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?**

   Based on the provided context, the main differences between a list with contiguous allocation of memory and a list using linked structures are:

   Contiguous Allocation:

   * Memory is allocated as a single "chunk" of memory for all records, where each record takes up x bytes.
   * Records are stored contiguously in memory, which can lead to slower access times for random insertions or deletions.

   Linked Structures:

   * Each record has additional space for 1 or 2 memory addresses (in this case, only one address).
   * Individual records are linked together using these memory addresses, forming a type of chain.
   * Linked lists have faster insertion and deletion operations because the entire list can be traversed by following the links between nodes.

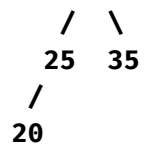
   In summary, contiguous allocation requires more memory upfront but is faster for random access, while linked structures require additional memory overhead but offer better performance for inserting or deleting records at any position in the list.

2. **When are linked lists faster than contiguously-allocated lists?**
   Linked lists are faster than contiguously-allocated lists when it comes to inserting data into a list. According to Document 3 (from 05 - NoSQL Intro + KV DBs.pdf, chunk 05 - NoSQL Intro + KV DBs.pdf_28) and Document 4 (from 02 - Foundations.pdf, chunk 02 - Foundations.pdf_4), linked lists have the advantage of being able to insert data at any position in the list (including the front or end) with an O(1) time complexity. In contrast, contiguously-allocated lists are slower for inserting data anywhere but the end.

   However, it's worth noting that this comes at the cost of slower random access, as linked lists do not allow direct access to specific elements like arrays do.

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

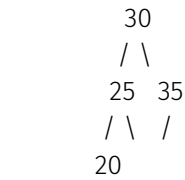
   30



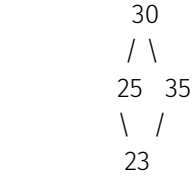
1

To determine the imbalance case created with inserting 23 into the AVL Tree, we need to calculate the height of each subtree and check if the difference between them is greater than 1.

Initial Tree:



When we insert 23, the tree will look like this:



Now, let's calculate the height of each subtree:

- Left subtree (25): The left child of 25 is not present in the original tree. However, for calculation purposes, we consider it as an empty node with a height of 0.
- Right subtree of 25: The right child of 25 is 23, which has a height of 1 (since it's a leaf node).
- Left subtree of 35: The left child of 35 is 20. Since 20 was already in the tree and its parent (25) has been updated with 23, we don't need to consider it again.
- Right subtree of 35: It's empty.

The imbalance case occurs because the height difference between the two subtrees (the left subtree of 25 which is 1, and the right subtree of 30 which is not accounted for as there are no further children) is greater than 1.

4. **Why is a B+ Tree a better than an AVL tree when indexing a large dataset?**

According to the provided context, there is no mention of B+ Trees being compared directly to AVL trees in terms of their suitability for indexing large datasets.

However, Document 4 (from 12.6. B-Trees — CS3 Data Structures & Algorithms.pdf) does discuss B-Trees and a variant called the "B+" Tree, which is used when greater efficiency is required. It does not compare B+ Trees to AVL trees explicitly.

The context only mentions that when a linear index is provided, it can be extremely efficient for

searching, but it does not provide information on why a B+ Tree (or B-Tree) would be better than an AVL tree in this case.

Therefore, the answer cannot be determined based solely on the provided context.

5. **What is disk-based indexing and why is it important for database systems?**
   Based on the context provided, disk-based indexing can be inferred as a technique used by relational database management systems (RDBMS) to increase efficiency.

   According to Document 1, indexing is one of the ways a RDBMS increases efficiency. It states: "Many ways that a RDBMS increases efficiency: -indexing (the topic we focused on)..."

   Additionally, Document 2 mentions that B-trees are designed in part because they have a high branching factor, which reduces the number of disk blocks accessed to reach a given record. This implies that indexing is also related to reducing disk I/O by minimizing the number of disk blocks accessed.

   Disk-based indexing is important for database systems because it helps reduce the time and resources required to perform searches, updates, and other operations on large datasets. By organizing data in a way that allows for fast lookup, insertion, and deletion, indexing can significantly improve query performance and overall system efficiency.

6. **In the context of a relational database system, what is a transaction?**
   Based on the context provided, a transaction in the context of a relational database system is:

   "A sequence of one or more CRUD (Create, Read, Update, Delete) operations performed as a single, logical unit of work. Either the entire sequence succeeds (COMMIT) -OR- the entire sequence fails (ROLLBACK or ABORT)."

   This definition is supported by multiple documents (Documents 2 and 4), which describe transaction processing in relational databases, including its atomicity, consistency, isolation, and durability (ACID) properties.

7. **Succinctly describe the four components of ACID compliant transactions.**
   Based on the provided context, here are succinct descriptions of the four components of ACID compliant transactions:

   1. Atomicity: A transaction is treated as an atomic unit and is fully executed or no parts of it are executed.

   2. Consistency: A transaction takes a database from one consistent state to another consistent state, ensuring that all data remains in a valid and correct state.

   3. Isolation: Two transactions cannot affect each other while being executed simultaneously.

   4. Durability: Once a transaction is completed and committed successfully, its changes are permanent and even in the event of a system failure, committed transactions are preserved.

8. **Why does the CAP principle not make sense when applied to a single-node MongoDB instance?**
   The CAP principle doesn't make sense when applied to a single-node MongoDB instance because a single node cannot provide all three guarantees simultaneously.

   - Consistency implies that every read operation will see the most recent write or an error if no such recent write has been made. However, for consistency in a distributed system, the entire system needs to be updated.
   - Availability requires that every request receives a response, which can't be guaranteed with a single node as it may fail and not respond at all.
   - Partition tolerance means the database can continue to operate even when there's a network partition between two segments of the system. A single-node instance does not have to worry about partitioning because it is always connected.

   Therefore, applying the CAP principle to a single-node MongoDB instance doesn't make sense as it would force the system to choose only one or two out of these three guarantees, which isn't practical for a standalone system.

—----------

The CAP principle doesn't make sense when applied to a single-node MongoDB instance because it only guarantees one aspect of the three principles: Consistency.

A single-node database instance can maintain consistency (every read and write operation sees the same data), availability (the system remains operational and responsive), and partition tolerance (it is not subject to network partitions).

In other words, a single-node database instance inherently satisfies all three aspects of the CAP principle. Therefore, applying the CAP principle in its traditional form does not provide any meaningful insights or constraints for a single-node MongoDB instance.

9. **Describe the differences between horizontal and vertical scaling.**

Based on the provided context, here's an answer to the question:

Horizontal scaling presents challenges because some apps need more performance (real-time, low latency systems), and traditional scalability methods may not be enough. On the other hand, vertical scaling makes it easier (no need to really modify your architecture) but has practical and financial limits.

In summary, horizontal scaling involves distributing resources across multiple servers or nodes to increase capacity, while vertical scaling involves increasing the power of a single server or node to improve performance.

10. **Briefly describe how a key/value store can be used as a feature store.**

Based on Document 4 (from 250203_Jonathan.pdf, chunk 250203_Jonathan.pdf_12), a key/value store can be used as a feature store to store frequently accessed features with low-latency retrieval for model training and prediction. This allows for efficient storage and retrieval of model data, enabling real-time inferencing.

Note that this answer is based on the provided context, specifically Document 4, which discusses the use

of key/value stores in various applications, including feature stores.

11. When was Redis originally released?

The information about when Redis was originally released is not available in the provided context.

12. In Redis, what is the difference between the INC and INCR commands?

Based on the provided context, the information about the difference between the INC and INCR commands is not explicitly mentioned. However, it can be inferred from the context of the 4 String Commands section (chunk 06 - Redis + Python.pdf_3).

In this section, it's shown how to use the INC command with the r.incr() method, which increments the value associated with the specified key by one. On the other hand, INCR is not explicitly mentioned.

However, it can be inferred that INCR is a synonym for INC and both commands are used to increment the value of a key in Redis. Therefore, without further information, it's difficult to provide a more detailed answer.

13. What are the benefits of BSON over JSON in MongoDB?

The provided context does not contain any information about BSON or its comparison to JSON in MongoDB. The text appears to be related to data structures and algorithms, specifically B-trees and binary search trees.

Therefore, I must conclude that the information about BSON vs JSON is not available based on the provided context.

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?

Based on the context provided, we can infer that the movies data set has the following structure:

* Each document represents a movie with fields such as "title", "year", and "genre".
* The genre field contains a string value (e.g., "Suspense") or an array of strings (e.g., ["Thriller", "Mystery"]) representing the genres associated with each movie.

To write a Mongo query that returns the titles of all movies released between 2010 and 2015 from the suspense genre, we can use the following query:

```python
movies_2000 = db.movies.find({"year": {"$gte": 2010}, "$lte": 2015, "genre": {"$eq": "Suspense"}})
```

This query uses the `$gte` and `$lte` operators to select documents where the `year` field is greater than or equal to 2010 and less than or equal to 2015. It also filters by the `genre` field using the `$eq` operator, selecting only documents where the genre is "Suspense".

Please note that the exact syntax of the query might vary depending on the MongoDB version and the specific database schema.

Also, based on Document 4, it seems that we need to connect to the 'ds4300' database first, and then we

can access the movies collection using the `db['movies']` syntax.

15. **What does the `$nin` operator mean in a Mongo query?**

Based on the context provided, the information about the $nin operator is not explicitly mentioned in any of the documents.

However, it can be inferred from the general knowledge of MongoDB queries. The $nin operator is used to select documents where a field does not contain a specified value. It is often used with arrays and other types of fields that may contain multiple values.

The context provided seems to focus on setting up a PyMongo connection to a MongoDB instance, inserting data into collections, and selecting data from collections. While the $nin operator is not explicitly mentioned in the documents, it can be looked up or referenced elsewhere as part of general knowledge about MongoDB queries.