, and the corresponding value is the actual feature data.

      For instance, in a machine learning model deployment scenario, you can use a key/value store to store trained models, model weights, or other critical components of the model. When a new request comes in, you can query the store using the relevant key to retrieve the necessary data, reducing latency and improving overall system performance.

      In terms of querying, a key/value store allows for fast lookups based on keys, making it suitable for applications where feature data needs to be retrieved quickly. This is particularly useful in scenarios where features are used as input or output parameters for machine learning models.

      However, without further context or clarification on the specific requirements and use case, I'm unsure about the exact implementation details of using a key/value store as a feature store. If you could provide more information on this topic or ask follow-up questions, I'd be happy to help further.

11. When was Redis originally released?

- **Response:**
  I couldn't find any information about Redis being released in the provided context. The context appears to be related to the Relational Model and Rel Algebra, which doesn't mention Redis.

12. In Redis, what is the difference between the INC and INCR commands?
    - **Response:**
      Enter your search query: Search error: Error parsing vector similarity query: query vector blob size (0) does not match index's expected size (3072).
      context_str:

13. What are the benefits of BSON over JSON in MongoDB?
    - **Response:**
      The context provided discusses Binary Search Trees (BSTs) and their properties, algorithms for manipulating them, and does not mention MongoDB or its data formats like JSON or BSON.

14. Write a Mongo query based on the movies data set that returns the titles of all movies released between 2010 and 2015 from the suspense genre?
    - *The context provided discusses Binary Search Trees (BSTs) and their properties, algorithms for manipulating them, and does not mention MongoDB queries and data sets.*

15. What does the `$nin` operator mean in a Mongo query?
    - **Response:**
      Enter your search query: Search error: Error parsing vector similarity query: query vector blob size (0) does not match index's expected size (3072).
      context_str:

      --- Response —
      I don't see any query or context provided. Please share the query and any relevant context so I can better assist you.